# UNIVERSITY OF LONDON THESIS

Degree PhD Year 2007 Name of Author ANDREW ROSS DINGWALL-SMITH

Department of Computer Science
University College London
University of London

# Run-Time Monitoring of Goal-Oriented Requirements Specifications

Andrew Ross Dingwall-Smith

Submitted for the degree of Doctor of Philosophy
at the University of London

June 2006





ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

I, Andrew Dingwall-Smith, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Signed: Date:

# Abstract

The environment in which a software system operates is as important to the correct operation of the system as the software itself. Most software development involves making assumptions about the environment in which the resulting system will operate. These assumptions may cease to be valid if the environment changes, causing the system to fail to operate correctly.

One solution to this problem is to use run-time requirements monitoring to determine, as a system operates, whether it is satisfying the requirements specified for it and to take action to rectify these problems.

This thesis describes work that has been carried out in the area of run-time requirements monitoring. A framework has been developed for monitoring requirements which are formally specified using temporal logic and the KAOS goal-oriented requirements specification language. The framework uses AspectJ to instrument the monitored system so that events are emitted which are used to determine whether the monitored system satisfies the requirements specification. The framework also provides a language which can specify a mapping between requirements and implementation which can be used to generate instrumentation code.

The monitoring framework supports monitoring of soft goals by allowing the formal specification of metrics which can be used to determine whether soft goals are in fact being satisfied.

These contributions are validated using a workforce scheduling system as a case study. This is a real world system and the requirements monitored were those considered useful by the developers of the system. The case study shows that the monitoring framework can be used to instrument a system to monitor hard and soft goals and that those goals can be monitored with reasonable performance overhead. Goal failures due to changes in the environment can be detected using the information supplied by the monitoring framework.

# Acknowledgements

I would especially like to thank my supervisor Anthony Finkelstein for his invaluable ideas, support, encouragement and guidance. I would like to thank Cecilia Mascolo, Wolfgang Emmerich and David Rosenblum for their help, advice and suggestions. I also wish to thank David Lesaint from BT for making the case study possible, for organising my time working at Adastral Park and for his help and advice.

Also many thanks to everyone from the software systems engineering group, including: Andy Maule, Ben Butchart, Ben Chen, Bozena, Bruno, Carina, Christian, Clovis, Costin, Daniel, Genaina, Ilias, James, Javier, Jidtima, Licia, Martin, Mirco, Miro, Nima, Panu, Rami, Stefanos, Torsten, Vito and Vladimir, who have all made my time at UCL so enjoyable.

I particularly want to thank my parents. Without their support, getting this far would have been impossible.

Finally, I want to gratefully thank BT and the EPSRC for providing sponsorship for this work.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Context

The current trends in software engineering are considered in[Finkelstein 00] where the authors state that software systems are becoming ever more complex and that as a result, the way software systems are developed is also changing. Fewer software systems are built from the ground up. Many systems are instead extensions of existing systems or are built using *off-the-shelf* components which are configured to interact with each other.

The demands of the organisations which utilise these systems are also changing and becoming more demanding. Business processes change frequently which can affect both the requirements of a software system and the environment which an existing system operates in.

The ability of a system to satisfy its requirements is dependent on both the system itself and the environment in which it operates[Jackson 95]. This means that a system that has been deployed successfully can still subsequently fail to satisfy requirements for two reasons; the system itself can evolve after deployment or the environment in which the software operates can change.

The demands of business for rapid and cost effective change can make it difficult to ensure that systems continue to satisfy requirements as they evolve. Good software engineering practices can help but these approaches can be slow and expensive to implement [Bennett 00].

Change in the environment in which a software system operates can be an even larger problem. Software systems are built based on assumptions about the environment in which they will operate. If the system is required to operate in environments subject to frequent changes which cannot be easily anticipated, it becomes a particular problem to ensure that no unwarranted assumptions about the environment are made. If changes in the environment cannot be anticipated during the design process then the behaviour of the system cannot be guaranteed to satisfy the requirements of the system.

These problems create the need for run-time requirements monitoring which is the subject of this thesis.

## 1.2 Run-Time Monitoring

Monitoring encompasses the gathering of information from a system about its execution, the analysis of that information and the final presentation of the results of the analysis. The results may be presented either to a human actor or to a computer system which is responsible for handling them in some way, such as modifying the system in

response to the information obtained.

Run-time monitoring refers specifically to monitoring in which all three stages happen at run time, while the system is executing. In particular the analysis of gathered information happens at run time in an incremental manner as information is received by the monitor.

Requirements monitoring is a form of monitoring which seeks to determine if a requirements specification is satisfied by the execution of a system. If this monitoring happens after deployment of the monitored system in its operating environment, as part of the normal operation of the system, then it is referred to as on-line monitoring. If requirements monitoring happens during the testing phase, to verify that the requirements specification has been satisfied, then it is referred to as run-time verification.

Requirements monitoring can be used to detect the failure of a system to satisfy its requirements due to changes in the implementation of the system or due to changes in the environment that the system operates in. It is a pragmatic approach to detecting problems in a system as failures will only be detected after they occur, allowing remedial action to be taken and changes made to prevent future re-occurrence.

In the case of changes to a system, it may be cheaper to monitor for requirement failures as and when they occur rather than to verify that all requirements are still satisfied after changes have occurred. This will be particularly true if changes to the system occur frequently and for requirements which are not critical to the operation of the system.

For systems which operate in dynamic environments, the case for requirements monitoring is stronger still. It is difficult and expensive to anticipate all changes which could occur to the environment which could affect the system and to design the system to deal with these changes.

## 1.3 Problem Description

The aim of this work is to provide a monitoring framework which allows monitors to be created which, at run time, determine whether the behaviour of the system being monitored satisfies a requirements specification. The monitors should provide information which helps to identify what changes need to be made to the system so that its behaviour will satisfy the requirements in future. The monitoring framework is intended to make as few assumptions as possible about the architecture of the monitored system so that it should be possible to monitor any system written using the Java programming language.

Run-time requirements monitoring involves a number of problems. It is necessary to decide on a suitable formalism for specifying the requirements against which the system is monitored. The system being monitored needs to be instrumented so that it will send information about the execution of the monitored system to the monitors. The monitors themselves need to be able to determine whether the requirements specification is satisfied from the information supplied by the instrumentation. There are additional complications which arise if the system being monitored is a distributed system such as ensuring that instrumentation information is processed in the correct order. It is further necessary to ensure that the monitoring process does not impact the performance of the system being monitored to an unreasonable degree.

Determining whether the system has satisfied the requirements from the information obtained by the instrumentation requires some way of interpreting information


Figure 1.1: Basic monitoring system.

relating to the execution of the system in terms of the requirements specification. This can be thought of as a translation process in which information and events gathered by the instrumentation are translated into information and events described in terms of the requirements model. Depending on how closely related the requirements specification is to the actual implementation, this translation process can be trivial or very complex.

The basic structure of a monitoring system is shown in figure 1.1. A monitoring system consists of a monitored or target system and a monitor server. This view of a monitoring system is quite simplified. The target system may be made up of distributed components. The monitor server will generally have a number of different monitors for monitoring different requirements. These monitors may also be distributed although a review of the literature shows that this is not a widely adopted approach. The translation between implementation and requirements level information may take place in either the instrumentation part of the target system or in the monitor server.

Execution of the monitor server can either be synchronous or asynchronous with the target system. Asynchronous operation is normally used when the monitor server operates on a different machine from the target system. Communication can either be bi-directional or single directional, from the target system to the monitor server.

The target is instrumented in some way so as to allow the monitor to gather information about the execution of the target. The instrumentation may actively send information to the monitor or it may wait until the monitor server requests information. Instrumentation is sometimes thought of as being made up of *sensors* with each sensor being responsible for gathering a particular type of information from the target system.

## 1.4 Scope and Assumptions

There are many reasons to use run-time monitoring. In [Schroeder 95] Schroeder describes seven areas in which run-time monitoring is used. The focus of this thesis is requirements monitoring, which most obviously comes under the heading of *correctness checking* in Schroeder's scheme. Schroeder defines correctness checking as

> ...monitoring an application to ensure consistency with a formal specification.

Requirements monitoring is correctness checking where the formal specification being checked is the requirements specification. In fact this definition is rather narrow as several other areas of monitoring can be related to requirements monitoring.

Of the other areas identified by Schroeder, *control* and *performance enhancement* involve modifying the system in response to information obtained through monitoring. This thesis only deals with checking requirements. It does not consider using the results of requirements checking to modify the behaviour of the target system.

*Performance evaluation* is related to the work in this thesis but is not a core concern. This work tries to determine conformance with specific requirements which may include performance related requirements but does not involve detailed performance

analysis. This framework assumes that high performance is not a crucial issue as the performance overhead is not necessarily small enough for these applications. Systems with looser performance constraints can be monitored using the framework.

*Debugging and testing* refers to monitoring activities which take place during testing. This work focuses on monitoring which takes place after deployment as part of the normal operation of a system. Debugging and testing techniques do however have relevance to requirements monitoring, particularly techniques which use high level debugging to abstract the information collected.

*Dependability* and *security* monitoring are relevant to the work in this thesis. Such concerns can be monitored using the framework described in this thesis if these concerns can be expressed formally in the requirements specification language. The work on monitoring soft goals may be useful in monitoring these types of requirements.

Two important considerations when monitoring a system is whether the results of monitoring are correct and whether the operation of the system is affected by the process of monitoring. In the monitoring framework described here, these considerations are largely left to the developer of instrumentation code. The monitor server will produce correct results given correct input but it is up to the instrumentation developer to provide the correct input. Similarly, there is nothing to stop the instrumentation developer functionally altering the behaviour of the monitored system although in practice this problem has been relatively easy to avoid.

## 1.5 Contributions

The goal of this thesis is to investigate run-time requirements monitoring and to provide a framework which assists developers in implementing run-time requirements monitoring. This section describes the contributions of this thesis.

### 1.5.1 Monitoring Framework

This thesis provides a framework for monitoring goal-oriented requirements specifications specified using temporal logic. Aspect-oriented programming techniques are used to instrument the target system so it can be monitored. Goal failures and the results of monitoring soft goals are displayed graphically to the user of the framework. The framework has been implemented, primarily using the Java programming language.

The notable features of the framework are:

**Goal-Oriented Specification** The monitoring framework uses KAOS, a goal-oriented requirements specification language, for specifying requirements. This language includes formal specification of hard goals in temporal logic which allows the goals to be monitored.

An existing requirements specification language was chosen so that the monitoring framework can more easily be integrated into the development process. The requirements specification language needs to take account of all the needs of the developers, not just the need for a requirements specification that can be monitored, so the needs of monitoring should not determine the requirements specification language. It is however necessary to have a formal specification to allow requirements monitoring to take place. KAOS is particularly useful here because it combines informal specification with formal specification which can be used where necessary. The developer could, if desired, use informal specifica-

tion in the development of the system and formally define only those goals which need to be monitored.

**Instrumentation Mechanism** An important part of the framework is the method for instrumenting the target system to emit events which the monitor can then use to check the requirements are satisfied. The instrumentation carries out two tasks: gathering data on the execution of the system and translating that implementation level data into requirements level events which can be understood by the monitor. The approach is one that is generally applicable to any Java system rather than to systems with specific architectures or which operate in specific domains.

The framework uses AspectJ, an aspect-oriented extension to the Java Programming language, to instrument the target system. This allows the target system to be instrumented in a non-invasive manner, without modifying the source code of the system. AspectJ provides a powerful set of language constructs for selecting execution events in the target system. The aspects which implement the instrumentation are also effective at encapsulating the relationship between implementation level events and requirements level events.

**Generation of Instrumentation From Mapping** In addition to being able to specify instrumentation in AspectJ, the framework also allows the instrumentation to be specified by defining a mapping between implementation and requirement level events. This mapping is used to automatically generate instrumentation aspects. While not as powerful as defining instrumentation using AspectJ, this approach makes it easier to define instrumentation for simple cases and makes the mapping between implementation and requirements levels explicit.

**Architecture** Suitable architectures for monitoring systems are examined. This includes considering problems such as distribution and performance. A general monitoring architecture has been developed in which the monitored system sends events, to a monitor server, which indicate changes to a requirements model which represents the monitored system at the instance level. Monitors responsible for checking particular goals then use the requirements instance model to detect violations and monitors for soft goal metrics use it to calculate a value for the metric. The framework can use one of two implementations for the requirements instance model. The first uses a database to store an instance of the KAOS object model while the second uses in memory data structures. Monitors can be constructed automatically from the requirements specification as they only have to communicate with the requirements instance model.

**Specification of Soft Goal Metrics** While KAOS allows hard goals to be formally defined, it does not offer any assistance in specifying soft goals in such a way that they can be monitored. Soft goals are likely to be of importance to stakeholders in the target system so it is important to monitor them. A method for formally specifying metrics which allow satisfaction of soft goals to be determined has been developed to allow monitoring to take place.

The approach used in the monitoring framework is to define metrics which are associated with soft goals. These metrics should capture the degree to which a goal is satisfied. The monitoring framework provides a language for specifying

soft-goal metrics at the requirements level. These metrics are defined using the same KAOS goal and object models which are used to specify the hard goals for the system.

**Graphical Display of Monitoring Results** The results of monitoring are graphically displayed to the user of the framework. Hard goals are displayed using the goal tree from the KAOS requirements model of the target system to indicate which goal has failed. Soft goal metrics are displayed using standardised gauges which are configurable by the developer so that users of the framework can determine at run time whether the associated soft goals have been satisfied. Developers can also use a standard interface and plug-in mechanism to implement new types of gauges.

These features combined and implemented make this monitoring framework a novel contribution to run-time requirements monitoring. The framework combines AspectJ instrumentation with KAOS requirements specification and builds on this foundation to allow monitoring of soft goals metrics.

### 1.5.2 Evaluation of Results

The contribution is validated using a substantial case study in which requirements are monitored for a work force scheduling system. This system was chosen because it is a real system which is reasonably large and complex and because there are requirements which the developers are interested in monitoring.

The case study aims to determine whether the monitoring framework meets the following criteria:

**Performance** Monitoring a system will inevitably result in the monitored system incurring some performance overhead. The monitoring framework is required to operate after deployment of the monitored system, so that failures due to incorrect assumptions about the environment can be detected. This means that the performance overhead must be as small as possible as any overhead will affect the actual operation of the system.

**Ease of Specification** It should be as easy as possible to specify what the requirements are that need to be monitored by the monitoring framework. For hard goals this is achieved using the KAOS goal-oriented requirements engineering language. For soft goals, a specialised language for formally specifying soft goal metrics is used and the ease of use of this language is evaluated.

**Ease of Development** It should be as easy as possible to implement run-time requirements monitoring for a system. As the monitor server is able to interpret the requirements specification directly, only the development of instrumentation code has to be evaluated here.

## 1.6 Thesis Outline

**Chapter 2 — Literature Review** In this chapter the literature related to run-time monitoring, particularly the literature most closely related to the contributions of this thesis, is reviewed.

**Chapter 3 — Background** This chapter describes background information which is useful for understanding the rest of the thesis. Goal-oriented requirements engineering using KAOS is briefly explained. AspectJ, which is used to develop instrumentation, is also described.

A peer-to-peer file sharing program, Limewire, a Gnutella client, was used as a test bed application for this work and examples from this system appear throughout this thesis this example is introduced here.

**Chapter 4 — Monitoring Temporal Logic Goals** This chapter describes the monitoring framework for hard goals. This includes a discussion of the design decisions which were made in the creation of the framework, how goals are checked and how the target system is instrumented.

**Chapter 5 — Monitoring Soft Goals** The extension of the monitoring framework to monitor soft-goal metrics is described in this chapter. This chapter discusses how metrics associated with soft goals are specified, how those metrics are evaluated at run time and how the results of monitoring the metrics are displayed to the user of the monitoring framework.

**Chapter 6 — NGDS Case Study** A case study was conducted using a workforce scheduling system. This system is a real application designed by BTexact. The implementation of monitoring for the case study and the results which were obtained are described in this chapter.

**Chapter 7 — Conclusions and Future Work** Finally, overall conclusions are discussed and suggestions of future work are made.

**Appendix A — Limewire Formal Specification** Presents the complete formal KAOS specification for the Limewire system.

**Appendix B — Mapping Language DTD** Presents the XML document type definition which defines the syntax of the mapping language used to generate instrumentation code.

**Appendix C — Goal Instance Metric Query Generation** The XSLT code used to generate SQL queries which evaluate soft goal metrics from XML specifications of those metrics.

# Chapter 2

# Literature Review

This chapter describes work related to the contributions made by this thesis in the area of run-time requirements monitoring. Work on run-time monitoring is found in a variety of locations, often associated with applications of monitoring rather than the topic of run-time monitoring itself. This means that locating related work can be difficult. While every effort has been made to ensure this is a comprehensive review it is possible that some work could have been overlooked.

In section 2.1, work is described in areas which are related to run-time monitoring but which generally precede the bulk of the work on run-time monitoring. These areas are debugging, logging and assertions. These areas are relevant because they overlap with run-time monitoring to some extent and the boundary between these types of approaches and run-time monitoring is not always clear. Generally debugging, logging and assertions tend towards low-level implementation details. Run-time monitoring uses higher level, abstract specifications. This sections does not provide a comprehensive review of these areas but instead presents work which is most relevant to the area of run-time monitoring.

The rest of the chapter reviews literature which is relevant to particular facets of run-time monitoring which are relevant to the work in this thesis. Section 2.2 looks at approaches to specifying monitorable requirements. Section 2.3 reviews approaches to instrumenting a system so that it can be monitored. Section 2.4 looks at different architectures for monitoring and approaches to monitoring distributed systems. Section 2.5 looks at work on specifying and monitoring soft goals. Section 2.6 looks at approaches to display of monitoring results to the user of the monitoring framework.

## 2.1 Areas Related to Run-time Monitoring

### 2.1.1 Debugging

An area which has been researched for several decades and is now in wide spread use is debugging. Debugging involves display of information at run time so that developers can understand the behaviour of a system and discover errors in the implementation. It may also be possible to modify the state of the system at run time using the debugger. Debuggers now form a standard part of most programming integrated development environments.

Debugging has several limitations which, while they do not hinder the aims of debugging, make the approach unsuitable for run-time requirements monitoring. Debugging provides complete information on the execution of a system. The instrumentation to obtain this information is normally inserted as part of compilation so that the

debugger is informed of every change of state in the system. Providing such a level of information implies a significant performance penalty which prevents the use of the debugger in a deployed system.

Debuggers normally provide low-level implementation details. In particular, they usually provide the current point in the execution of the program and the value of all variables in the current scope. In most development environments, no attempt is made to link events which are reported at the implementation level to any higher level specification.

A further development of debugging is to try and provide more abstract views of a system. This is of particular interest in distributed systems where understanding execution is considerably more difficult than single process systems.

In [Bates 83] a distributed systems' behaviour is viewed as a stream of events. Abstraction is obtained by expressing higher level events as sequences of low-level events and by filtering out certain low-level events based on attributes of those events. This is important as it represent an attempt to move from implementation level information to higher level information although the developer is left to determine what higher level events are of interest. This is a similar idea to the translation of implementation level events to requirements level events in the monitoring framework although the actual details of the translation are very different.

A further development of this approach is to provide visual displays of the execution of a distributed system as in the POET system[Kunz 97]. This approach also uses abstract events which are described by combining primitive debugging events. This is important because visual displays of primitive debugging events are likely to be too complex to interpret meaningfully. The abstract events and their relative order are displayed, along with which execution trace or traces the events belong to. This idea of abstract events is important in the development of the monitoring framework in this thesis, although in POET abstract events are generated by combining primitive events rather than a more general translation. The POET system also attempts to provide graphical feedback to the user although that feedback has a different form and a different aim than that provided by the work in this thesis.

### 2.1.2 Logging

Another area related to monitoring is logging which refers to the process of instrumenting a system to output events. The term normally suggests that these events are output to a file for later analysis although it is sometimes used to refer to systems in which log files are analysed at run time, which is one way of implementing run-time monitoring.

The log file analyser described in [Qiao 99] is used to find errors in parallel programs. The monitored system is built using MPI(Message Passing Interface), a standardised library for message passing. Each process produces its own log file of communication events. These log files are analysed after execution completes to detect errors.

Instrumentation is easy to implement here because there is a standardised communication method provided by the MPI library. Instrumentation is simply a matter of adding a wrapper around the library which writes to a log file when communication functions are invoked. The log file analyser finds common communication problems, rather than checking the log files against a specification. It also does not operate at run time.

In [Andrews 98] log files are analysed to determine whether they are compatible

with formally specified requirements. The requirements are specified by describing state machines. These descriptions are compiled into an executable program which can be run on the log file to discover if the state machines enter illegal states. If they do then a violation is detected.

The log file analyser developed as part of this work can be run concurrently with the execution of the monitored system. This effectively makes it a run-time monitoring system. While the approach uses a separate formal specification, the specification is still quite low-level and closely related to the design and implementation. The system to be monitored has to be instrumented manually.

The most basic logging systems are simply a way to record information about the execution of a system. What is interesting about them is that they often involve some form of abstraction so that events are not described simply in terms of function execution and similar implementation level events. More complex systems, such as those described above, begin to resemble run-time monitoring systems as they check the output logs against some form of specification, possibly at run time. In general, using a log file is not a particularly efficient or well structured way of implementing complex monitoring so other approaches are preferred.

### 2.1.3 Assertions

Also related to monitoring are assertions. Assertions are statements that something should be true at a particular point in the execution of a system. They are embedded directly in the code at the point where they apply although they are not part of the logic of the system itself. They are checked at run time and some action occurs if they are not true. The term suggests that the assertions are evaluated by the same process and thread as the code which they apply to.

Assertions can be part of a programming language, as with the Java *assert* keyword introduced in Java 1.4. In this case, the normal syntax of the language is used to specify the boolean condition to be satisfied. Alternatively, assertions can be written in their own specialised language and embedded in comments. Often such comments are transformed to code in the language of the source by a pre-processor.

Regardless of whether assertions are written in the language of the source code or a different language, they are are distributed throughout the source code and their location in the source code determines what part of the execution the assertion applies to. They do not form a separate specification so they are not a good way to monitor an existing requirements specification such as a KAOS specification.

Assertions are normally used to enforce parts of the system design which are not otherwise enforced by the programming language. For example, assertions can be used to enforce pre- and post-conditions of methods in the Java programming language. The Java type system has no way to declare that an object parameter should be non-null or that an integer parameter should be less than ten but a method may be written assuming these conditions to be true. Assertions allow these conditions to be checked at run time (as opposed to compile time, which is when type errors are detected in the Java language).

An example of this approach is [Rosenblum 95] where assertions are added to C code. The assertions are specified inside special comments, although they use C syntax for expressions. The assertions can be used to check pre- and post- conditions, function return values and intermediate states of function bodies. The comments are pre-processed to generate standard C code. The problems which are uncovered by these

assertions are typical programming errors.

Another common situation is that methods of classes need to be called in a particular order. For example, an initialisation method may need to be called before any other method can be called on an object. Assertions can help deal with such situations. This is the type of problem which can be handled by the Temporal Rover tool[Drusinsky 00]. Assertions are specified in temporal logic and then transformed into code which can then be compiled. This has some similarities to the work in this thesis in that temporal logic specification is used but the specification is inserted directly into the code as annotations and are checked synchronously with the code.

In general, assertion systems are intended to discover low level problems in the implementation. Typically these are inconsistencies between the design of the system and its implementation. The work in this thesis aims to discover failures due to erroneous assumptions about the environment which requires considering failures in terms of early stage requirements.

## 2.2 Specification of Monitorable Requirements

A requirements monitor needs to be told what requirements should be monitored in an unambiguous way. This allows the monitor to process information from the monitored system to determine whether that information is consistent with the requirements specification. This section looks at different approaches to specification of requirements for monitoring which have appeared in related work.

The requirements can be specified implicitly by writing monitoring code specifically for each monitored requirement, as in [Dasgupta 86]. This code receives information from the monitored system and uses it to determine whether a particular requirement is satisfied. This is an approach which is relatively quick to implement in the absence of a monitoring framework and is suitable for monitoring small numbers of requirements which can be translated directly from their specifications to code by the developer. This approach is not used in this thesis because it requires rewriting the monitoring logic for each new requirement, leading to both greater work in the long term and a greater probability of incorrect monitoring code.

It is more common when carrying out requirements monitoring to formally specify requirements in a suitable language. This allows monitoring code to be written once which takes a formal requirements specification as input and then checks information from the monitored system against that specification.

If formal specification of requirements is not required during development of the monitored system then the formal requirements specification can be tailored to the needs of monitoring. One way this can be done is by using informal requirements during development of the system and then formally specifying the requirements after implementation so that the requirements can be monitored. This allows the requirements to be formally specified in terms of properties of the implementation of the system, which would clearly not be possible until the system was implemented. This approach makes instrumentation easier as the necessary code can be automatically determined from the requirements specification. The formalised specification is not really a requirements specification though as it refers to the implementation and so could not be used as part of the development of a system. The specification is instead an expression of requirements or constraints related to those requirements at the implementation level.

An example of this type of specification is found in [Liao 92] where the requirements specification is written using predicates which are true at particular points in the execution of the target system or are true between two points in the execution. These events can also have parameters such as the value of variables at the time an event occurs. The requirements are then written using a formal language (the example given is domain relational calculus) to define more complex predicates from these basic predicates.

In the work by Sanker and Mandal [Sankar 93] Ada programs are annotated with formal requirements specifications which use a combination of normal Ada syntax and extensions based on first order logic. As the requirements are written as annotations to the source code, and the interpretation of those requirements depend on their position with respect to the source code, this type of formal requirements specification depends on the implementation of the target system.

Both these approaches depend on the developer taking a requirements specification and, after the target system has been implemented, translating the specification into the form used by the monitoring framework. This approach is not used in this thesis because it is preferable to be able to use a requirements specification directly in the monitoring framework.

In [Chodrow 91] the requirements specification is independent of the implementation, referring instead to labels which are added to the source code. This means that the requirements specification can be written formally before the system is implemented as the labels can be added to the code during or after implementation. The specification language itself uses real-time logic. The separation of implementation and monitored specification is a good one and is the general approach used in this thesis. The actual mechanism of referring to labels which are used to annotate the source code is not used because it involves modifying the source code and because it is quite limiting in terms of what relationships between implementation and requirements can be expressed.

Monitoring frameworks in which the monitored requirements specifications are independent of the implementation of the monitored system also include the Java-MaC framework [Kim 01], the Java Path Explorer framework [Havelund 01] and the DynaMICs framework [Gates 01]. Java-MaC uses a specification language which focuses on specifying safety properties. Java Path Explorer uses a more general requirements specification which uses linear temporal logic. DynaMICs monitors constraints which are extracted from the requirements specification and includes extensive support for obtaining those constraints. The work in this thesis builds upon these approaches but instead of using a specialised specification language selected with monitoring in mind, it uses an existing requirements engineering language.

In [Chen 03] the authors describe their approach to what they call *monitoring-oriented programming*. Their monitoring framework allows developers to choose the logic they wish to use to formalise requirements. The developer specifies rewriting rules which specify how the logic should be interpreted by the monitor at run time. Allowing developers to specify the formal language they wish to use in addition to the specification is a powerful approach but is not used in this thesis as it was felt that concentrating on a single specification language would better allow progress to be made in other areas such as instrumentation.

In [Skene 04a] service level agreements(SLAs) are used as the requirements specification which is to be monitored. This is a specialised application of monitoring which

is more focused on checking that a service a user is provided with meets that user's requirements rather than allowing a developer to check their own system. The language used to define the SLA which is monitored is named SLAng and is described in [Skene 04b]. The language uses models using classes relationships and attributes to define the domain to which the SLA applies and uses the Object Constraint Language to define the constraints placed on the model by the SLA. This is a specialised approach which is able to effectively monitor particular types of system is not applicable for more general cases.

Of particular relevance to the work in this thesis are monitoring frameworks which monitor requirements specified using the KAOS goal-oriented requirements engineering language [Fickas 95, Feather 98, Robinson 03]. Unlike the requirements specifications used in the works described previously, KAOS is a complete requirements engineering language with an associated visual language, methodology and formal specification. It is intended to be used in the development of systems, rather than as a language for monitoring, but it is still allows formal specifications which are quite suitable for monitoring. One effect of this is that the language is quite complex compared to most formalisms chosen specifically for monitoring.

The work in this thesis adopts the same approach as the other work mentioned in using KAOS to specify monitorable requirements but builds on this work by supporting instrumentation of the monitored system to a greater extent and by allowing metrics for soft goals to be specified using the KAOS specification.

KAOS has several attributes which make it a good specification language for runtime monitoring. Goals are identified at an early stage in the requirements engineering process and provide a very abstract view of the system. Particular attention is given to the interaction between the system and its environment. This is described in terms of goals and agents which are responsible for achieving them.

KAOS incorporates formal specification using linear temporal logic which is suitable for monitoring although such formal specification is considered optional. Even if requirements are not defined formally during development, KAOS still makes it easy to add formal specification for requirements which need it by providing a framework into which formal specifications fit easily.

Monitoring a requirements language which is suitable for use during development is desirable as it is not necessary to then write a requirements specification specifically for monitoring. This has the potential to lower the cost of using requirements monitoring for developers.

## 2.3 Instrumentation

Instrumentation is code which emits events about the execution of a computer system. Depending on the implementation, this information can be used to monitor the execution of a system in a separate thread, process or machine. This differs from approaches which use assertions, which are checked in immediately in the same thread of execution as the event which triggers the check. Some approaches to instrumentation are reviewed here.

In CARISMA[Capra 03], the physical properties of mobile devices such as battery life remaining, bandwidth available, display properties and so on are monitored. This is possible because the devices monitored come with the capability for such monitoring. In this case, instrumentation is not necessary because the information needed for the

sort of requirements being monitored is already made available. This approach is particularly powerful as it allows some aspect of the environment to be measured directly using the capabilities of the hardware. This approach is, however, only really relevant in particular domains such as mobile devices and robotic control[Peters 02].

There has been some work on monitoring which makes use of *object-oriented* operating systems [Dasgupta 86, Snodgrass 88]. Such operating systems represent system resources, such as data and program files, as objects which code can be added to. All objects created by a program are created through the operating system. The operating system itself provides the support for passive, but not active, instrumentation by allowing objects to be queried for their status. It is obviously not usable with operating systems other than object-oriented operating systems which provide the instrumentation support. It is however a very simple and powerful approach when such operating systems are used. The lower level instrumentation at the operating system level, rather than the application level, may also be better able to detect problems which occur as the instrumentation is closer to the boundary between the system and its environment. This approach is not really relevant to the sort of systems which are currently developed as object-oriented operating systems are not in general use.

A similar approach uses a *law-governed architecture*[Minsky 96] which controls the interactions between classes. This architecture can prevent classes from making calls which are not explicitly permitted or to perform operations whenever a call is made. The system is instrumented by defining rules which require a monitor to be informed when calls are made which need to be monitored. An important advantage of this approach is that it can guarantee that instrumentation is side-effect free meaning that the monitored system will not be modified by the instrumentation. This is done by writing a rule which prevents the monitor code from calling functions which create objects or modify variables. This approach has similar advantages and disadvantages as the use of object-oriented operating systems to instrument the system. The approach makes instrumentation quite straightforward but it requires the use of a law-governed architecture which is not something which is widely used in general computing.

Several authors have published work relating to run-time monitoring of web-services[Robinson 03, Mahbub 04, Lazovik 04] and in the Cremona architecture [Ludwig 04]. Instrumenting web services tends to be easier than a solution for instrumenting any type of system as assumptions can be made as the instrumentation can be built into the architecture that supports the web services.

For example, in the work by Mahbub and Spanoudakis, the monitored system is made up of web services which are composed into a system using BPEL4WS to specify a business process. An execution engine serves as a central controller for the system and contains the instrumentation necessary to monitor requirements which are expressed at the same level of abstraction as the BPEL4WS specification.

The obvious downside of this approach to instrumentation is that the approach only works for the specific class of web service applications. Other types of systems do not have the same degree of standardisation as web services which would allow a similar approach in these domains.

There are several approaches which instrument systems at the application level. A simple way to do this is to add annotations to the source code as in [Chodrow 91] where labels are added to the source code in comment statements which indicate when events should be emitted. These comments are transformed into instrumentation code

in the implementation language as a pre-compilation step.

This approach, where source code is annotated, is also used in [Chen 03, Sankar 93] where the code is annotated with actual formal requirements. A pre-compilation step is also used to generate instrumentation code to emit events which will allow the requirements to be checked.

Approaches which instrument a system by annotating the source code are similar to assertions but events are emitted to a monitor instead of being checked as part of the execution of the monitored system. This is important for more complex requirements specifications as the processing overhead involved in checking these requirements can be relatively large. This approach separates instrumentation and code to an extent as the instrumentation is placed in comments and so is clearly identifiable as such. There are limitations on the type of events which can be identified easily in this way. For example, it is easy to place an event which should occur at the start of a method but hard to capture calls to that method which originate only from another object or to capture all calls which modify an object member variable. To provide greater flexibility, instrumentation is written using AspectJ in the monitoring framework described in this thesis.

In [Liao 92] the requirements are separated from the implementation code although these requirements are still written in a way that is dependent on the details of the implementation. The requirements specification is used to modify the source code in a pre-compilation step in a similar way to the approaches which use code annotation.

In Java PathExplorer[Havelund 01] the target system is instrumented using a tool which modifies Java byte code. This allows the instrumentation to be defined separately from the source code. It also allows for a greater range of events to be captured. For example, it is much easier to find all the modifications of a variable in byte code as byte code is simply a series of operations and only a small number of these operations can modify a variable.

Both Java PathExplorer and the work by Liao and Cohen implement instrumentation systems which work in a similar way to AspectJ, by modifying either source code during pre-compilation or byte code after compilation. This allows modification of the target system without altering the original source code. The development of AspectJ means that such approaches are no longer necessary as AspectJ provides a general solution to the need for non-invasive modification of the behaviour of a system.

An important way of approaching instrumentation is presented in the work on Java-MaC [Kim 01] in which instrumentation is generated from mapping between requirements specification and the implementation. This is done by specifying when the boolean variables used in the requirements specifications are true by referring to variables and methods in the implementation. The work in this thesis build on this approach. In particular, the idea of a mapping language, which is used to generate instrumentation code comes from here. This idea is adapted to allow mapping of KAOS requirements to implementation code and to automatically generate AspectJ code from this mapping.

## 2.4 Monitor Architecture

### 2.4.1 Monitor Implementation

In [Snodgrass 88] makes use of a historical database which is an extension of a relational database which also stores previous values of fields as well as the most recent

value. The database is accessed using a specialised query language which can retrieve this historical information. This is an effective way of implementing a monitoring system as it provides exactly the capabilities needed to monitor requirements which include temporal specifications. Unfortunately, historical databases and query languages are not readily available.

In the absence of historical databases in wide use, plain relational databases have been used[Robinson 03] and this is also the approach taken in this thesis. Obviously normal relational databases can still be used to store historical information although not as simply as a historical database.

### 2.4.2 Distributed Monitoring

There is some published work which deals specifically with the problems associated with distributed monitoring. The problem of message ordering when monitoring distributed systems is the focus of the GEM system[Mansouri-Samani 97]. GEM assumes a globally synchronised clock which allows events to be time stamped and put in the correct order once all relevant events have been received. The difficulty is determining when it is safe to start processing events as it is not known if there is still a delayed event with a time stamp earlier than the latest event which has been received.

One approach to this problem is to delay every message by the maximum possible communication delay before processing it. This ensures that messages will be processed in order if the delay introduced really is the maximum possible. This approach is rejected on the grounds that it may introduce unnecessary delays, it is inefficient because it orders messages that may not need to be ordered with respect to each other and it delays messages which may already be ordered, for example if they were generated locally.

The approach adopted in GEM is to delay only those messages necessary. This still requires a maximum tolerated delay to be specified but this delay is applied to specific messages rather than to every message. The user of the framework may specify delays for individual messages where necessary.

The event correlation approach in GEM puts some extra work on the user of the framework by requiring individual delays to be specified for each message used by each rule. It is also necessary to determine maximum tolerated delays. It is clear that there is a trade-off to be made between the responsiveness of the monitoring system and the failure rate caused by delayed messages.

The approach used in this thesis is to explicitly inform the monitor server when each distributed component has no more messages to send. This approach focuses on accuracy at the expense of responsiveness but is well suited to the needs of monitoring temporal logic specifications.

The work described in [Sen 04] deals particularly with the problems associated with distributed monitoring. An interesting feature is that the monitoring algorithm is itself distributed. The work uses past-time temporal logic to specify requirements. This is extended with operators that refer to the last *known* state of a remote process. The monitors communicate with each other by piggy-backing monitoring information on the existing message which are passed by the system as part of its normal execution. This is an interesting approach, particularly when communication overhead is a concern, but it is not used in this thesis where distributed monitoring is not a core concern.

## 2.5 Formal Specification and Monitoring of Soft Goals

In [Robinson 03] the monitoring framework includes aggregate monitors which look for repeated failure of hard goals. This is an approach to monitoring soft goals. A formally specified hard goal can be made soft by only requiring that it is satisfied in most cases. This approach is built on in this thesis by providing a more complex language for specifying soft goals, including the concept of aggregating results from many hard goal instances.

Soft goal specification is tackled in [Letier 04] which is an extension of the KAOS methodology to soft goals. The approach works by defining quality variables for goals using natural language definitions. Objective functions are then defined which specify quantities to be maximised or minimised. The objective functions are mathematically defined using the quality variables. Target values are also specified for objective functions.

The approach includes guidelines for identifying relevant quality variables and objective functions. Propagation rules allow the degree of satisfaction of a goal to be determined from its sub-goals.

This approach is similar to the approach adopted to specifying soft goal metrics described in this thesis although the quality variables are not defined formally. The work in this thesis builds on the approach in the work of Letier and van Lamsweerde, which is not primarily intended to allow monitoring, by formally specifying goal instance metrics. These metrics, which are similar to quality variables, are formally defined using the KAOS object model so that they can be monitored.

## 2.6 Display of Monitoring Results

There has been relatively little work done in providing visual feedback from monitoring. There is a particular lack of research in what sort of feedback is useful to the user of a monitoring framework.

The work described in [Robinson 03] provides some visual feedback. Goal hierarchies can be displayed which show which goals have suffered failure. UML sequence diagrams can also be used with additional information such as average response times and failure rates for the messages added to the diagram. The display for hard goal failures in this work is essentially similar.

## 2.7 Summary

This chapter has reviewed work on run-time monitoring and some relevant work in other related areas. The areas of debugging, logging and assertions all have similarities to run-time monitoring although the aims are different and so the solutions which are arrived at are not necessarily suitable for run-time monitoring.

Run-time requirements monitoring differs from debugging and other forms of execution monitoring because run-time requirements monitoring aims to provide an abstracted view of system execution. The aim of debugging is to provide an implementation level view of the execution. Requirements monitoring tries to abstract this low level view to provide a view at the same level of abstraction as the requirements model. The advantage of this abstraction is that it allows developers to see large scale issues which affect the monitored system. In particular, failures caused by changes in the environment in which the system operates are likely to be easier to identify at the requirements level as the environment is modelled explicitly at this level.

There have been a number of frameworks for run-time monitoring proposed. Some of these frameworks deal with monitoring requirements specified in temporal logic. Checking these goals once the necessary information has been gathered is relatively straightforward and the work in this thesis follows a similar approach.

Most run-time monitoring frameworks do not provide any support for the instrumentation of the monitored system. A few frameworks do provide some support although they use their own particular requirements specifications. The work in this thesis combines much more extensive support for instrumentation, using AspectJ or a mapping language, with an existing requirements engineering language.

Another common approach is to include instrumentation code as part of a middleware or other implementation of a domain specific architecture such as web services architectures. This allows any system which uses the middleware or architecture to be monitored without the need to write new instrumentation code for each new system. Of course, the instrumentation is specific to that one domain. In contrast, the monitoring framework described in this thesis is intended to allow instrumentation of any system.

There has been some work on specifying soft goals more rigorously, although a completely formal definition is not really compatible with the definition of a soft goal. Only a small amount of work has been done on actually monitoring soft goals.

Little research has been done on communicating monitoring results back to the user or using monitoring results to directly modify a system. Some work has been done using goal graphs and sequence diagrams to display monitoring results.

# Chapter 3

# Background

## 3.1 Goal-Oriented Requirements Engineering

Goal-oriented requirements engineering forms an important foundation for the work in this thesis as it is used to specify the requirements which are monitored. This section gives a brief outline of goal-oriented requirements engineering and a description of the KAOS goal-oriented requirements approach which is used to formally specify hard goals so that they can be monitored. KAOS also forms the basis for the formal specification of soft goal metrics which are discussed in chapter 5.

An overview of the area of goal-oriented requirements engineering is given in [van Lamsweerde 01], in which van Lamsweerde defines a goal as follows:

> A *goal* is an objective the system under consideration should achieve. Goal formulations thus refer to intended properties to be ensured; they are optative statements as opposed to indicative ones ...

Goals are normally identified in the early stages of the requirements engineering process. They can be derived directly from stakeholders, from an existing system or from requirements documentation provided by stakeholders. Once an initial set of goals has been identified it is possible to elaborate this model by identifying additional goals related to the initial goals.

Goals can be categorised into functional and non-functional goals as well as hard and soft goals. Soft goals are goals which do not have formal criteria for satisfaction while hard goals do. So, for example, a goal stating that users want to be able to download a file which appears in a search result is a functional, hard goal. A goal stating that users want download times for those files of no more than one minute per megabyte is a non-functional, hard goal. If the goal states only that users want download times which are reasonably fast then that is a non-functional soft goal.

Satisfaction of soft goals often involves trade-off with other soft goals. The stakeholders in the system want each soft goal to be optimised to as great a degree as possible but improving one soft goal can have a negative impact on another. For example, usability and security are two areas which often conflict with each other. Adding features which support a security soft goal such as adding password protection and enforcing password changes make the system harder to use, as users must remember their passwords and change them regularly, but should improve security.

### 3.1.1 The KAOS Approach

The KAOS approach[Dardenne 93, Letier 01] is a goal-oriented requirements approach. It includes a visual language for specifying relationships between goals, objects

and agents, a formal specification layer for goal specification and a methodology for developing requirements specifications using these tools. This section summarises only those parts of the KAOS approach which are relevant to the work in this thesis.

KAOS uses five different models to represent different views of a system. The models which are most relevant to monitoring are the goal and object models. These two models form the core of the requirements specifications which are monitored by the monitoring framework described in this thesis. The other models are the agent responsibility model, which defines which agents are responsible for satisfying which goals, the operation model, which defines operations which need to be carried out to satisfy goals, and the agent interface model which defines what variables an agent has access to and what variables it controls.

KAOS models involve three levels of specification. The KAOS meta-model defines the concepts which are used in KAOS such as entities, agents, relationships and the type of associations which are permitted between the concepts. For example, an agent may be associated with a goal through a responsibility link. This level specifies the KAOS language and is used in the specification of the KAOS methodology.

The domain model is created by instantiating the KAOS meta-model for a particular domain. For example, in the domain of file-sharing applications, concepts such as files and file-sharing clients will appear in the domain model. This is the level which is normally used during requirements engineering.

The instance model contains instances of concepts in the domain model. These instances represent individual objects. A specific file is an example of an instance concept. Instance models are not often used during requirements engineering but they are important for run-time requirements monitoring. At run time, the monitor is able to construct an instance model of the running system. This model is modified as the actual system changes to reflect those changes.

## The Goal Model

The goal model identifies the goals for a system and shows the relationship between them. In the KAOS methodology this is the starting point for the development of the requirements specification. Once initial goals have been identified from stakeholders, the goal model is elaborated to identify additional goals.

KAOS goals are related to each other using AND/OR links. An AND refinement means that if all the sub-goals of the refinement are satisfied then the parent goal will also be satisfied. An OR refinement means that if any of the sub-goals in the refinement are satisfied then the parent is satisfied. OR refinements represent alternative strategies for implementing a parent goal. Normally only one alternative OR refinement is selected to be implemented. The decision of which refinement should be implemented is often determined by which alternative best satisfies the soft goals which affect the system.

The goals and the AND/OR refinements that relate them form goal graphs. These can be represented graphically as shown in figure 3.1. Here goal A is AND refined into two sub goals, goal B and goal C, which is shown by the horizontal line linking the branches of the tree. Goal C has two alternate refinements, goal D and goal E. This is an OR refinement which is indicated by the lack of a horizontal line. Goal E is further refined but the sub-goals are not shown in the diagram.

The refinement of the goal model is guided by asking 'How?' and 'Why?' questions about the initial goals. 'How?' questions ask how an existing goal should be

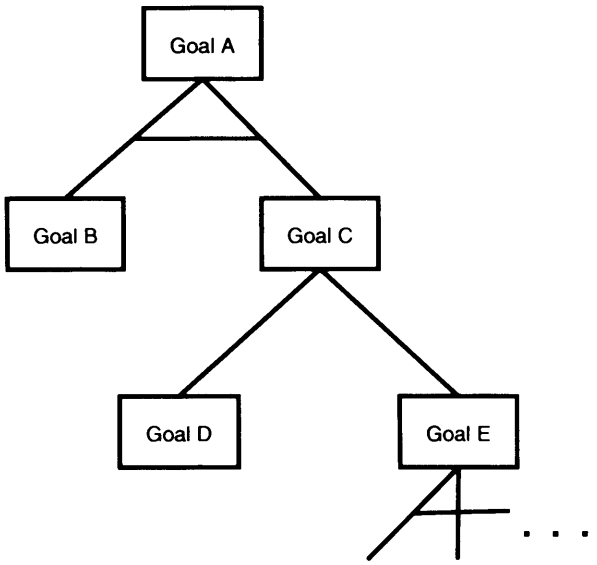



Figure 3.1: An example of a goal graph, showing AND/OR refinements.

satisfied. This results in one or more alternative AND refinements which satisfy the existing goal. 'Why?' questions ask why the stakeholders want to satisfy a goal. These questions result in the identification of parents of existing goals and the identification of siblings of the initial goal which are necessary to satisfy the new parent goal. It may also result in the identification of alternative strategies for satisfying the new parent goal which do not involve the initial goal.

The elaboration of the goal graph continues until each leaf goal can be assigned to a single agent which is responsible for satisfying it. Agents can be humans or devices which are capable of carrying out operations which satisfy goals. They may be part of the system or part of the environment in which the system operates. Goals which are the responsibility of an agent which is part of the system are requirements while those that are the responsibility of agents which are part of the environment are assumptions.

In the visual modelling part of KAOS, agents are indicated by hexagons. It is often useful to add agents to goal graphs and to display the responsibility links between agents and goals. Figure 3.2 shows an example of this. Goal B is the responsibility of Agent 1 while goal D is the responsibility of Agent 2.

## Goal Specification

KAOS has two levels of specification for goals; semi-formal and formal. At the semi-formal level of specification, goals are specified informally using natural language while at the formal level, temporal logic is used to specify goals.

In the semi-formal specification of goals, each goal is assigned a pattern depending on what type of behaviour should be exhibited to satisfy the goal. There are four goal patterns in KAOS: achieve, cease, maintain and avoid. Achieve goals require that some property should hold at some time in the future. Cease goals require that a property should no longer hold at some point in the future. Maintain goals require that a property should always hold. Finally, avoid goals require that a property should never hold.

Formal specification of goals uses real-time linear temporal logic.The following temporal operators are used in KAOS to formally specify goals:

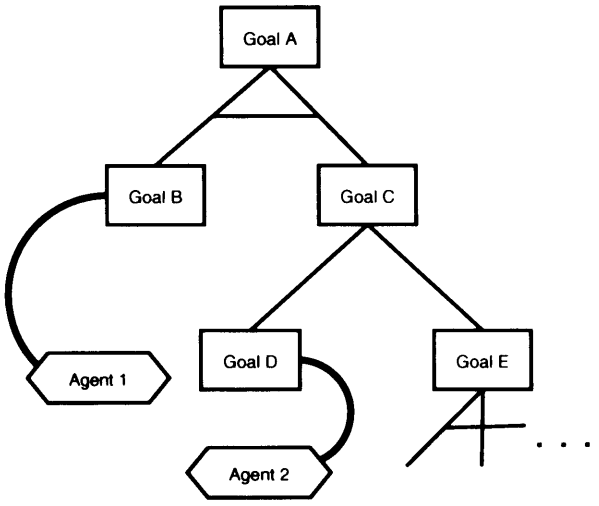



Figure 3.2: An example of a goal graph, showing agent responsibility links.

| | |
|---|---|
| $\Diamond$ some time in the future | $\blacklozenge$ some time in the past |
| $\Box$ always in the future | $\blacksquare$ always in the past |
| $W$ always in the future unless | $B$ always in the past back to |
| $U$ always in the future until | $S$ always in the past since |
| $\circ$ in the next state | $\bullet$ in the previous state |

The temporal logic specification of a goal is related to the goal pattern. Typical goal patterns for achieve and maintain goals from [Letier 01] are:

achieve: $P \Rightarrow \Diamond Q$ $P \Rightarrow \Diamond_{\leq t} Q$ $P \Rightarrow \bigcirc Q$

maintain: $P \Rightarrow Q$ $P \Rightarrow \Box Q$ $P \Rightarrow QWR$

The predicates in KAOS goal specifications are based on the KAOS object model. Predicates are typically either the existence of a relationship, a binary comparison between attributes or the occurrence of an event.

## Object Model

The specification of individual goals leads to the creation of the KAOS object model. This models the agents, entities and relationships between them which are used in the specification of the goals.

If goals are specified formally, then the object model can be derived from the goal specification. Any object which is used in a goal specification must be present in the object model. In practice, the formal specification of goals and the elaboration of the object model is likely to happen simultaneously.

KAOS has four types of objects: agents, entities, relationships and events. All objects have a name and can, in principle, have attributes. Agents are objects which can perform operations and goals are satisfied by agents which perform operations. Agents are related to goals through responsibility links which show which goals need to be satisfied by a given agent. Entities are similar to agents except that they are passive objects which cannot perform operations.

Relationships are also considered objects. A relationship links two or more other objects, called roles. The linked objects are almost always agents and entities although strictly they can also be other relationships or events. In formal goal specifications, the predicate:

$R(r_1, r_2, ... r_n)$

is true when an instance of relationship 'R' exists, which links roles $r_1$ to $r_n$. Relationships are frequently used in the definition of KAOS goals. For example, the following specification:

$$\forall \text{ a:A, b:B}$$
$$P(a, b) \Rightarrow \Diamond\, Q(a, b)$$

says that whenever any instance of relationship P exists, a corresponding instance of relationship Q, with the same objects as roles, should exist at some time in the future.

The final type of object are events which are objects which exist only instantaneously. These are not considered further in this thesis as event monitoring can be implemented by a relationship which is created and then immediately destroyed, resulting in a relationship which only exists instantaneously.

### 3.1.2 Soft Goals

The monitoring framework described in this thesis also takes attempts to monitor soft goals to determine if they are satisfied by a monitored system. The work on which the notion of soft goals is based is described below.

#### The $i^*$ Framework

The $i^*$ framework[Yu 97] is a framework for early-phase requirements engineering. The organisation in which a software system is to operate is modelled using dependency relationships between actors and tasks, goals and resources. More complex models describe the interests of actors and how their needs are addressed by the proposed system.

The work in this thesis builds on the notion of soft goals in $i^*$ and other work. In $i^*$, soft goals are related to other concepts in the model using contribution links which show that a concept contributes negatively or positively to a soft goal. Soft goals may also be used in task decompositions where they contribute to the completion of a task.

#### NFR Framework

The NFR framework[Mylopoulos 92, Mylopoulos 99] is a framework for modelling non-functional requirements which makes use of soft goals. Rather than satisfaction, the concept of satisficeability is used when analysing soft goals. These concepts are defined as follows:

> Soft goals are goals that do not have a clear-cut criterion for their satisfaction. We will say that soft goals are *satisficed* when there is sufficient positive and little negative evidence for this claim, and that they are *unsatisficeable* when there is sufficient negative evidence and little positive support for their satisficeability.

The simple examples of the NFR framework use four types of relationship between soft goals to analyse their satisficeability. Negative and positive influences, where one goal contributes negatively or positively to the satisficing of another goal, are used as described in the definition of satisficeability above. The other two types of relationship are AND and OR relationships. In the case of the AND relationship, the parent goal is satisficed when all the child goals are satisficed and no goal exerts a negative influence

on the parent goal. For the OR relationship, the parent goal is satisficed when one of the child goals is satisficed and there is no negative influence on the parent goals.

The methodology proceeds by first identifying very abstract soft goals for the system being constructed. Two examples are usability and flexibility. These soft goals are then refined using AND/OR refinement in a process which is similar to that followed for hard goals in KAOS. Once this process has been carried out for all the abstract soft goals, lateral links are made between the goals in these goal trees. This shows how non-functional requirements constructively and destructively interfere with one another.

The next step is to relate the hard goals for the system to the soft goals using positive and negative relationships. These relationships are then used to select between alternate hard goal refinements.

While this approach is effective for the analysis of soft goals, it is not designed to allow monitoring of these goals, which requires some form of formal specification. The concepts of soft goals which are developed in the NFR framework and other work are built on in this thesis to allow the development of formal specifications for metrics which can be monitored.

### Tropos

Tropos[Castro 02] is a software development methodology, based on soft goals and related concepts described in the NFR and $i^*$ frameworks. This methodology extends these concepts from early stage requirements engineering through to late stage requirements engineering, architectural design and detailed design. It is further suggested that the detailed design can be naturally implemented in using an agent-oriented programming platform.

Such a development process would certainly make monitoring easier to implement as it solves a number of problems which arise in requirements monitoring. In particular, the problem of tracing requirements to implementation is made easier as there is good traceability from requirements to detailed design and a clear link from detailed design to implementation using an agent-oriented platform. However, the methodology is still in development and is considerably different to most software development methodologies which are in current use so this work is not used in this thesis.

## 3.2 Aspect-Oriented Programming

Aspect-oriented programming is a technique designed to aid the separation of concerns. In particular, it aims to separate concerns which are 'cross-cutting' in object-oriented programs. A concern is cross-cutting if the code to implement that concern is split up among different modules in the system. A related problem, which is also solved by modularising cross-cutting concerns, is 'tangling', in which code which addresses several different concerns is found in a single module.

The problems of cross-cutting and code tangling are a particular problem for concerns which are related to non-functional requirements. For example, concerns such as synchronisation and persistence may be difficult to encapsulate in object-oriented programs and may be best encapsulated using aspects.

Aspect-oriented programming adds additional abstractions to those normally used in object-oriented programming. The aim of these abstractions is to allow cross-cutting concerns to be encapsulated. They are also intended to allow the modules which are cross-cut to remain oblivious to the cross-cutting module. This principle is essential to removing tangling when several cross-cutting concerns are related to a single module.

In the monitoring framework, aspect-oriented programming is used to allow instrumentation code to be modularised and to ensure that the code in the monitored system is oblivious to the instrumentation code. There are several proposed approaches to aspect-oriented programming which are summarised below. The approach adopted in this thesis is AspectJ, which is the approach which currently has most development behind it.

### 3.2.1 AspectJ

AspectJ[Kiczales 01] is an extension to the Java language which adds a new language construct called an 'aspect'. An aspect is a modular unit much like a class or interface. The purpose of an aspect is to modularise a concern which cuts across the structure of the base code.

Aspects contain pointcuts and advice. These are defined in[Kiczales 01] as follows:

> Join points are well-defined points in the execution of the program; pointcuts are a means of referring to collections of join points and certain values at those join points; advice are method-like constructs used to define additional behaviour at join points.

AspectJ pointcuts are constructed by combining primitive pointcuts which select different types of join points. The principle primitive pointcuts which are used in this thesis, and generally the most commonly used primitive pointcuts, are:

**call / execution** These primitive pointcuts match the execution of methods which match a pattern. The pattern can use wild card characters to, for example, match all the methods of a particular class. Call and execution pointcuts have slightly different meanings but the distinction is not particularly important to the work presented in this thesis.

**get / set** Primitive pointcuts which match reading and modifying fields respectively. The fields which are matched are selected by a pattern, in a similar way to the call and execution pointcuts.

**cflow** This primitive pointcut takes another pointcut as a parameter. It matches all join points which are within the control flow of join points matched by the parameter pointcut.

**this** Pointcut which matches at any join point where the executing object is an instance of a given type. This is an example of a dynamic pointcut as it is evaluated at run time. The argument to this pointcut can be a pointcut parameter of a specified type, in which case the value of the parameter is set to the currently executing object whenever the pointcut matches.

**target** This pointcut is similar to the 'this' pointcut except that it matches when the target object is an instance of a specific type. For 'call' pointcuts, the target object is the object being called and for 'set' pointcuts it is the object being modified.

**args** The 'args' pointcut matches any join point in which the arguments at the join point match the pattern supplied. In the case of a method call, the arguments are the method arguments. At a 'set' pointcut, the argument is the value which the field is being set to. The pattern may include pointcut parameters.

Primitive pointcuts are combined into more complex pointcuts using AND, OR and NOT operators. Pointcuts may be anonymous, in which case they must be associated directly to advice which executes at them, or named, in which case they can be associated with advice by name or used in the definition of other pointcuts. Whenever a pointcut matches a join point, the advice associated with that pointcut is executed. The code in the body of the advice has access to the parameters of the pointcut.

As well as this dynamic behaviour, AspectJ also allows static cross-cutting. This is done using *inter-type declarations*. An aspect may declare fields and methods which belong to another class, thereby modifying the interface of the class.

AspectJ is the most developed of the aspect technologies and is still under active development as an eclipse.org project. This means that new capabilities are constantly being added to the project. Recent additions include byte code weaving, which allows class files rather than source files to be used when compiling aspects; incremental compilation which means that only files which have changed need to be recompiled and IDE support in eclipse. The active development also means that AspectJ has kept up with changes in the Java language, such as the in the Java 1.5 release.

### 3.2.2 Hyper/J

Hyper/J[Tarr 99] concerns are programmed using normal Java classes. A separate specification in a custom language then describes how the different classes should be merged with each other to create a composite system which implements all the concerns.

In Hyper/J the implementation of a concern and the mapping with respect to other concerns are separated, unlike AspectJ in which they are combined within an aspect.

### 3.2.3 Dynamic Aspect Weaving

Both AspectJ and Hyper/J perform aspect weaving at compile time. There are also proposals for dynamic aspect weaving, which would allow aspects to be added or removed from a system while the system is running. In [Popovici 03], aspects are woven into a running system using a just-in-time compiler. In [Baker 02], language constructs are used to allow the system to add aspects to itself at run time using hooks which are woven into the program. In [Pawlak 01], hooks are also added to the program to allow aspects to be added and removed dynamically. Similar functionality may be incorporated into AspectJ at some time in the future.

### 3.2.4 Domain Specific Aspect Languages

The technologies described previously all handle aspects in a general purpose way. Another approach is to use domain specific aspect languages. An example of this is found in the precursor to the AspectJ project[Kiczales 97]. For example, synchronisation is described in a language which defines sets of methods to be mutually exclusive. In another example, a numerical algorithm is defined in the base language while the type of data structures to use, such as sparse matrices, are defined in a separate aspect.

The advantage of this approach is that aspects become very easy to program for a programmer with knowledge of the domain the aspect language is designed to address. The disadvantage is that there is a large up front cost in witting an aspect weaver for a new domain specific language. The approach adopted in this thesis, where a mapping language is used to generate AspectJ code is based to some extent on the idea of domain specific aspect languages. By building this language on top of AspectJ, work is saved in writing a specialised weaver to implement the language.

## 3.3 Peer-to-Peer File Sharing Example

This thesis uses a running example in which goals were monitored for a peer-to-peer file sharing program. The program used was Limewire which is an open source peer-to-peer client written in Java, which uses the Gnutella protocol. This section provides background information on the Gnutella protocol before introducing the goal-oriented requirements specification against which the program was monitored.

The Limewire example was selected to assist in development of the monitoring framework and it is used to provide examples in this thesis. Limewire was chosen because it is open source, written in Java and is reasonably complex. It is not intended to be an evaluation of the monitoring framework but to serve to illustrate the monitoring framework.

There are some issues with monitoring Gnutella networks which are not explored in this example. There are many different Gnutella client implementations available and a Gnutella network is typically composed of many different types of client communicating with each other. As only Limewire peers are instrumented, it is not possible to monitor the whole network.

Another problem is the scalability of the approach. As the monitoring framework uses a central server to receive results, performance of the monitoring system is likely to degrade as the number of peers increases. As a peer-to-peer network can be made up of thousands of nodes, scalability problems are likely.

While developing the monitoring framework, testing was done using very small Gnutella networks which were artificially constructed. Many goals can be monitored by considering only a single peer. Goals which involve multiple peers were monitored by connecting two or three peers which were all monitored. Deploying monitoring in a real network would require some way of determining which goals can be monitored using the available information. Deploying large scale monitoring would require a solution to the scalability problem, most likely using distributed monitoring.

### 3.3.1 The Gnutella Protocol

Peer-to-peer file sharing programs allow users to search for files stored by other peers in the network and download those files. They also allow the user to make their own files available for download. The first successful example of a peer-to-peer file sharing network was based on the Gnutella protocol.

When using the Gnutella protocol, peers connect to several neighbours. The exact method by which they find and connect to these neighbours is not covered here. Once a peer is connected to a suitable number of neighbours it is able to take advantage of the network.

A Gnutella search starts when a user enters a search query. This search is communicated to other peers as shown in figure 3.3. Each neighbour forwards the query to all its neighbours and so on until the query has been forwarded on a set number of times. The query has a field, called the time-to-live (TTL), which is decremented each time the message is forwarded. If a peer receives a query with a TTL of zero it is not forwarded to any neighbours. Normally the TTL is set to an initial value around six although the example in the diagram shows the progression of a query with an initial TTL of two.

In addition to forwarding queries, peers should also reply to queries which match files which they are sharing. The path taken by a query reply is shown in figure 3.4. The query reply travels along the reverse path of the query, back to the original sender.

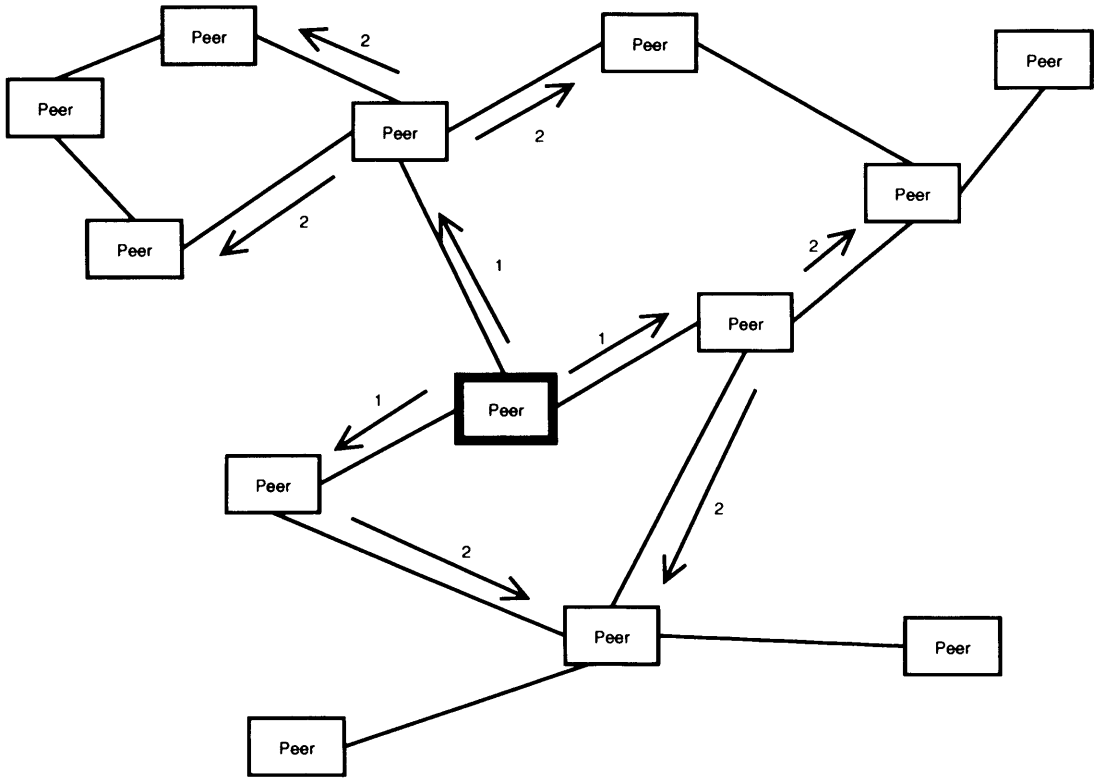



Figure 3.3: The propagation of a Gnutella query, with an initial TTL of two, through a Gnutella network.

To do this, each peer must keep track of the GUID (Globally Unique Identifier) of each query which it has recently forwarded, along with the peer which sent it. Query replies have the same GUID as the query they answer. Each peer can then forward a query reply that it receives to the peer from which it received the matching query.

A query may match several files on a particular peer so each query reply may contain several results, each one of which details a file which may be downloaded. The user can select one or more results to download. The client will then connect directly to the peer which generated the query reply (using the IP address included in the query reply), as shown in figure 3.5.

There are two examples of ways in which the Limewire implementation of the Gnutella protocol has evolved which are interesting in the context of monitoring. These examples illustrate what types of evolution can occur which will have impact at the requirements level and which monitoring of soft goals can potentially help with.

The first change which has occurred in the Limewire client is the introduction of super nodes. The purpose of this change was to reduce the bandwidth used in passing messages, particularly search queries, for clients which are connected to the network by slow connections. In this scheme, most clients connect to a single super node. Each super node is connected to a large number of other clients and a small number of other super nodes. This shields normal clients from the need to forward large numbers of queries and route query replies. The network still remains a peer-to-peer network as normal clients are appointed as super nodes if an existing super-node drops from the network and they have a high speed and reliable connection.

The second change which occurred was the introduction of segmented downloads.

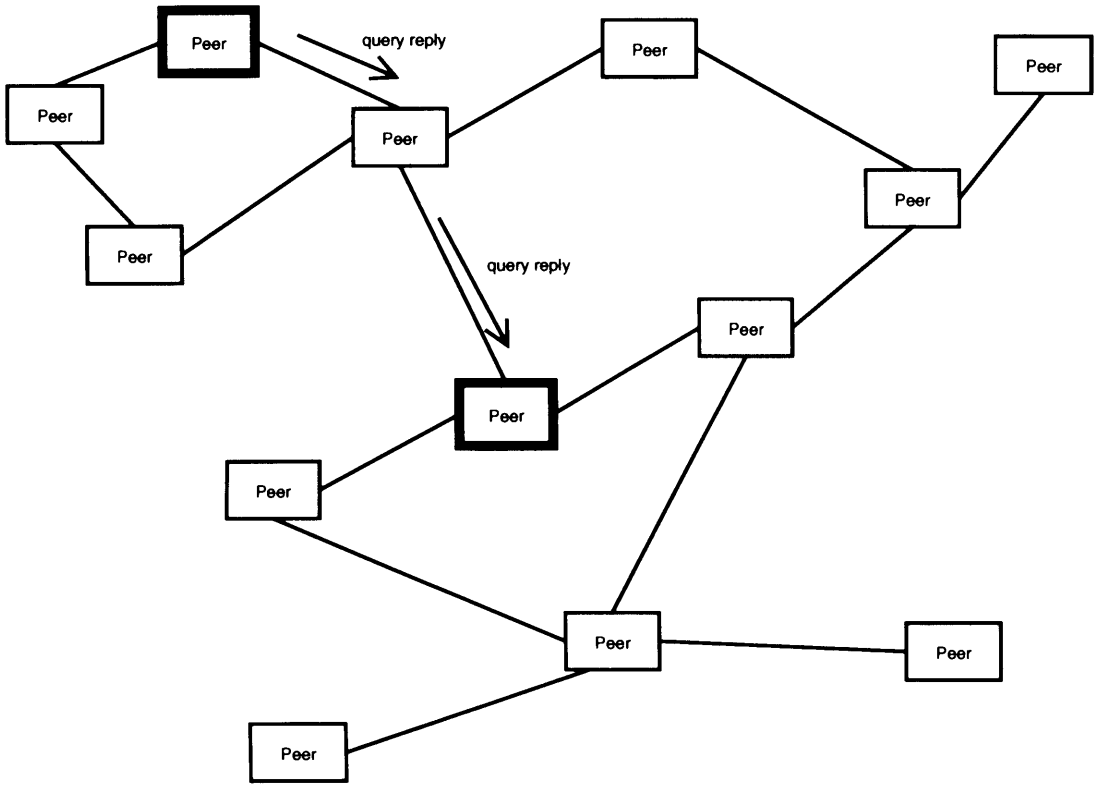



Figure 3.4: The path taken by a Gnutella query reply through a Gnutella network.

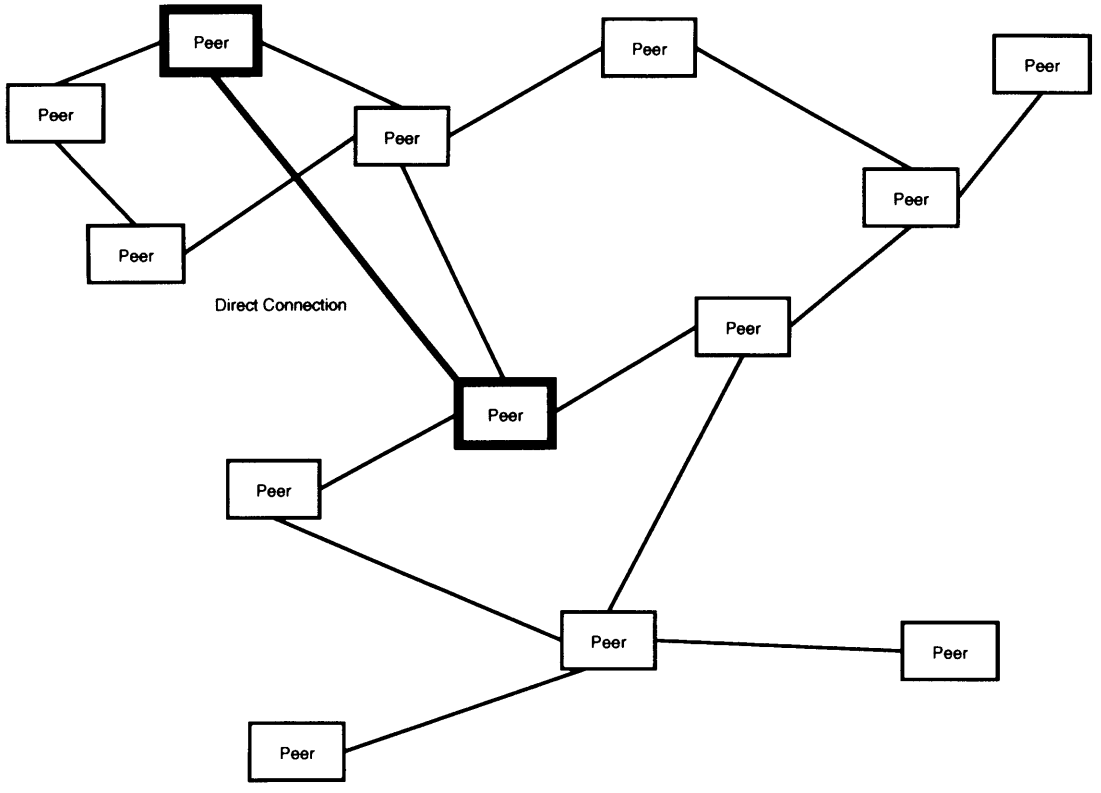



Figure 3.5: Downloading a file in a Gnutella network. Files are downloaded by establishing a direct connection between a peer which sent a query and a peer which replied to that query.

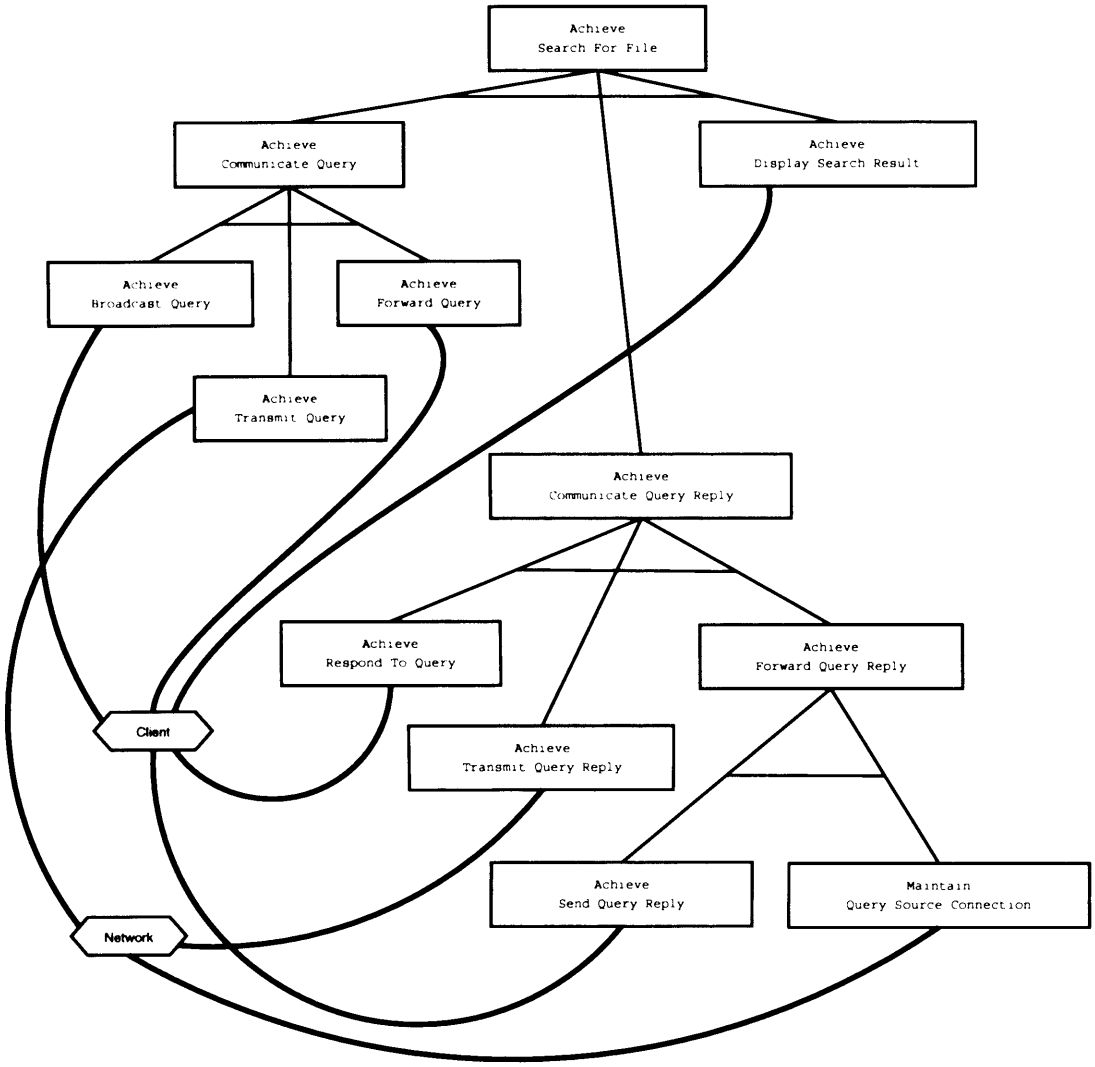



Figure 3.6: Goal refinement for the goal 'Achieve[Search For File]'.

The purpose of this change was to improve download performance by allowing parts of a file to be downloaded from several clients simultaneously. When several query results for the same file are received, a client can request different segments of the file from different peers. As each peer is likely to have limited upload bandwidth, higher download speeds can be obtained by downloading from multiple peers.

### 3.3.2 Goal-Oriented Requirements Specification

The main functions of a peer-to-peer file sharing program are to search for files and to download files. Satisfying these goals requires the cooperation of many peers. These two functions can be represented as goals and these goals can be refined, thereby generating the goals trees in figure 3.6 and figure 3.7.

The goal tree in figure 3.6 shows the goal 'Achieve[Search For File]' and its sub-goals. This goal expresses that the desire that the system should allow users to find files which match a search criteria and display those results. This is achieved through three sub goals. The first, 'Achieve[Communicate Query]' requires that queries should be propagated from the originating peer to other peers within a specified number of hops. The goal 'Achieve[Communicate Query Reply ]' requires that any peer which

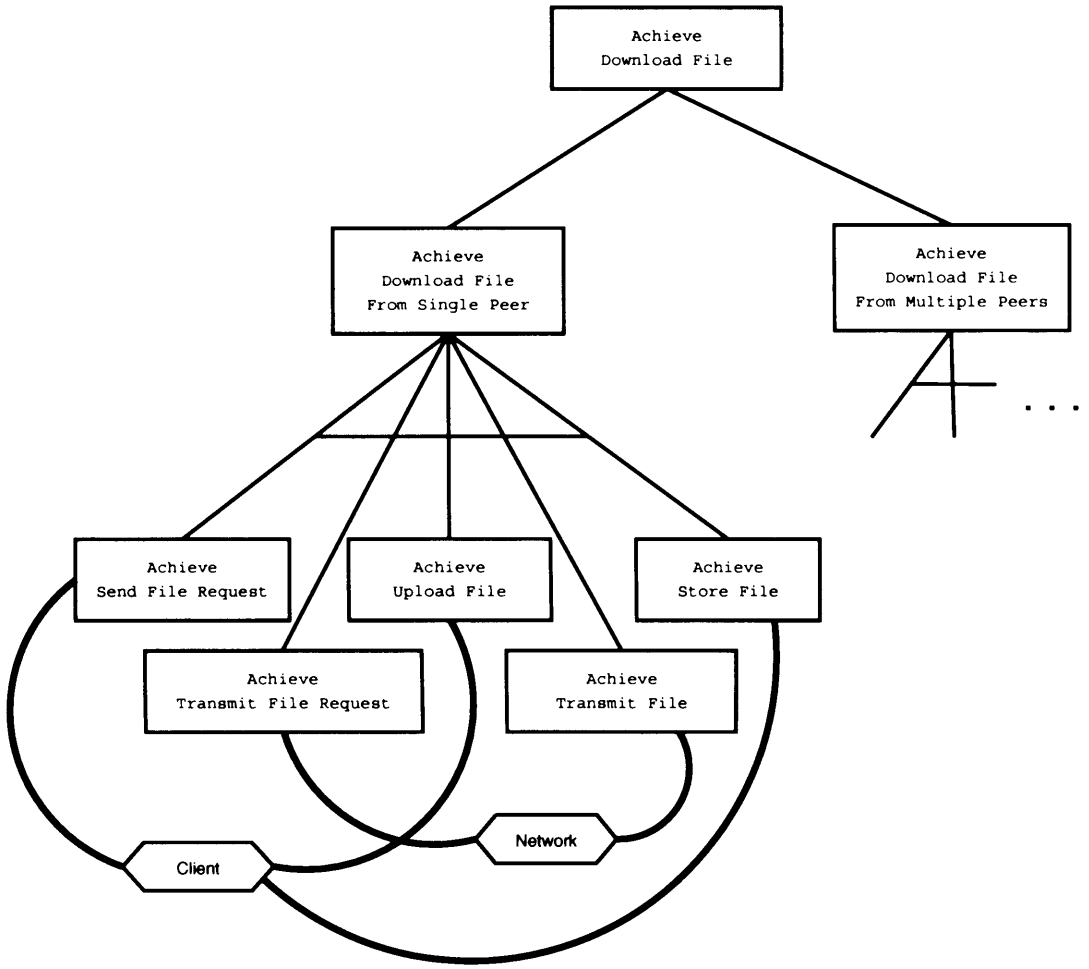



Figure 3.7: Goal refinement for the goal 'Achieve[Download File]'.

received a query should reply with any matches it has and that these matches should be returned to the originator of the query. Finally, the goal 'Achieve[Display Search Result]' requires that the peer which originated a query should display the results of the query to the user. The formal definitions of these goals are found in appendix A, although specific examples will be introduced as they become necessary in the text.

The goal tree in figure 3.7 shows the goal refinement for the goal 'Achieve[Download File'], which expresses the desire of the user that a file should be downloaded when it is requested.

There are two alternative refinements for this goal. As explained previously, files can either be downloaded from a single client or segments can be downloaded from multiple clients. Only the refinement for downloading from a single client is shown here as the refinement for multiple clients is similar.

The goal 'Achieve[Download From Single Peer]' is satisfied by five sub-goals, three of which are requirements which are the responsibility of peer-to-peer clients and two of which are assumptions which are the responsibility of the network which the clients use to communicate. The requirement 'Achieve[Send File Request]' requires that when a file is requested by the user, a request is sent to the client referred to by the file descriptor of the requested file. The assumption 'Achieve[Transmit File Request]' requires that a file request sent by one client will be received by its recip-

ient. The requirement 'Achieve[Upload File]' requires that when a file request is received, the corresponding file will be sent to the source of the request. The assumption 'Achieve[Transmit File]' requires that if a file is sent then it will be received. Finally, the requirement 'Achieve[Store File]' requires that when a file has been received that it is saved by the client which received it.

The goal trees also show the agent responsibility links which identify which agents are responsible for satisfying which leaf goals. The agents used here are 'Client' and 'Network'. It is not altogether clear where the boundary between the system and the environment lies with respect to the 'Client' and 'Network' goals. The view taken here is that all the clients make up the system. The network is part of the environment which the system has to operate in. This is not the only possible view. It is also possible to consider the network to also be part of the system, particularly if it were a closed network, but as the network in question is the public Internet, it seems most appropriate to treat it as part of the environment. Another view is that a single client should be considered as the system and all other clients should be considered part of the environment but given the cooperative nature of the Gnutella protocol it seems more appropriate to consider all clients as part of the system.

Given this choice of which agents are considered to be in the system and in the environment, those leaf goals assigned to the 'Client' agent are considered to be requirements while those assigned to the 'Network' agent are assumptions. The network is responsible for various assumptions that Gnutella messages will be delivered to their recipients.

# Chapter 4

# Monitoring Temporal Logic Goals

This chapter describes how the monitoring framework implements run-time monitoring for KAOS goals which are formally expressed using temporal logic. There are three main tasks which are carried out by the monitoring framework. The first is to instrument the target system. The second is to represent the state of the monitored system using a requirements level model of the system. The third is to use that model to check whether the requirements specification is satisfied.

The target system (the system to be monitored) is instrumented using AspectJ. This is used to both collect information from the system and to translate that information into requirements level events. Instrumentation can be specified directly in AspectJ or a separate mapping language can be used to describe the relationship between the implementation and requirements levels. Aspects are automatically generated from this mapping to instrument the target system. These two approaches are complementary; the mapping approach clearly shows the relationship between the implementation and requirements level for simpler cases but does not have the same flexibility as using aspects directly.

The monitoring framework uses the events, provided by the instrumentation in the target system, to build a model of the state of the target system. This model is an instantiation of the KAOS object model of the system. The monitoring framework includes two different implementations of this requirements instance model. The first implements the model in a relational database. The second implements the model using Java objects which are stored in memory. The developer is free to choose which of these two implementations to use depending on the particular situation.

The monitor framework automatically construct monitors from the goal specifications which check for failure of those goals. These goal checkers use the requirements instance model to determine whether individual goals have been satisfied by the target system. Goal checkers can attach listeners to the requirements level object model so that they are informed if changes occur. This then starts the execution of the listening goal checker which can make additional queries if it needs more information from the model. The goal checkers forward their results to a live display which shows any goal failures which have occurred in the target system and information about those failures.

Section 4.1 describes design issues which need to be considered when building a monitoring framework and the design decisions which were made for the monitoring framework described in this thesis. Section 4.2 describes the overall architecture of the monitoring system and the implementation of the requirements level object model and the goal checker. Section 4.3 describes how the target system is instrumented. Section 4.4 describes how the information obtained by monitoring is displayed to the

users of the framework.

## 4.1 Design Considerations

Constructing a monitoring framework requires that various design decisions are made. Chapter 2 contained reviews of a range of monitoring frameworks which have made various different choices when considering these issues. Most of these decisions have advantages and disadvantages and the best choice will depend on assumptions about what the target system and what the monitoring system is intended to achieve.

The target system is assumed to be written in Java. This is an assumption made more for convenience than anything else. Similar techniques could be applied to other languages although the instrumentation used in the monitoring framework relies on AspectJ which is specific to Java. There are also aspect extensions in development for other programming languages which would allow a similar approach to be adopted for these languages.

It is assumed that while performance may be important it is not critical. There are generally two types of system where performance is critical in different ways. Some systems have hard real-time constraints. The system cannot, under any circumstances, take longer to carry out an operation than the maximum time allowed. An example of a system with this type of constraint is an aircraft control system. Monitoring can be used in these types of applications as long as the real-time constraints can be applied to the instrumentation code. Unfortunately, the instrumentation code used in the monitoring framework described in this thesis is not subject to hard real-time constraints and so is not suitable for monitoring systems with such constraints.

Performance is also critical in systems where calculations take a long time to run and the execution time should be minimised. A common example of this is in scientific computing contexts. Instrumentation cannot be applied to the part of the system responsible for the calculations without having some impact on the performance of the system. It is, however, possible to monitor the parts of such a system which are not performing the actual scientific calculations to gain high level information about the execution of the system.

Discounting these types of systems still leaves many other useful applications for run-time monitoring. In general, systems written in Java will not fall into the categories described above in any case as the use of a virtual machine and garbage collection make the language unsuitable when performance is a critical issue. Most business systems are not time critical to the same extent as the types of systems mentioned previously. Interactive systems are normally limited by the speed at which the user interacts with them rather than the performance of the system. It is still useful to monitor how long tasks take to perform in such environments but the constraints are much less rigorous.

It is also assumed that timing of events will not be overly critical. Events related to a single goal are likely to have a separation of a second or more if they are generated by different distributed components, meaning that available systems for clock synchronisation should be adequate.

Another assumption that the instrumentation there may be several developers working on a system. It is possible that instrumentation code and system code will be written by different developers. This means that it is beneficial if these concerns can be separated as much as possible.

This section discusses the issues which are involved in building a monitoring

framework and what choices were made in the design of the monitoring framework described in this thesis, taking account of the above assumptions.

### 4.1.1 Message Translation

An important issue in run-time requirements monitoring, and a key feature which distinguishes requirements monitoring from other types of monitoring, is the need to translate events which are collected by instrumentation into events which relate to the requirements model in which the requirements are specified. The complexity of this translation depends on both the formalism of the requirements specification and the implementation language. The closer the two are related, the easier it is to translate from one to the other.

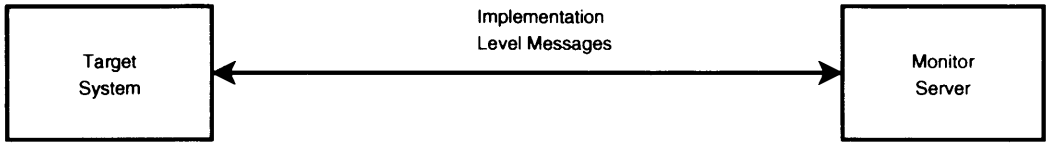
The KAOS language has an object model which is used in the temporal logic specification of goals. As the target system is assumed to be written in Java, which is an object-oriented language, translation of events between the two involves translating from one object model to the other. While the KAOS object model forms the basis of the translation for the framework, the translation problem is still not easy. KAOS requirements are very abstract and the object model reflects this fact. The actual object model which corresponds to the implementation of the target system will be far more complex. The translation is not trivial and needs to be explicitly defined by the developer.

There are three possible locations to handle translation in the system. The translation can take place in the monitor server (figure 4.1a), in a separate stage between the two (figure 4.1b) or in the instrumentation of the target system (figure 4.1c).

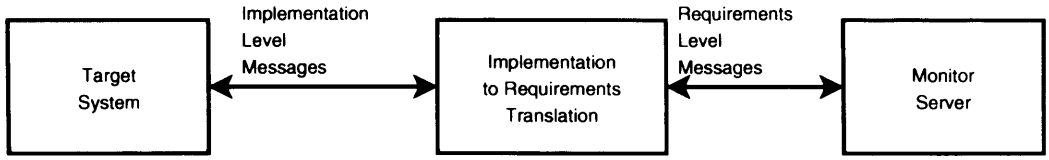
In the first of these options, the target system is instrumented to emits messages which contain implementation level events. These events are received by the monitor server which must use these events to evaluate the satisfaction of the requirements specification. To do this, the monitor server must have some knowledge of the relationship between implementation and requirements events. As a consequence, the monitor server is dependent on the implementation of the target system. This is undesirable as it means the monitor server has to change to reflect changes in the target system. This approach is most often used when the requirements specification itself is dependent on the implementation so that changes to the implementation already necessitate changes to the monitor.

In figure 4.1(b) an intermediate message translation stage is added to the architecture. This translates implementation level events to requirements level events. This is the approach taken in [Kim 01] for example. This approach means that the monitor server is only dependent on the requirements model, not on the implementation of those requirements.

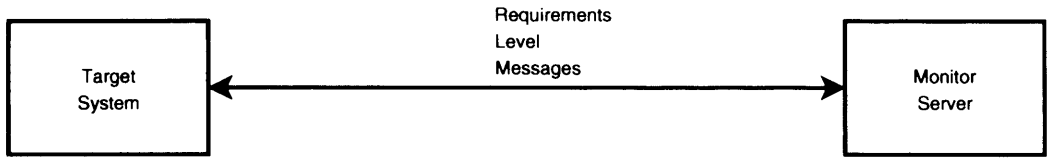
The approach taken by the monitoring framework described in this thesis is the one in figure 4.1(c). The translation of implementation level events to requirements level events is performed as part of the instrumentation of the target system. The messages emitted by the instrumentation are then requirements level events. This approach also means that the monitor server is independent of the implementation of the system. This approach is chosen over the second one because of the good fit with the AspectJ language which is used for instrumentation. In the monitoring framework, implementation level events are captured using AspectJ pointcuts. Examples of implementation level events are then calling a method, modifying a member variable and reading a member variable as well as more complex combinations of events which can be specified using




(a) When translation takes place in the monitor, implementation level messages are exchanged between the monitor and the target.




(b) When translation takes place in the target system, the exchanged messages are requirements level messages.




(c) With a separated message translations stage, messages are translated from the implementation level to the requirements level between the target and the monitor.

Figure 4.1: Approaches to message translation.

AspectJ.

Requirements level events are modifications to the KAOS object model. Examples of such events are creating a new instance of a relationship or entity and modifying the value of an attribute. The translation between these two types of events is achieved by advice which is attached to the pointcuts. This advice is called when an implementation level event occurs and creates the corresponding requirements level events.

### 4.1.2 Active and Passive Instrumentation

The purpose of instrumentation code is to allow properties of the system to be measured. There are two approaches which can be taken to this measurement; active and passive instrumentation. Active instrumentation produces messages when the value of a measured property changes. When the target system is instrumented using active instrumentation it is said to be traced. Passive instrumentation supplies the value of the measured property when requested, usually at regular intervals. A system instrumented with passive instrumentation is said to be sampled.

The main advantage of active instrumentation is that changes to a property will never be missed by the goal checker. With passive instrumentation, the goal checker can miss a property change if the property changes twice or more between samples. The monitor for a temporal logic specification cannot tolerate missed events if it is to accurately check the requirements specification. The monitoring framework must therefore either use active instrumentation or passive instrumentation which is sampled at a sufficiently high rate that events are never missed. It is difficult to determine exactly what sampling rate is necessary to ensure that no events are missed. This is likely to lead to a higher rate of sampling than is actually necessary as the target system will be sampled even when no change has occurred. Even then, it is difficult to be sure that

no event has been missed. The overhead of active instrumentation, in contrast, is the minimum possible for the rate at which events occur[Kaelbling 90].

A secondary advantage of active instrumentation is that the time between a property changing and the change being detected by the goal checker will normally be less than for passive instrumentation. Active instrumentation will communicate a change in a property immediately while the delay for passive instrumentation can be as large as the sampling period.

The disadvantage of active instrumentation is that it is more complicated to implement as it must be determined both when the property has changed and what the value of the property is. Passive instrumentation only has to determine the value of the property in response to a query from the monitor. Despite this greater complexity, the likely improved performance and guaranteed accuracy of active instrumentation mean that this is the approach which was taken for the monitoring framework.

### 4.1.3 Instrumentation Method

There are several ways to implement instrumentation. The most straightforward is to include instrumentation as part of the development of the system. This means that instrumentation code will be entangled with the rest of the system code. This is a particular problem because instrumentation code is likely to be found in many parts of the system.

Another approach is to use tools which modify byte code to instrument the system. This can be automated using some mechanism to specify what should be instrumented. This is easier than instrumenting at the source code level because byte code is simpler and therefore easier to process automatically. This also keeps instrumentation code separate from the code of the target system. The disadvantage of this approach is that it only provides a way to extract information from the system. The extracted information must still be translated to the requirements that are being monitored.

The approach used in this work is to use aspect-oriented programming techniques to implement instrumentation. AspectJ does the same basic job as other instrumentation approaches, in that it captures execution events in the instrumented system while keeping instrumentation code separate from the code of the monitored system. There are however a number of advantages in using AspectJ.

AspectJ provides a rich set of language constructs for defining pointcuts, which select execution events, and for constructing complex pointcuts from primitive pointcuts. For example, AspectJ allows a pointcut to be defined which selects all events which occur within the control flow of a particular method. As the language has evolved, new primitive pointcuts have been added, increasing the power of the language.

As AspectJ is an existing language, developers may already have experience in using it. If this is the case, it reduces the difficulty in implementing instrumentation code.

The structure of AspectJ code naturally fits the instrumentation code which is needed for the requirements monitoring framework where implementation level events are captured and then translated into requirements level events. Using AspectJ, pointcuts capture implementation level events while the advice attached to those pointcuts translated the events to requirements level events.

Finally, AspectJ instrumentation code is not dependent on the mechanism used to include that instrumentation in the system. For example, AspectJ is capable of both source code weaving, where aspects are included in the system at compile time, and

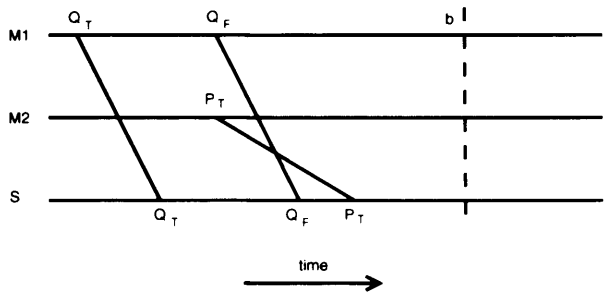



Figure 4.2: A failure is erroneously detected for P⇒ $\Diamond_{\leq b}$Q.

byte code weaving, where aspects are added to code which is already compiled.

### 4.1.4 Message Ordering

There are particular problems which need to be addressed when monitoring a distributed system. One problem is that there is a delay between a message occurring in the target system and the message being received by the monitor. These delays will generally be variable and unpredictable. The delay may not always be the same for a particular source, although it is assumed that all messages from a particular source will arrive in the order they were sent. This problem can however result in messages from different sources being received in a different order than they were sent.

Such ordering problems can result in both false positives,where failures are detected erroneously, and false negatives, where failures are not detected when they should be. A situation where a failure is detected erroneously is demonstrated in figure 4.2 which shows two distributed machines, M1 and M2, communicating with a monitor server, S, which is checking the goal P⇒ $\Diamond_{\leq b}$Q. It is assumed that the conditions P and Q are initially false. The machine M1 sends a message, $Q_T$ to the monitor server telling it that Q is true. The machine M2 then sends a message saying that P is true. M1 subsequently sends a message $Q_F$, telling S that Q is no longer true. The error occurs because the message from M2 is delayed, arriving after the second message from M1. The true order of events is thus $Q_T$ $P_T$ $Q_F$ but the events are received in the order $Q_T$ $Q_F$ $P_T$. This results in a false positive as failure is detected when the goal is actually satisfied.

The monitor will fail to detect a failure for the same type of goal in the situation shown in figure 4.3. In this case, the actual ordering of the messages is $Q_T$ $Q_F$ $P_T$. This means that the goal fails because the condition 'Q' never holds again after 'P' holds. The order the events are received is $Q_T$ $P_T$ $Q_F$ which appears to the monitor as though the goal has not failed because 'Q' holds at the time the event 'P' is received.

These types of problems can also occur with maintain goals of the form P⇒ $\Box_{\leq b}$Q. In figure 4.4 a failure is detected when none has occurred because the ordering of events received by the monitor is $P_T$ $Q_T$ which means that 'Q' does not hold until after 'P' does, violating the goal. The correct ordering, $Q_T$ $P_T$ means that 'Q' already holds before 'P' does and the goal does not fail.

In figure 4.5, the monitor will not detect a failure even though a failure has occurred. The order of events are $P_T$ $Q_T$ which means that 'Q' did not hold when 'P' became true. The order of events received by the monitor was $Q_T$ $P_T$ which does not register as a failure.

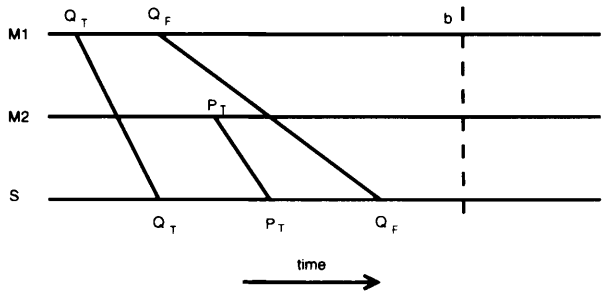



Figure 4.3: A failure is not detected for P⇒ $\Diamond_{\leq b}$Q.

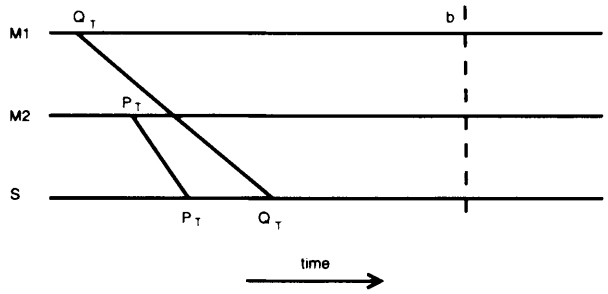



Figure 4.4: A failure is erroneously detected for P⇒ $\Box_{\leq b}$Q.

Delays in message transmission can also cause problems when a message is sent just before the time bound of a goal is reached. For example, in figure 4.6, an achieve goal is being monitored. The event $Q_T$ is sent just before the time bound is reached but is not received until after the time bound. This means that the monitor will detect a failure because it will not have detected a $Q_T$ before the time bound is reached. In fact, no failure occurred, because the event $Q_T$ actually occurred before the time bound.

A similar situation occurs in figure 4.7 where the event $Q_F$ is sent before the time bound but not received until after. The order of events should result in a failure of the goal because 'Q' ceased to hold before the time bound was reached. The monitor does not detect this failure because it does not receive the message $Q_F$ until after the time bound is reached.

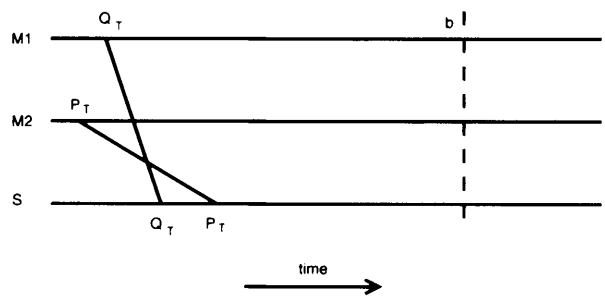



Figure 4.5: A failure is not detected for P⇒ $\Box_{\leq b}$Q.

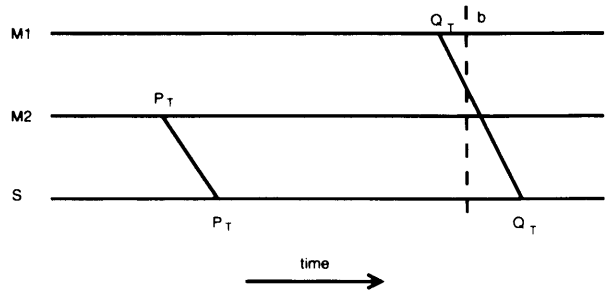



Figure 4.6: A failure is erroneously detected for P⇒ $\Diamond_{\leq b}$Q because the event Q is not received until after the time bound.

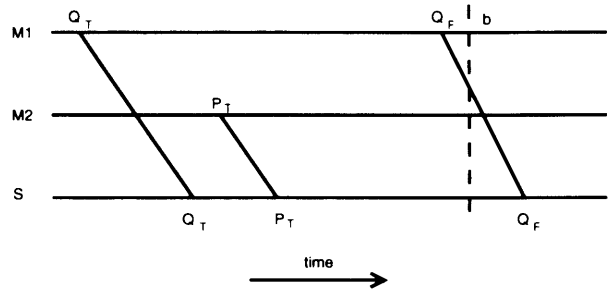



Figure 4.7: A failure is not detected for P⇒ $\Box_{\leq b}$Q because event ¬Q is not received until after the time bound.

## Approaches to Message Ordering

There are several possible approaches which can be taken to this problem of message ordering. The simplest approach is to simply ignore the communication delay completely and treat communication as instantaneous. Using this approach, all events are considered to have occurred when they are received by the monitor server and are processed as they arrive in order of arrival. As has been shown, this approach can result in both erroneous detection of failures and non-detection of failures if events are closely spaced compared to the communication delays and delays between distributed components are variable. If the communication is sufficiently quick compared to the separation between events, or the delays are uniform for different components, then this approach is feasible and has the benefit of being far simpler than the other options. Such an approach is not suitable if absolute accuracy is required but may be good enough, particularly when the goals are being monitored to facilitate soft goal monitoring rather than to detect individual failures.

Ideally, rather than assuming events occur at the time they are received by the monitor, the event messages should be time stamped with the time the event occurred in the target system. The problem with this is that it requires the clocks of the distributed components to be synchronised. This is a difficult problem to solve[Lamport 78] because physical clocks do not run at a uniform rate and because synchronisation messages for setting the clocks are also subject to delays. Fortunately considerable work has been done in this area resulting in the *Network Time Protocol*[Mills 91], which has been implemented on most operating systems. This protocol allows the local clocks of the distributed machines to be synchronised to a high degree of accuracy (typically within

a few milliseconds). This accuracy should be sufficient for the types of systems which the monitoring framework is intended to operate with.

By synchronising the clocks of the machines in a distributed system, and adding the time stamp to the event message, the monitor can determine what time events occurred rather than when they are received. There is however still a problem determining when an event can be processed as events can no longer be processed in order of arrival but must instead be processed in order of their time stamps. It is always possible that another event could be received with an earlier time stamp than an existing message which would change the result determined by the monitor.

The easy solution to this problem is to set a maximum delay which will be tolerated. Messages are sorted by their time stamps as they are received. They are then processed in order of their time stamps once those time stamps are older than the maximum allowable delay. This approach has the disadvantage that there is always a possibility that a message will exceed the maximum delay which could invalidate messages which have already been processed.

The approach used in this thesis is more complex, but guarantees accuracy as long as the local clocks are synchronised. In this approach all the machines in the target system inform the monitor at regular intervals that they have no more messages to send before a given time stamp by sending a *coordination* message. The server then stores the time stamp of the most recent coordination message for each target machine. Normal messages can be processed, in order from oldest to most recent, as long as their time stamp is earlier than all these time stamps.

This approach requires that the messages from a given machine are received in the same order they are sent. This is a reasonable assumption as most communication systems can satisfy this requirements. TCP guarantees this for Internet Protocol communications for example.

This approach is also more complex than other approaches and requires additional messages to be sent over the network resulting in greater bandwidth usage and a greater performance penalty for the target system. This approach does however guarantee that messages are processed in the correct order as long as the clock synchronisation is sufficiently good.

An additional benefit of this approach is that it can detect communication failures between the target system and the monitor server as it will always be expecting a message even if no events have occurred. A maximum period can be set after which the monitor will report a failure if a coordination has not been received. This could either be caused by a failure in the target system which, has prevented the coordination message from being sent, or it could be a failure of the communication channel between the target system and the monitor. Regardless of the cause, the users of the monitoring framework need to be informed so that the problem can be investigated.

### 4.1.5 Synchronous and Asynchronous Temporal Logic

KAOS uses a synchronous temporal logic, in which the system is viewed as a series of states at fixed time intervals. Zero, one or more events may occur during the transition from one state to the next. In contrast, asynchronous temporal logic views the system as states which change after each event occurs. Exactly one event occurs during each state transition[Letier 05]. This is the model used by the monitoring system.

This issue can cause problems in monitoring. In KAOS, several events can happen in the transition from one system state to the next, within one time unit. In the system,

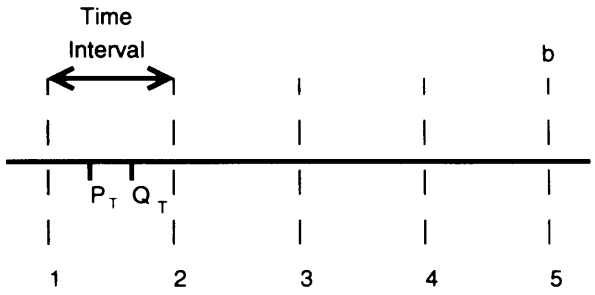



Figure 4.8: This ordering of events will satisfy the goal P⇒ $\Box_{\leq b}$Q in the synchronous view but the goal will fail in the asynchronous view.

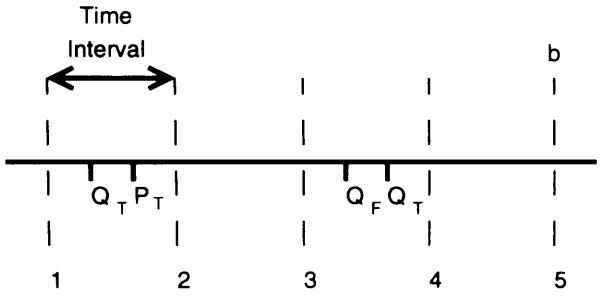



Figure 4.9: This ordering of events will again satisfy the goal P⇒ $\Box_{\leq b}$Q in the synchronous view but will not be satisfied in the asynchronous view.

the monitor will update after each event. This problem can cause some failures to be detected erroneously and other failures to be missed.

Figure 4.8 shows a timeline in which the states in a synchronous temporal logic are shown as dotted lines. Two events, $P_T$ and $Q_T$ occur between states one and state two. In the synchronous view, the goal P⇒ $\Box_{\leq b}$Q is satisfied by this ordering as in state one neither of the conditions $P$ or $Q$ are true and in state two they are both true. In the asynchronous view, this goal fails after $P_T$ occurs because the condition $Q$ does not hold in the interval between the two events. To satisfy this goal in the asynchronous view, $Q$ must be true before $P_T$ occurs.

Figure 4.9 shows another time line with four events. The goal P⇒ $\Box_{\leq b}$Q will again be satisfied by the synchronous view but will fail in the asynchronous view. The events between states three and four cause the condition $Q$ to be false briefly but $Q$ is true in both state three and four so the goal does not fail in the synchronous view. In the asynchronous view, the state changes after each event so the goal fails when $Q_F$ occurs.

Figure 4.10 shows a case where the goal P⇒ $\Diamond_{\leq b}$Q fails in the synchronous view but is satisfied in the asynchronous view. In the synchronous view, $Q$ is not true in state three or four (or any other state) as the event $Q_F$ sets it to false before the next state occurs. In the asynchronous view, as soon as the event $Q_T$ occurs, the goal is satisfied.

Despite these problems, the monitoring system uses an asynchronous view of the temporal logic specifications. This is done because although KAOS takes a synchronous view, the discrete nature of time is a convenience for modelling rather than a

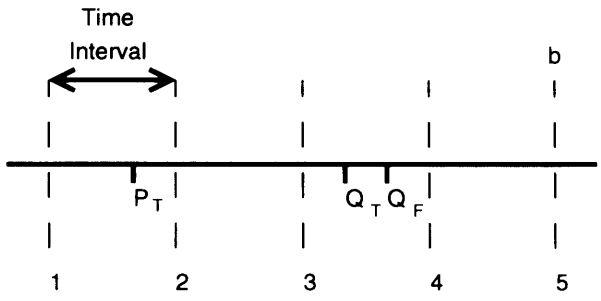



Figure 4.10: This ordering of events will fail to satisfy the goal P⇒ $\Diamond_{\leq b}$Q in the synchronous view but the goal will be satisfied in the asynchronous view.

feature which is desired in the implementation. KAOS does not define the size of the time interval used but rather leaves this as an implementation detail. At the implementation level, the discrete nature of time is likely to disappear as different decisions are made as to allowable intervals for different goals. The monitoring framework accepts that some inconsistencies between the two views may occur and leaves it to users to intelligently interpret monitoring results.

### 4.1.6 Object Identity

When the system being monitored is made up of distributed components, a problem which can occur is that the several of the monitored components may refer to the same entity. The same requirements level entity may be represented in different ways in different distributed components but the monitor must have a consistent way of recognising each entity so that it tell when different components are referring to the same entity. This is done by giving each entity instance an identifier string. If two distributed components refer to the same entity then they should use the same entity identifier. Similarly, if they refer to different entities then they should use different entity identifiers.

An example of this problem in the Limewire example is the relationship 'ConnectedTo(Client c1, Client c2)' which indicates indicates that one client is connected to another so that they can pass Gnutella messages between themselves. To create an instance of this relationship, it is necessary to identify the client objects involved. At the implementation level, a remote client is represented by a `Connection` object which manages the connection to a particular client. The local client is not really represented explicitly but it can be considered to be represented by any object for which a single instance is created for a single client, such as the single instance of the `RouterService` class which connects the back end client to the graphical user interface. The problem is solved by using the IP address of the machine which a client is running on to identify the client. As clients should know their own IP address and will know the address of any client they are connected to, they will be able to use the same identifiers to refer to the same client. As there should be only one client running on each machine, each client entity will have a unique IP address. There are additional problems which might occur due to the use of network address translation(NAT), meaning that each client does not in fact have a unique IP address. In this situation, some other approach has to be found. One possible solution would be to combine the IP address of the peer on the local network with the global IP address of the NAT router.

If two distributed components both refer to the same entity, it should be possible to determine an identifier that can be used by both components to refer to the entity. In the worst case, it should be possible to construct an identifier from all the values of the attributes of an entity which will identify it.

In cases where a particular entity only needs to be referred to by a single distributed component, it is not necessary for the instrumentation developer to choose an identifier as any random identifier can be used as long as it is used consistently. An identifier which is unique across all distributed components can be constructed by combining a randomly generated, locally unique identifier with the IP address on which the component is running.

### 4.1.7 Effects of Instrumentation

When monitoring a system, a serious concern is the effect instrumentation can have on the execution of the target system. These changes impact the validity of the monitoring results as the monitored system can behave differently from the un-monitored system. This is actually a less severe problem when monitoring is a permanent feature of the deployed system as when monitoring is used during testing only, and the instrumentation is removed prior to deployment, there is a danger that the system behaviour can change when the instrumentation code is removed thereby invalidating the monitoring results. When monitoring is also used in the deployed system there is no change in the system and so no change in behaviour caused by monitoring. Monitoring after deployment does however place greater demands on the performance of the instrumentation code. When monitoring takes place only during testing, a performance penalty can sometimes be tolerated which would not be acceptable in the deployed system.

The only way to achieve instrumentation with absolutely no effect on the monitored system is to use dedicated hardware which allows execution data to be collected as the system runs with no performance overhead. This is a valid approach for some high performance embedded systems but is not possible with most computer systems which run on general purpose hardware.

#### Performance Impact

The performance overhead caused by instrumentation is limited in the monitoring framework because the instrumentation only emits significant events. Significant events are those which correspond to changes in the requirements level object model. This is in contrast to systems which operate using a typical debugging approach in which every method call and the value of every variable are tracked. The performance impact of such debugging systems is generally too high to be used in a deployed system.

The instrumentation approach used means that it is hard to quantify what the impact of instrumentation is on the performance of a target system because it is heavily dependent on exactly what parts of the system are monitored. Instrumentation could generate events many times a second or only a few times an hour depending on the system and what is instrumented. It is however useful to evaluate the average performance overhead of a single message. This gives some indication of what performance overhead can be expected in a given scenario assuming the frequency with which messages will be emitted can be estimated, although there could be scalability issues which occur when messages are generated at a high frequency. It should also be possible to determine typical overheads for specific types of systems.

The performance overhead of an individual message depends on the method used

to capture information from the system, the complexity of the instrumentation code itself and the method used to communicate with the monitor. This last issue is a particular problem for distributed systems as it is necessary to send messages over a network connection which can be a source of significant performance overhead. It can however be advantageous to use a remote monitor, even when the target system is not distributed. The gain in performance by avoiding remote communication can be offset by the loss caused by running the monitor on the same machine as the target system.

A concern when monitoring distributed or multi-threaded systems is that changes in performance can also affect the behaviour of a system because changes in performance can actually affect the order in which actions occur within the system. Systems which behave in this way are generally undesirable in any case but it is possible that the instrumentation can mask such problems, by changing the behaviour of the system, rather than detecting them. This is not such a big concern when monitoring is part of the deployed system because the instrumentation will continue to have the same affect on the system after deployment and any failure which actually occurs will still be detected.

### Functional Impact

As well as affecting the performance of a monitored system, instrumentation can change the actual behaviour of the system directly which is obviously undesirable. The purpose of instrumentation should be to observe the system not to change it. This problem is handled quite effectively in [Minsky 96] which allows the developer of the target system to specify that certain methods should be side effect free. The target system operates within an environment which ensures that these methods really are side effect free by checking that they fulfil certain conditions such as not assigning a value to an object attribute. This approach limits the target system to operating within a *law-governed* architecture and so it is a significant restriction on the developer. It still also requires the developer of the target system to mark methods as being side effect free, although it automatically checks that they remain so in the future.

The approach used by the monitoring framework described in this thesis uses the much less restrictive AspectJ language for instrumentation. Unfortunately, AspectJ has no way to ensure that actual behaviour is not affected. The developer must take care, particularly when calling methods in the target system. These methods should obtain information only and should not have side effects. Accessor type methods which simply return a value stored by a class are generally the safest type of method but it may sometimes be necessary to call methods which perform more complex calculations to return a value. This may involve calling other methods on other classes and these must be side effect free if the calling method is to be side effect free. It is also possible that a method which was side effect free when the instrumentation was developed could later be modified so that it has side effects. To avoid this, all developers should carefully document their methods to specify whether they are allowed to have side effects. A method which has been documented as being side effect free should not be changed to have side effects at a later date.

## 4.2 Monitor Server Implementation

This section discusses the implementation of the monitor server part of the monitoring framework. The architecture of the monitor server is first discussed. The monitor server is split into two parts; the requirements instance model component and the goal

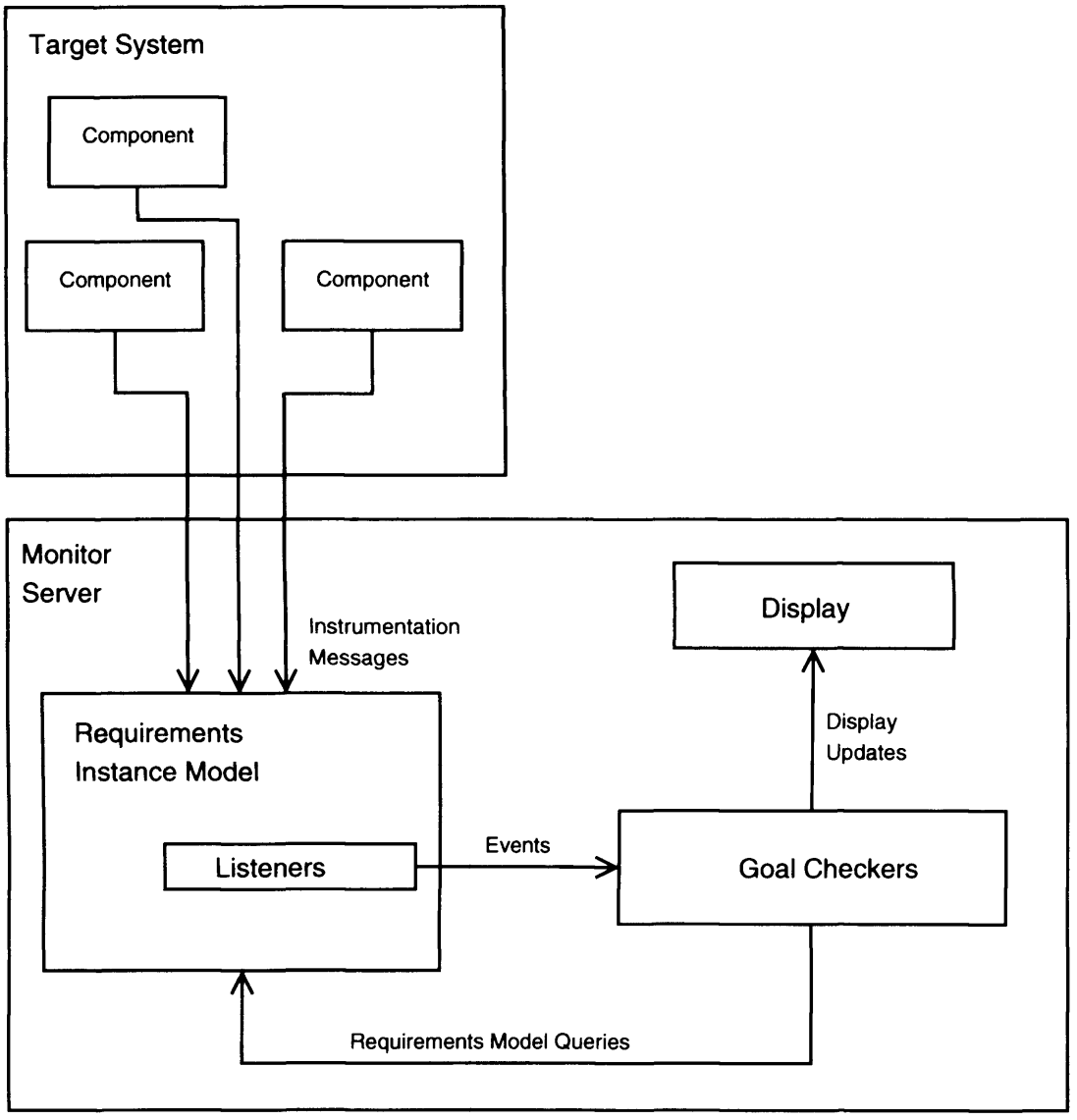



Figure 4.11: The architecture used by the run-time monitoring framework.

checker component. The requirements instance model component represents a particular instance of the KAOS object model which reflects the state of the target system at run time. The goal checker uses this model to check whether specific KAOS goals are violated. These two components of the monitor server are described in detail in this section.

### 4.2.1 Monitor Architecture

The requirements monitor is made up of the requirements instance model and goal checkers as shown in figure 4.11. At run time, the instrumented system emits messages which are received by the monitor server. These messages contain requirements level events which describe changes which should be applied to the KAOS object model of the system so that it reflects the current state of the target system. These messages are received by the requirements instance model component which updates its model of the system using the events it receives.

The goal checker communicates with the requirements instance model to check

specific KAOS goals, specified in temporal logic, and determine whether these goals are satisfied or have failed. This is done by a combination of listeners and queries. When using a listener, the goal checker asks the requirements instance model component to inform it whenever certain types of events occur. A listener might inform the checker which registered it whenever a new instance of a particular relationship is instantiated or the value of an attribute changes. The goal checker can also query the requirements instance model component for some piece of information about the current state of the model. An example would be to ask for all the instances of a particular entity type.

The requirements instance model component only stores information about the current state of the requirements model so individual goal checkers must store the specific temporal information that is needed to monitor particular goals. This ensures that the historical information which needs to be recorded is kept to a minimum as only information which is relevant to a monitored goal is stored.

### 4.2.2 Requirements Instance Model

At run time, the state of the system is represented by the requirements instance model. This model is an abstract representation of the state of the system (at the level of the requirements specification). The requirements instance model contains instances of KAOS entities and relationships. The values of entity attributes are also stored.

There are two different implementations of the requirements instance model included in the monitoring framework. The first uses a relational database to implement the model. The database itself cannot contain listeners so this component also has additional code to query the database at regular intervals to detect changes which are relevant to the goal checker. The goal checker uses SQL to query the database directly. The instrumentation also uses SQL to modify the requirements instance model.

The second implementation stores the requirements instance model in system memory as Java objects which represent instances of entities, requirements and attributes in the KAOS object model. Listeners inform the goal checker of changes and the model can be queried by calling methods on the model which allow details of the model to be accessed.

The monitoring framework makes both these alternatives available so that the monitor developer can decide which implementation best suits a given monitoring problem. These two approaches each have their own advantages and disadvantages. The object approach is more responsive, at least for small systems. This is the case because the database approach involves querying the database at intervals whereas the object approach informs the goal checker immediately that a change occurs. The database approach may scale better to larger systems and be more robust although it has not been possible to test if this is true.

There are some limitations of these two approaches as they are implemented. The object approach only implements monitors for hard goals. Soft goal monitoring is not implemented as the method used for the implementation of soft goals relies on generating SQL queries to calculate the value of the soft goal metric, as will be described in chapter 5. There is no reason that soft goal monitoring could not be implemented for the object approach but this would require a separate strategy for interpreting the soft goal specification and evaluating it by calling methods on the requirements instance model component.

The database approach assumes that there is no delay in receiving instrumentation messages. Events are assumed to occur when they are received by the database.

The object approach, in contrast, implements message ordering and uses coordination messages to determine when they can be processed.

The database approach is likely more suited to long running target systems where instrumentation messages are relatively infrequent. The object approach is likely more suited to systems with frequent instrumentation messages but smaller systems.

## Database Implementation

In this implementation, the requirements instance model is represented using a relational database. The database used in the monitoring framework is MySQL which was chosen primarily because it is open source and is available for free. There is no particular reason why the monitoring framework should not work with other relational databases which use SQL as a query language. MySQL is relatively lacking in features compared to commercial relational databases so most other databases should also implement the features which are used by the monitoring framework.

Because the database itself is unable to register listeners and inform the listening object when changes occur, it is necessary to include code which carries out these tasks by periodically checking the database for changes and informing the goal checkers of these changes. This approach also increases the complexity of the database as it must keep track of which changes have still to be processed and allow them to be processed in the correct order.

The database schema is shown in figure 4.12. The entity table in the schema stores instances of KAOS entities with each row in the table representing one instance. The entity instance is an instantiation of the entity named in the 'type' field. The 'idString' field is used to identify the entity instance as discussed in 4.1.6.

The 'attribute' table is used to store changes to attribute values. All attribute values are stored in the 'value' field, represented as strings regardless of their actual type. The type of the attribute is stored in the 'value_type' field and determines how the value is interpreted. For example, if the 'value' field contains the string '100' the it can be interpreted as a string or if the type of the value is an integer it will be interpreted as a number. The identifier of the entity to which the attribute belongs is stored in the 'entity_id' field and the name of the attribute is stored in the 'name' field.

Each entry in the 'attribute' table represents a change in the value of that attribute. The 'new' field is set to 'TRUE' by default when a row is added to the 'attribute' table. The current value of an attribute always has the 'new' field set to 'FALSE' and this value is always used when the goal checker wants to obtain the current value of an attribute. When a change to an attribute value is processed, the row for the old value is first deleted. The 'new' field of the new value is the set to 'FALSE' to show that it is now the current value. The 'time' field is automatically set to the time at which the attribute change occurs and is used to process attribute changes in the correct order.

This mechanism of creating a new row whenever the attribute value changes is necessary because the value of an attribute can change several times between the database being checked for changes. If the attribute is used in the specification of a goal then missing one of these changes could prevent the monitor from operating correctly. To prevent this, each changes in the value of an attribute must be stored individually. If the attribute is not used in the specification of a goal, but is instead only used in soft goals then it is not necessary to capture every change in value. In these cases, a single row can be used to store an attribute. Changes in the value of the attribute are implemented by modifying the 'value' attribute of the table row.

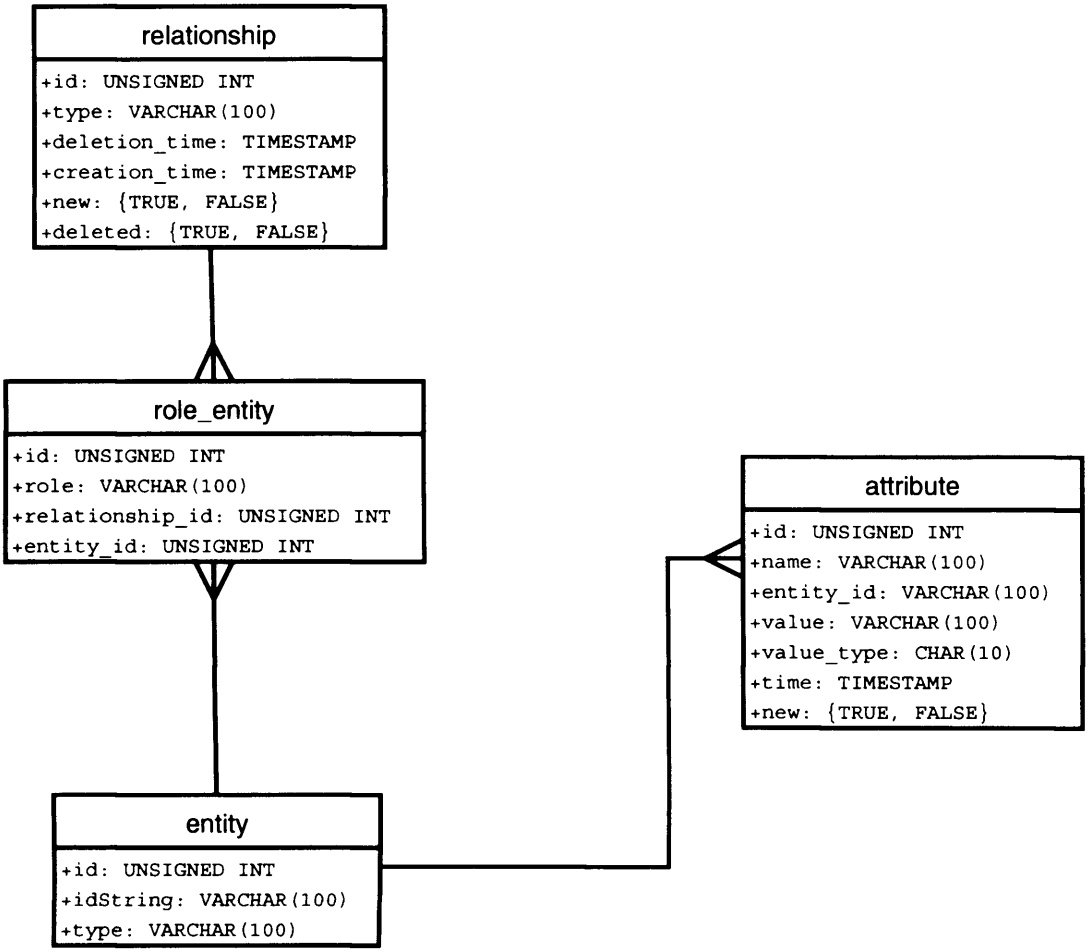



Figure 4.12: The database schema for the requirements model instance.

The relationship table stores KAOS relationship instances. The 'type' field stores the name of the relationship which is instantiated. The 'creation_time' field stores the time at which the relationship instance is created and the 'deletion_time' stores the time at which it is deleted. These two fields are used to process relationship creation and deletion in the correct order. Two fields are necessary because the database is checked for changes at intervals and a relationship could be created and deleted between two checks.

The remaining two fields are boolean values. The 'new' field is set to 'TRUE' by default for each row which is added to the table. This tells the goal checker that a new relationship instance has to be processed. Once the goal checker has finished processing the new relationship instance, the 'new' field is set to 'FALSE'. The 'deleted' field is set to 'TRUE' when the monitored system indicates that a relationship instance has been destroyed. This also tells the goal checker to process the relationship instance, after which the table row is deleted.

Each relationship has two or more roles. Each entity can be a role in zero or more relationships. To allow this many-to-many association, a junction table, 'role_entity', is used. Each row of the table represents a single role in a relationship instance. The field 'relationship_id' identifies the relationship instance to which the role belongs to. The field 'entity_id' identifies the entity instance which fills the role. Additionally, the

name of the role is stored in the 'role' field.

The requirements instance model database has to be regularly checked for changes. To find relationships which have been created or destroyed since the last check, a query is executed which returns all the rows of the relationship table for which the 'new' or 'deleted' field are set to true. These rows represent relationship instances which have been created or destroyed since the last time the database was checked. The rows are returned, sorted by the 'time' field which holds the time at which the row was created or modified. The relationship instances are processed starting with the earliest one which was modified.

When a new relationship instance is processed, the 'new' field is first set to false. Goal checkers which have listeners for the relationship type are then informed of the new instance. For relationship instances which have been destroyed, the relationship listeners are also informed and the row for that relationship instance is deleted from the database.

## Object Based Implementation

In this implementation, the requirements instance model is represented using Java objects which are stored in memory. The instrumentation messages are sent to the monitor using TCP sockets, using a single TCP socket for each machine in the distributed system.

**Message Ordering Implementation** The messages are sorted into the correct order using the system of ordering described in 4.1.4 which uses coordination messages to help ensure messages are processed in the correct order. The sorting algorithm is implemented by creating an ordered queue to store messages which are waiting to be processed. Messages are inserted into this queue in order of their time stamps with the oldest message at the front of the queue.

When messages are processed they are taken from the front of the queue. Associated with each TCP socket is the time of the last coordination message received by that socket. Whenever a new coordination message is received, the message queue is compared to the coordination messages last received by each TCP socket. If any messages are older than the oldest coordination message then those messages are processed by taking them from the front of the queue.

**Object Model Implementation** The monitoring framework builds an instance of the requirements level object model using the classes shown in figure 4.13. The classes `KAOSEntity` and `KAOSRelationship` represent the KAOS meta-model concepts of entities and relationships. Agents are not necessary here because for these purposes they can be treated identically to entities. During initialisation of the monitoring framework, these classes are instantiated for each entity and relationship mentioned in the requirements specification. These instances represent the KAOS object model for the target system.

The classes `KAOSEntityInstance` and `KAOSRelationshipInstance` are instantiated at run time to store the information sent by the instrumentation in the target system. These classes model the current state of the target system.

The goal checkers are informed of changes in the requirements instance model by attaching listeners to the parts of the model that they are interested in. If a goal checker wants to be informed about all new instances of a particular entity or relationship type, the method `addInstanceListener` of `KAOSEntity` is used to add a listener. If a checker want to know whenever a particular entity attribute changes, for all instances of

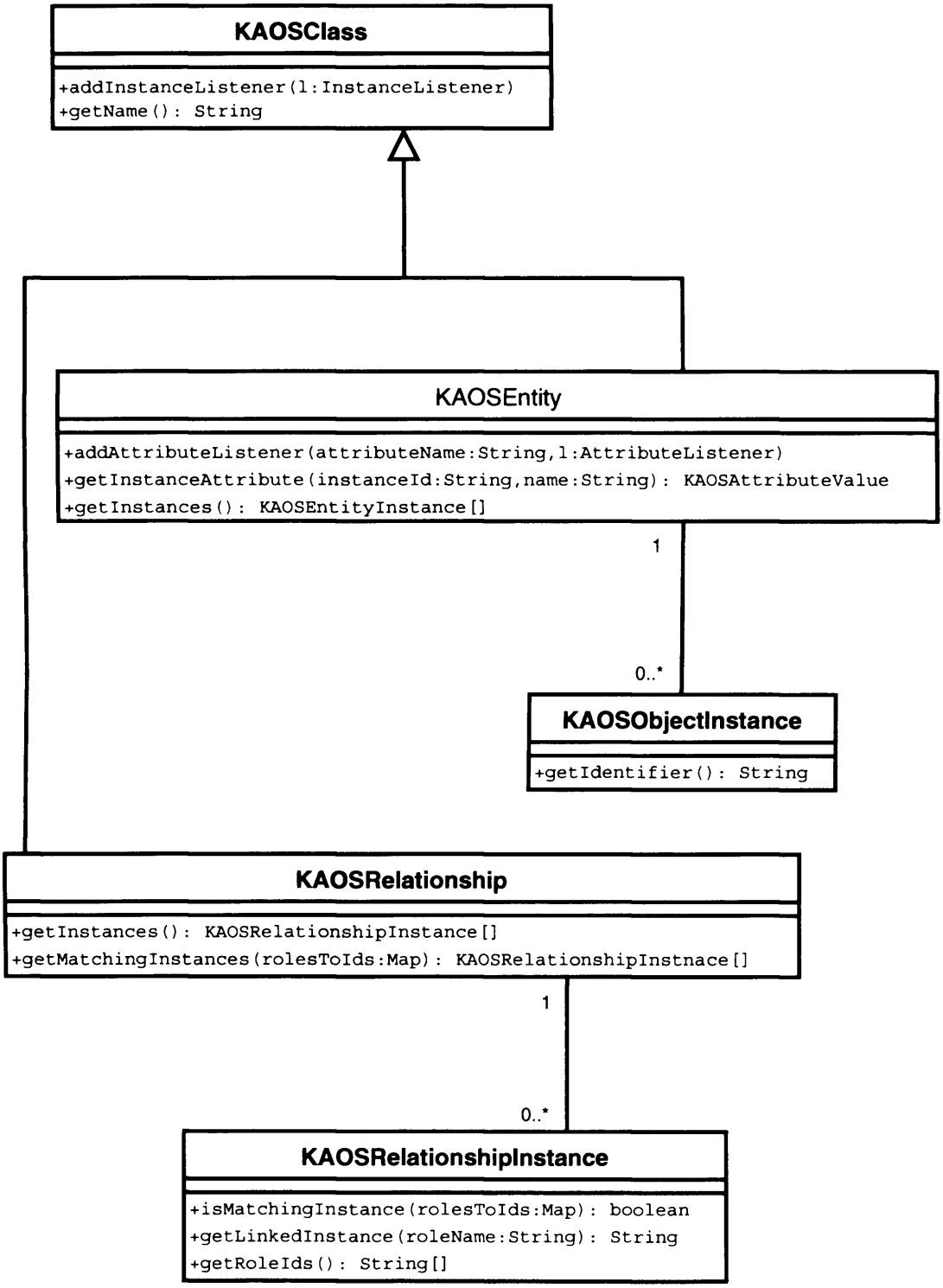



Figure 4.13: Classes used in the object based implementation of the requirements level object model.

a particular type, the method `addAttributeListener` of `KAOSEntity` is used to add an attribute listener. These listeners are then informed whenever the relevant event occurs.

As well as detecting changes through listeners, goal checkers can also request information about the current state of the model. For example, the checkers can obtain all instances of a particular type by calling the `getAllInstances` method of `KAOSClass` or `KAOSRelationship`. The instances can then be used to obtain further information such as what entities are linked by a particular relationship instance.

### 4.2.3 Goal Checker Implementation

The goal checker is responsible for detecting when goals are violated or satisfied. This is determined by examining the requirements instance model to determine the state of the target system.

The requirements instance model only stores the current state of the system. The goal checker is responsible for storing the historical information necessary for checking goals. This means that whenever a goal is instantiated, the goal checker has to store some information about the instantiation of the goal. This information is then used to determine when the goal is satisfied or fails.

A goal checker is constructed by parsing the temporal logic specification, stored in an XML format, and building up a tree of objects which correspond to the structure of the temporal logic formula. These objects together evaluate whether the system has satisfied the monitored goals. The temporal logic specification of the monitored goals are stored in an XML format so that they are easier for the monitoring framework to parse.

The root of the checker tree is always an object responsible for monitoring a particular goal pattern. The type of this object depends on the goal pattern of the monitored goal. This object will store information associated with each goal instance, particularly the time bounds associated with those goals. The root object will have a number of children which correspond to the predicates which appear in the goal specification.

An example object model, for the goal 'Download File' from the Limewire example, is shown in figure 4.14. This goal is formally defined as:

**Achieve[Download File]** $\forall$ c:Client, f:File, fd:FileDescriptor
RequestingFile(c, fd) $\Rightarrow \Diamond$ SavedFile(c, f) $\wedge$ f.name = fd.name

In this case, the goal is an 'achieve' goal so an instance of the `AchieveMonitor` class is created to check this goal.

The root object has references to objects representing predicates which appear in the temporal logic formula. There are two types of predicates in the tree; atomic predicates and compound predicates which are created by combining atomic predicates. The atomic predicates available are relationship predicates and comparison predicates. A relationship predicate is true if for a given set of objects if the relationship exists for those objects. A comparison predicate compares the value of an attribute to another attribute or a constant using operators such as 'equals', 'greater than' and 'less than'. The predicate is true if the comparison is true. Compound predicates are combinations of atomic predicates using the 'AND', 'OR' and 'NOT' operators. The checker in the example has objects for checking the relationships 'Requesting File' and 'Saved File' and an `EqualityMonitor` object which is a comparison predicate responsible for checking the equality 'f.name = fd.name'.

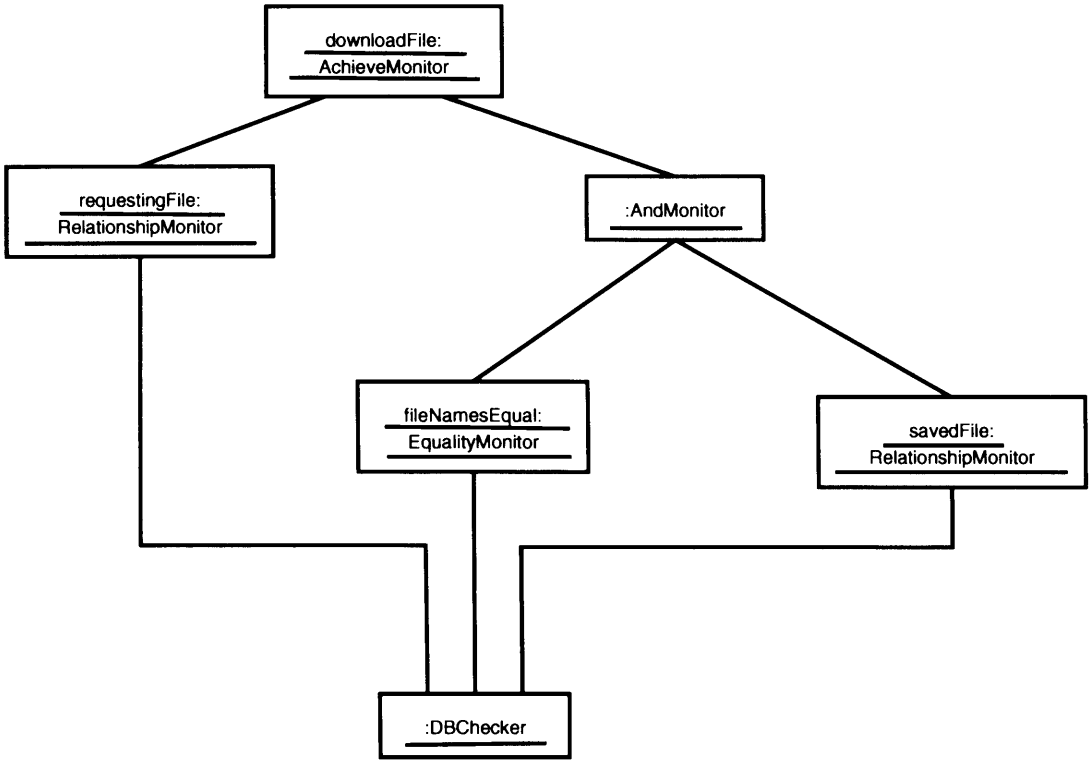



Figure 4.14: The object model for a goal checker which is checking the goal 'Download File'.

As KAOS predicates are parameterised, a predicate object is responsible for determining the truth of the predicate for any set of parameters. The parameters of a predicate are indicated by a set of labels. For example, the relationship 'SavedFile' in the example above has parameter labels 'c' and 'f'. The compound predicate 'SavedFile(c, f) $\wedge$ f.name = fd.name' has parameter labels 'c', 'f' and 'fd'. The predicate is parameterised by assigning entities to these labels.

Predicate monitors have two responsibilities. First, they must inform their parent monitor whenever the monitored predicate becomes true for a given set of parameters and what those parameters are. Secondly, when presented with a list of labels and entities which are bound to those labels, they must be able to determine if the predicate is true for those labels. The list of label bindings does not need to contain values for every label in the predicate. If an incomplete list is provided, there may be several possible bindings for the unbound labels which would satisfy the predicate and the predicate object should reports what these values are. For example, the predicate monitor for the relationship 'Requesting File' must determine whether the predicate is true for any given value of the labels 'c' and 'fd'. It must also be able to determine whether a relationship instance exists for which only one of the labels 'c' or 'fd' is fixed and what values are allowable for the unbound label.

Each atomic predicate object registers a listener with the requirements instance model. Relationship predicate objects register listeners with the relationship type they are checking. Comparison predicate objects need to register attribute listeners for any attributes used in their definitions. The goal checker begins to execute whenever one of the listeners informs the checker of an event.

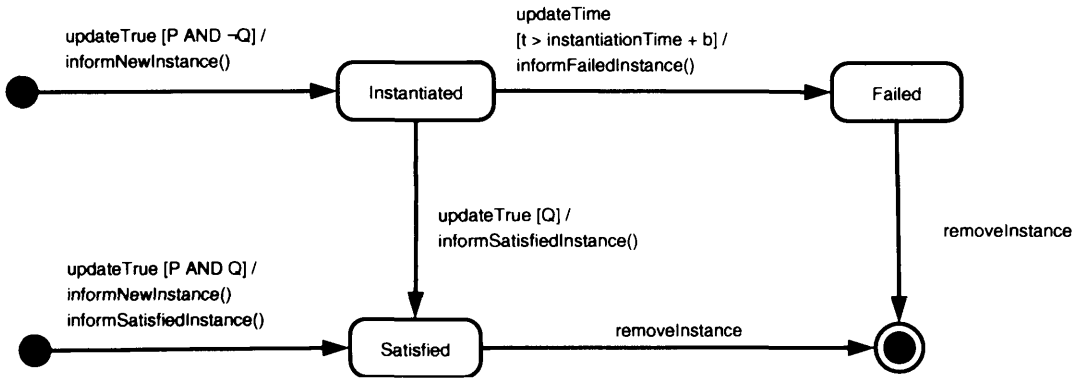



Figure 4.15: State diagram showing the implementation of a checker for bounded achieve goals (P⇒ $\Diamond_{\leq b}$Q.

### Evaluation of Temporal Logic

The monitoring framework does not implement checkers for arbitrary temporal logic formulae but instead is capable of checking only those formulae relating to specific goal patterns which are part of KAOS. This choice was made mainly because it was decided that it was easier to implement specialised checkers for specific patterns than to implement a generalised checker which can handle any temporal logic formulae. Since KAOS encourages the use of a restricted set of temporal logic formulae there is little need for a generalised checker.

Each temporal logic checker corresponds to one of the goal patterns in KAOS. A state diagram illustrating the operation of the checker for bounded achieve goals is shown in figure 4.15. A goal instance is instantiated when the condition P becomes true. That goal instance is satisfied immediately if Q is already true. If it is not then the goal is satisfied if Q subsequently becomes true. The goal fails if the time bound $t$ is exceeded.

The state diagram for goals of the 'after' invariant type, which is the most common type of maintain goal used in KAOS specifications, is shown in figure 4.16. An instance of the goal is created when P becomes true. If Q is not already true then the goal immediately fails. The goal also fails if Q subsequently becomes false. The goal is satisfied when the time bound is reached without the goal entering the failure state.

## 4.3 Instrumentation for Monitoring KAOS Goals

Instrumentation performs two roles within the monitoring framework. Firstly, the instrumentation code is responsible for collecting information from the target system about the execution of the system. Secondly, the instrumentation has to translate that information into a form which can be used by the monitor to determine whether the goals in the requirements specification are being satisfied by the target system.

The type of events which the monitor understands are creation of a KAOS relationship or entity instance, destruction of an instance or a change in the value of the attribute of an entity. The instrumentation must then gather events from the target system such as execution of method, creation of an object and changes to a variable. The instrumentation must then translate these implementation level events into requirements

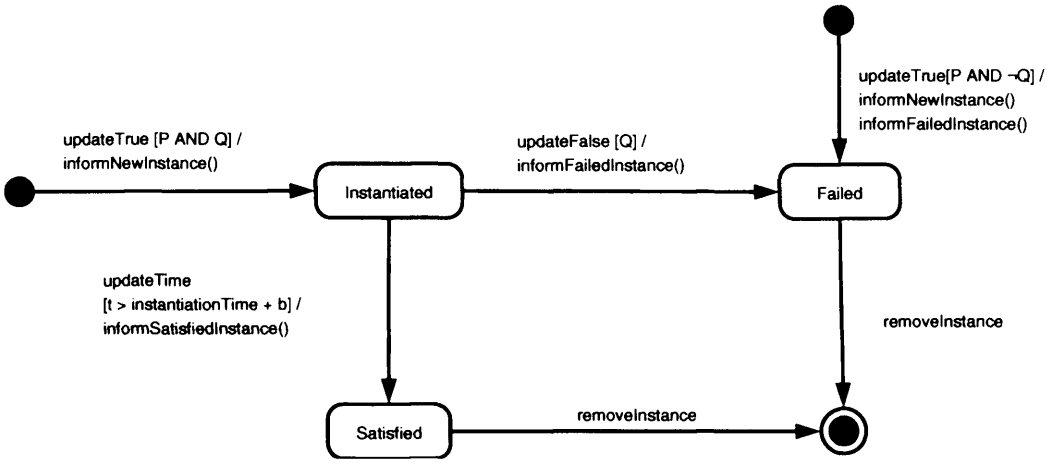



Figure 4.16: State diagram showing the implementation of a checker for 'after' invariant maintain goals (P⇒ $\Box_{\leq b}$Q.

level events.

Two approaches have been provided for creating this instrumentation code. Both attempt to simplify the task of creating instrumentation for requirements monitoring using KAOS goals. The first approach described is to write instrumentation code using AspectJ. These instrumentation aspects are supported by classes generated from the requirements specification. The second approach described is to use a mapping from the requirements level relationships and entities to the implementation level to automatically generate instrumentation aspects. These two approaches each have their advantages and the two can be used in combination if necessary.

The rest of this section describes these two approaches to mapping with the help of an example based on the Limewire system. The example uses the relationships and entities used in the specification of the goal 'Achieve[Download File]' which was initially described in section 3.3.2 and formally defined in section 4.2.3. This goal specification makes use of the entities 'Client', 'File' and 'FileDescriptor' as well as the relationships 'SavedFile' and 'RequestingFile'. To allow the goal 'Achieve[Download File]' to be monitored, it is necessary to create instrumentation code which allows these entities and relationships to be monitored.

### 4.3.1 Instrumentation Process

The instrumentation process is illustrated in figure 4.17. The instrumentation code is arranged into three packages, arranged in layers, with classes in the lower layers being extensions of classes in the upper layers. These layers correspond to the three levels of KAOS models. The top layer contains classes which correspond to the KAOS meta-model, the middle layer contains classes which correspond to the KAOS domain model and the lower layer contains classes which correspond to the KAOS instance model.

The classes in the top layer represent the concepts of entities, relationships and attributes. These classes are responsible for communicating changes in the KAOS instance model to the monitor server. Because these classes represent the KAOS meta-model concepts, they are the same for all monitored systems.

The classes in the domain level package represent specific relationships and entity types. These classes support the developer in writing the instance level aspects by providing an interface which contains all the allowable events which can occur for the

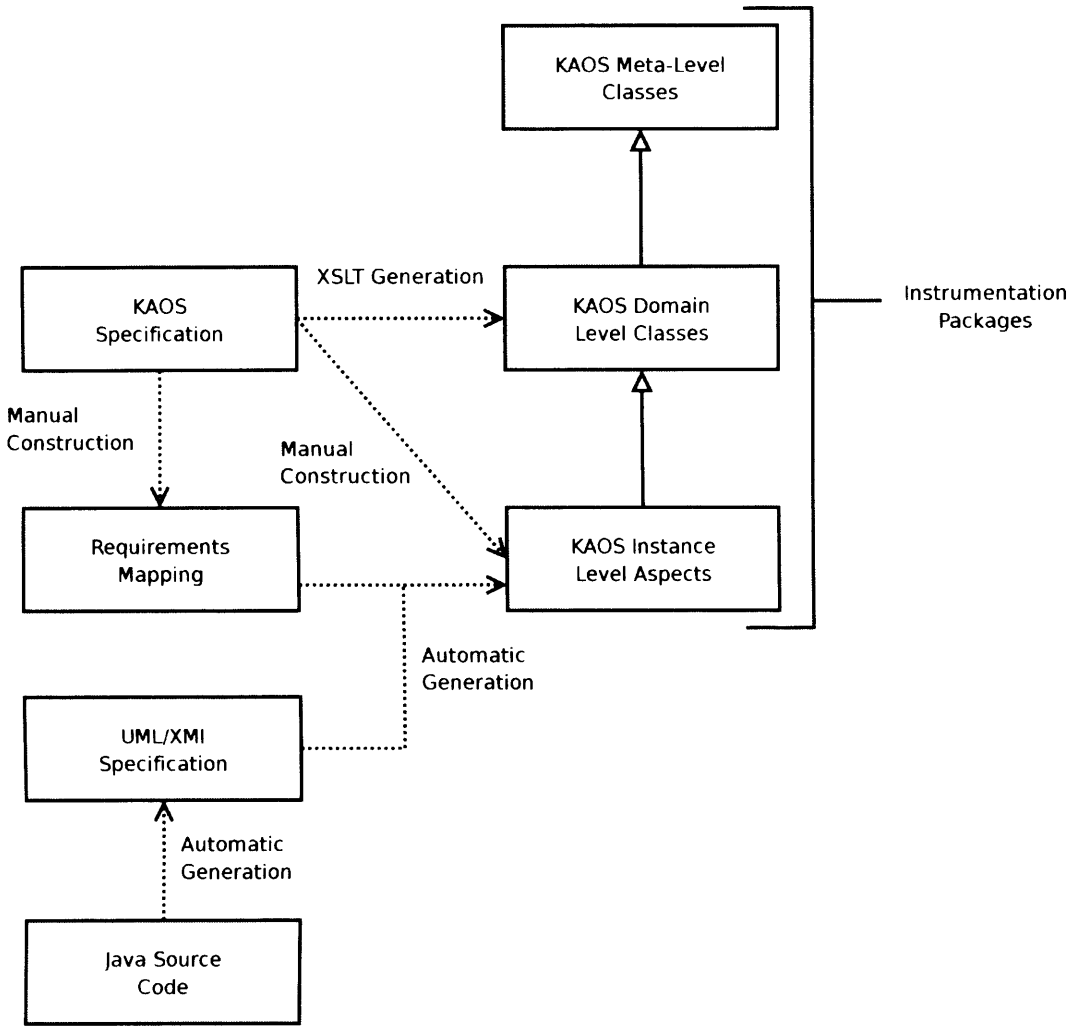



Figure 4.17: The instrumentation process.

particular domain model for the monitored system. For example, only relationships, entities and attributes which are named in the KAOS object model can be referred to in changes to the instance model. Similarly, when new relationships are instantiated, they must have the correct number and types of roles.

The classes in the domain level package are generated automatically from the KAOS specification. This is done by reading an XML representation of the specification and generating Java source code from it using XSLT[W3C 05]. A class is generated for each entity and relationship type which exists in the KAOS specification. These classes have methods which can are called to create relationships and update attribute values. These classes pass these changes to the meta-level classes which communicate this information to the monitor.

The KAOS instance level aspects do the job of actually instrumenting the monitored system to obtain information on its execution. The aspects then translate this implementation level information into events relating to the KAOS instance model by calling methods in the domain level package. These calls represent the creation, destruction or modification of individual instances in the KAOS model.

There are two ways of creating the instance level aspects, as shown in the di-

agram. The first method is for the instrumentation developer to write AspectJ code manually, making use of the KAOS domain level classes to do so. AspectJ pointcuts are used to obtain implementation level events. The advice attached to these pointcuts translates implementation level events to requirements level events by calling methods in the domain level package. This is done by using information contained in the parameters of the pointcut and additional information which can be obtained from the monitored system to determine the values which should be passed to the classes in the domain level package. The second method is for the instrumentation developer to manually construct a requirements mapping which maps relationships and entities in the requirements specification to Java classes, methods and attributes. AspectJ code is then automatically generated from this mapping and information about the implementation coded in UML/XMI format.

### Example Class Model

An example of the classes generated from the specification for the goal 'DownloadFile' for the Limewire system is shown in figure 4.18. In this case there are three entities called 'Client', 'FileDescriptor' and 'File'. A class is generated to represent each of these types. The monitor is informed of the new entity when the method `initInstance` is called. The no argument version of this method automatically generates a globally unique identifier for the entity. The other version of the method provides a `String` argument which is used as the identifier for the entity. The 'File' entity has an attribute called `name` which can be modified by calling the `nameUpdated` method of the `FileType` class.

There are also two relationships called 'RequestingFile' and 'SavedFile' in the specification of the goal and a class is generated for each of these relationships. The roles of these relationships are set, and the relationships created in the monitor's requirements model, by calling the `initInstance` methods on the classes representing these relationships. The instances are removed from the requirements model by calling the `destroyInstance` method.

An example of generation of the domain level classes is illustrated in figure 4.19. The upper-left of the diagram shows the specification of the goal 'DownloadFile' represented in XML (the requirements specification only contains goal as the object model is derived implicitly from the goal specification). This goal contains a relationship called 'RequestingFile' and the lower-right part of the diagram shows the class that is generated for this relationship. The arrows show the areas of the domain level class which are filled in using information from the requirements specification.

## 4.3.2 Instrumentation and Translation Using AspectJ

An instrumentation aspect is required for each relationship and entity in the KAOS object model. As stated previously, instrumentation aspects can either be written manually or they can be generated automatically from a mapping between the requirements model and the implementation of a system. In either case it is necessary to consider how KAOS relationships and entities relate to the implementation in code.

### Entities

KAOS entities are mapped onto the implementation level by relating a KAOS entity to one or more implementation classes. A KAOS entity can often be mapped onto an implementation level class on a one-to-one basis. Entity attributes may also map onto member variables of the implementation object. Such mappings are very easy to

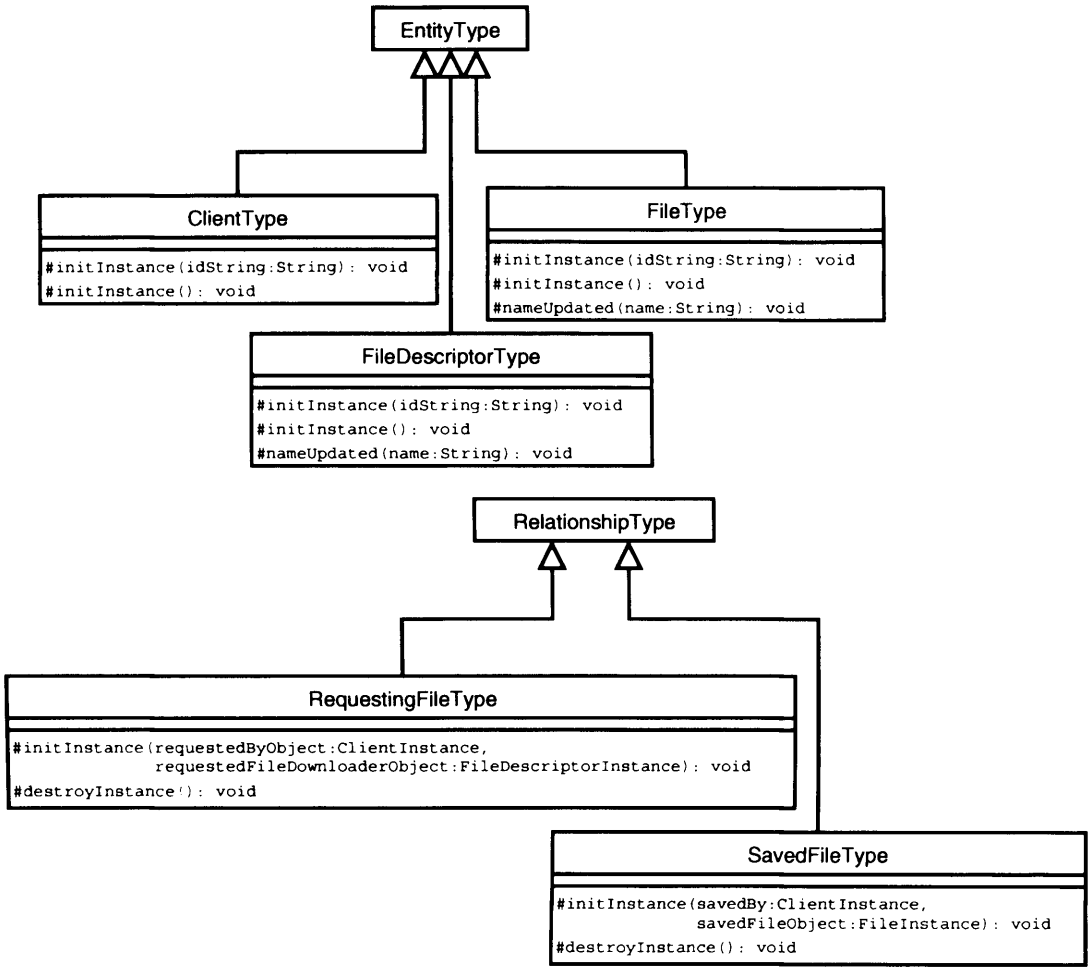



Figure 4.18: Type model generated from requirements specification for Limewire system.

specify as it is simply necessary to show which entities correspond to which implementation classes and which attributes map to which member variables. An example of this type of mapping is demonstrated by the instrumentation aspect for the entity 'FileDescriptor' in the specification of the goal 'Achieve[Download File]'.

```
1  public aspect FileDescriptorInstance
2      extends FileDescriptorType
3      pertarget(execution(RemoteFileDesc.new(..))) {
4
5      after(RemoteFileDesc f) :
6          execution(RemoteFileDesc.new(..)) &&
7          target(f) {
8
9          String name = f.getFileName();
10         initInstance(name);
11         nameUpdated(name);
12     }
13 }
```

The entity 'FileDescriptor' maps directly to the `RemoteFileDesc` class in

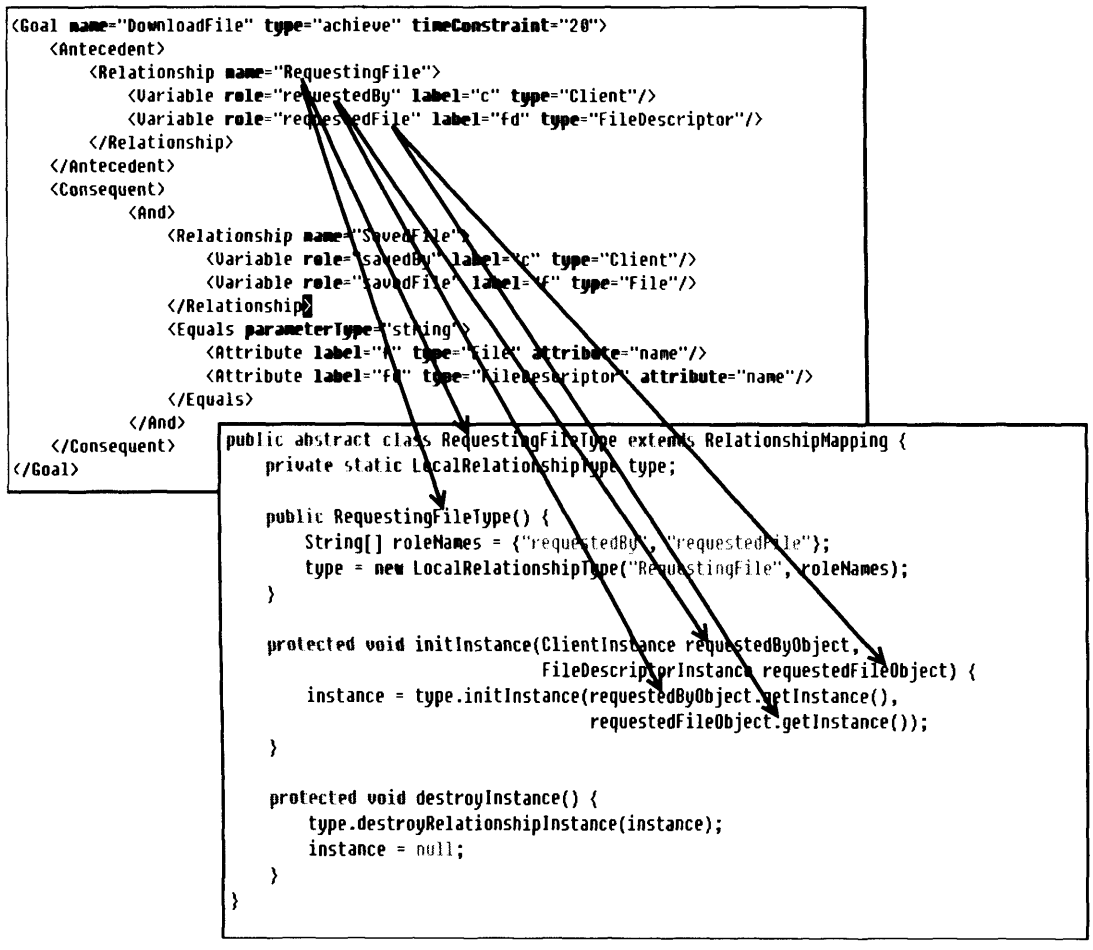



Figure 4.19: Generation of classes from the specification of the goal *DownloadFile*.

the Limewire implementation. To implement this in AspectJ it is necessary to use a `pertarget` clause (line 3) so that an instance is created for each instance of `RemoteFileDesc` which is created. The advice at line 5 executes after the constructor of the `RemoteFileDesc` class. This advice indicates that the monitor server should be informed that the entity has been instantiated by calling the `initInstance` method (line 10). The value of the 'name' attribute is also set at this time by calling the `nameUpdated` method (line 11). Since the file name does not change, it is not necessary to change the value of this attribute after the initial instantiation of the aspect.

More complex mappings are also a possibility. For example, a KAOS entity can map onto more than one implementation class or onto a particular state of an implementation class. An example of a more complex mapping is the 'Client' entity in the Limewire example. Each client has knowledge about both itself and the other clients it is directly connected to. Each client is identified in the monitoring system by the IP address of the machine it is running on. This is represented in the following aspect.

```
1  public aspect ClientInstance extends ClientType
2      pertarget(execution(DownloadManager.new(..)) ||
3                execution(ConnectionManager.new(..)) ||
4                execution(* Connection.initialize(..)) ) {
5
6      before() : execution(DownloadManager.new(..)) ||
```

```
7                   execution(ConnectionManager.new(..)) {
8
9           initInstance(Util.getLocalIP());
10      }
11
12      after (Connection c) returning :
13          execution(* Connection.initialize(..)) &&
14          target(c) {
15
16          initInstance(c.getInetAddress().getHostAddress());
17      }
18  }
```

Instances of this aspect are created for each `DownloadManager` and `ConnectionManager` which are created. Each of these classes are instantiated once for each client so the client can be associated with either of these objects. Both objects are used for convenience as in some cases it is easier to use the `DownloadManager` object to identify the client instance and in other cases the `ConnectionManager` object is easier to use. On line 9, an entity instance, identified by the IP address of the machine the client is running on, is created whenever either of the objects is instantiated. In a normal execution of the system, both of these classes will be instantiated once meaning either object can be used to access the client entity representing the local client. As the entity created will have the same identifier (the IP address) in either case, there will only be one 'Client' entity created on the monitor server. The second call to `initInstance` will have no effect on the instance model stored on the monitor server.

Instances of the 'Client' entity are also created for each 'Connection' object which is instantiated. These entities represent the remote client which the local client is connected to. On line 16 a 'Client' entity is created for each connection, identified by the IP address of the remote end of the connection. If the remote client is also being monitored and is connected to the same monitor server then an entity on the server may already be created with the same identifier. In this case, the second attempt to create the entity is ignored as in both cases, the same 'Client' entity is being referred to.

## Relationships

The aspects written to provide instrumentation for KAOS relationships make use of the aspects for entities as these aspects identify the roles in the relationships. The aspect representing the entity instance can be recovered from the implementation object it is associated with, using the `aspectOf` method. The `aspectOf` method is a static method which is automatically added to all aspects by AspectJ. It gets the aspect instance of a particular type which is associated with the object provided as a parameter. For example, if the aspect `A` is defined as:

```
aspect A pertarget(execution(B.new(..))) {}
```

then given an object `b` which is an instance of `B`, the corresponding aspect instance is returned by the call:

```
A.aspectOf(b)
```

It has been found that KAOS relationships normally map onto the implementation level in one of two ways. A KAOS relationship can either maps onto a member variable of an class or it can map onto a method call.

In the first of these two cases, the relationship is true when a member variable has a certain range of values. This could simply be when a member variable which is a reference to another object is non-null, representing a relationship with another class at the implementation level. In this case the object referred to is likely to represent one of the roles in the KAOS relationship. Alternatively the relationship could be true when the member variable has a particular range of values which would represent a particular state of the object.

An example of such a mapping is the aspect for the relationship 'SavedFile', used in the specification of the goal 'Achieve[download File]', which is shown below:

```
1  public privileged aspect SavedFileInstance
2      extends SavedFileType
3      pertarget(execution(ManagedDownloader.new(..))) {
4
5      private pointcut setState(ManagedDownloader downloader,
6                                int state) :
7          set(int ManagedDownloader.state) &&
8          args(state) &&
9          target(downloader);
10
11     after(ManagedDownloader downloader, int state) :
12         setState(downloader, state) &&
13         if (state == ManagedDownloader.COMPLETE) {
14             initInstance(
15                 ClientInstance.aspectOf(downloader.manager),
16                 FileInstance.aspectOf(downloader));
17     }
18 }
```

This instrumentation aspect uses a `pertarget` clause so that one instance of the relationship is associated with each 'ManagedDownloader' object. The relationship does not actually hold as soon as an instance is created but when the object enters the 'COMPLETE' state, represented by the 'state' member variable. This is implemented by the pointcut named `setState` which is defined on line 5 which matches any change to the `state` variable. The advice, defined on line 11, executes when this pointcut is matched and the state variable has the value 'COMPLETE'. When the advice is called, a new instance of the relationship is created.

When `initInstance` is called to create an instance of the relationship, the instrumentation aspect has to supply the objects which represent the roles of the relationship as parameters of the method. The 'Client' entity maps onto a class in the implementation and so can be accessed using `aspectOf` with that object as a parameter. The second role is a 'File' which corresponds to the `ManagedDownloader` class as one instance of this class is responsible for a single download.

In the second type of relationship mapping, the relationship holds during the execution of a method; from the time the method is called until the time that execution completes. This execution period includes the time spent executing method calls made from within the original method. In this case, the roles in the KAOS relationship are

likely to be represented at the implementation level by parameters of the method or by member variables of the class which contains the method.

An example of this second type of mapping can be seen in the aspect for the 'RequestingFile' relationship:

```
1  public privileged aspect RequestingFileInstance
2    extends RequestingFileType
3    percflow(execution(* ManagedDownloader.doDownload(..))) {
4
5    pointcut downloadPointcut(ManagedDownloader downloader,
6                              HTTPDownloader httpDownloader):
7        execution(* ManagedDownloader.doDownload(..)) &&
8        target(downloader) && args(httpDownloader, *);
9
10   before(ManagedDownloader downloader,
11            HTTPDownloader httpDownloader) :
12            downloadPointcut(downloader, httpDownloader) {
13
14       initInstance(
15           ClientInstance.aspectOf(downloader.manager),
16           FileDescriptorInstance.aspectOf(
17               httpDownloader.getRemoteFileDesc()));
18   }
19
20   after(ManagedDownloader downloader,
21           HTTPDownloader httpDownloader) :
22               downloadPointcut(downloader, httpDownloader) {
23
24         destroyInstance();
25   }
26 }
27
```

Here the aspect is instantiated during the execution of the method `doDownload`. The relationship is instantiated when this method is called, using the pointcut `downloadPointcut`, defined on line 5, which matches the execution of the `doDownload` method. There are two pieces of advice. The first is a 'before' advice, on line 10, which creates an instance of the relationship while the second, on line 20, is an 'after' advice which destroys the relationship after the `doDownload` method completes its execution. The 'Client' roles in the relationship is associated with the `ManagedDownloader` object on which the `doDownload` method is called and so is easily obtainable. The 'FileDescriptor' role is associated with the `RemoteFileDescriptor` class corresponding to the file which is requested. The entity corresponding to this role is obtained on lines 15–16. The `HTTPDownloader` object which is the first parameter of the method call is obtained. The `ManagedDownloader` object is obtained by calling the method `getRemoteFileDesc` on this object.

Of the two types of relationship mapping, it is the second type of mapping, in which the relationship exists only during the execution of a method, which has been found to be more common. This is not what was expected as the first type of mapping,

in which relationships are mapped to particular values of attribute, is closer to the concept of the relationship at the implementation level as, relationships in UML designs are normally implemented by adding a member variable to a type.

### 4.3.3 Instrumentation Using Mapping

The monitoring framework provides an alternative approach to writing aspects directly. This approach is to provide a specialised mapping language which allows the relationship between the requirements specification and the implementation of a system to be specified explicitly. This is then used to generate instrumentation aspects similar to those described previously.

The language is written using XML so that it is easy to parse and also that it can easily interact with other tools if necessary. The implementation of the system is also described using an XML file, in this case a UML/XMI description of the implementation. This document specifies every class, attribute and method in the target system but does not contain any details of the implementation of the methods. The XMI file can be automatically generated from the source code of the target system using a UML case tool which is able to reverse engineer source code and export in XMI format.

The mapping language links KAOS entities and relationships to corresponding classes, attributes and method calls in the implementation using XPaths which refer to elements in the XMI document which describes the implementation. The syntax of the language is described by the DTD in appendix B.

The following listing shows the mapping for the 'FileDescriptor' entity from the example presented previously.

```
1  <Object name="FileDescriptor"
2    objectElement="//UML:Class[@name='RemoteFileDesc']">
3    <ObjectID object=".//UML:Attribute[@name='_filename']"/>
4    <Attribute name="name"
5      attributeObject=".//UML:Attribute[@name='_filename']"/>
6  </Object>
7
```

On line 1 the 'Object' element specifies that this is a mapping for an object (i.e. agent or entity) and that the name of the entity is 'FileDescriptor'. The 'objectElement' attribute on line 2 is an XPath which points to the element for the `RemoteFileDesc` class in the XMI document. This specifies that an instance of the entity should be created whenever a new instance of the class `RemoteFileDescriptor` is instantiated. The identifier which should be used for the entity is specified by the 'object' attribute of the 'ObjectID' element on line 3. The XPath in this attribute is relative to the node identified by the 'objectElement' attribute of the 'Object' element. It is a general principle of the language that XPaths in child elements are relative to the nodes identified by XPaths in parent elements. The 'object' attribute thus identifies the `_filename` member of the `RemoteFileDesc` class as identifying the object. Finally, lines 4 and 5 specify that the 'name' attribute is also associated with the `_filename` member of the `RemoteFileDesc` class.

This mapping will generate a single aspect which is functionally identical to the 'FileDescriptorInstance' aspect in section 4.3.2 although the generated aspect is not quite as concise. One difference is that the aspect generated by the mapping language accesses the `_filename` member directly, as it seems more natural to think of map-

ping entity attributes to member variables, while the hand written aspect uses an accessor function as is normal when writing Java code.

The mapping language is only able to handle simple, one-to-one mappings between entities and implementation classes. More complex mappings, such as required for the 'Client' entity in the Limewire example, need to be coded directly in AspectJ. This is a limitation of the mapping language rather than a fundamental limitation and further development of the language might allow it to handle such cases.

The mapping language also allows instrumentation code for relationships to be generated. The mapping for the relationship 'SavedFile' from the Limewire example is shown below:

```
1  <Relationship name="SavedFile">
2      <StateMapping
3          class="//UML:Class[@name='ManagedDownloader']"
4          attribute=".//UML:Attribute[@name='state']">
5      <State value="true"
6          valueObject="//UML:Interface[@name='Downloader']//
7                       UML:Attribute[@name='COMPLETE']"/>
8
9      <Role name="savedBy"
10           type="Client"
11           roleObject=".//UML:Attribute[@name='manager']"/>
12     <Role name="savedFile"
13           type="File"
14           roleObject="."/>
15   </StateMapping>
16 </Relationship>
```

As stated in section 4.3.2, the 'SavedFile' relationship is true while the `ManagedDownloader` class is in a particular state, represented by the `state` member variable. The mapping for the relationship is contained in the 'Relationship' element, on line 1, which states the name of the KAOS relationship which is mapped. The 'StateMapping' element on line 2 indicates that this is a mapping related to a particular state of an object. The 'class' attribute indicates that the class in question is the `ManagedDownloader` class, by referring to the node corresponding to that class in the XMI file. The attribute on which the relationship depends is the `state` attribute of that class, which is specified by 'attribute', which contains an XPath relative to the 'class' XPath. The 'StateMapping' element can contain one or more 'State' elements which indicate in what states of the object the relationship is instantiated or destroyed. In this case there is only one 'State' element, on line 5, as once an instance of the relationship 'SavedFile' is instantiated it is never destroyed. The 'value' attribute indicates that in this state, the relationship is created, as it has the value 'true'. If the 'value' attribute is 'false' then the relationship is destroyed when the object enters that state. The 'valueObject' attribute of the 'State' element, on line 6, indicates that the relationship will be instantiated when the `state` attribute has the value `COMPLETE`, which is represented by the static member variable referred to by the XPath.

The two 'Role' elements in this example indicate the values for the roles in the relationship. The 'Role' elements specify the names of the roles they refer to in the 'name' attribute and the type of the entity which fills the role in the 'type' attribute. The 'roleObject' attribute identifies the object which the entity which ful-

fils the role is associated with. In this case, the 'savedBy' role is associated with the DownloadManager object, referred to by the manager member variable of the ManagedDownloader object. The 'savedFile' role is associated with the ManagedDownloader object itself, so the XPath simply points to the same node as the 'class' attribute in the parent 'SateMapping' element.

The 'RequestingFile' relationship is instrumented by mapping the KAOS relationship to the execution of a method. This is the second type of mapping described in section 4.3.2. In the mapping language, this difference is made explicit by using a 'Transition' element, rather than a 'StateMapping' element, as can be seen in this example:

```
1  <Relationship name="RequestingFile">
2    <Transition position="around"
3        location="//UML:Class[@name='ManagedDownloader']//
4                    UML:Operation[@name='doDownload']">
5
6      <Role name="requestedBy"
7            type="Client"
8            context="class"
9            roleObject=".//UML:Attribute[@name='manager']" />
10     <Role name="requestedFile"
11           type="FileDescriptor"
12           context="method"
13           roleObject=".//UML:Parameter[@name='downloader']"
14           objectID="getRemoteFileDesc()"/>
15   </Transition>
16 </Relationship>
```

This relationship is created when execution of the method doDownload of the ManagedDownloader class begins and is destroyed when it ends. The 'position' attribute of the 'Transition' element has the value 'around' to show that the relationship is true before this execution and false afterwards. Alternative values are 'before' and 'after' which are used in conjunction with a 'value' attribute to indicate either the creation or destruction of a relationship, when the mapping is not just to the execution of a single method. The 'location' element identifies the method at which the relationship is created or destroyed. In this case it is an XPath identifying the doDownload method.

The two roles for the 'RequestingFile' relationship are mapped by the two 'Role' elements in this example. Here, the 'Role' elements have 'context' attributes which determine what the context node is for the XPath in the 'Role' element. If the 'context' attribute has the value 'method' then the node for the method at which the transition occurs, in this case the doDownload node, is used as the context node. If the 'context' attribute has the value 'class' then the node corresponding to the class to which the method belongs, in this case the ManagedDownloader node, is used as the context node. The 'roleObject' elements identify the objects associated with the entities which fill the roles, as in the previous example. In the case of the 'requestedFile' role, there is also an 'objectID' attribute on line 14. This specifies additional code which should be called to get the object associated with the entity. In this example, the generated code will find the appropriate RemoteFileDesc object by calling:

```
downloader.getRemoteFileDesc()
```

Obviously, any extra code added in the 'objectID' element has to be very simple as it can only call methods which do not have parameters.

### 4.3.4 Comparison Of Instrumentation Methods

The two approaches to instrumentation presented here complement each other. The method which is based around a mapping expressed in XML is the preferred method for simpler cases which correspond to the types of mappings which it is designed to represent. It is more concise in simpler cases, is also easier to integrate into other tools which might assist in creating the mapping and represents the mapping explicitly.

The mapping language approach is limited in that it can only express mappings of the types which are included in the design. It is not able to specify some more complex mappings. For example, a relationship may map to calls to a particular method, but only from within some other method. The mapping language does not support this type of mapping. If such mappings are necessary then they can be expressed directly in AspectJ. The structure within which these aspects are written and the generation of code from the requirements specification help to keep this code concise and well structured.

If it is found that there are common cases in which the mapping language is unable to express the mapping then it suggests areas in which the language could be expanded. Without a large amount of real world experience to draw on to identify these areas, the capability to write instrumentation aspects directly ensures that it will be possible to handle these cases regardless.

## 4.4 Monitor Display

The monitoring framework displays the results of monitoring the target system to the users of the framework. This should allow failures to be identified and information obtained which can assist in determining the severity of the problem and what action should be taken.

The display shows two types of information. First, it displays goal violations and details about those violations. It is assumed that goal violations are the most relevant information. Goals which are satisfied are obviously of less interest. Secondly, the total number of instances of a goal which have been instantiated, satisfied and failed are displayed. This gives an overall picture of how severe the number of goal failures is, as compared to the total number of goals. This is particularly useful in situations where some failure can be tolerated, which is in effect a simple form of soft goal monitoring.

Goal violations are displayed using a KAOS goal model diagram, showing goal refinements. Goals which are not being monitored are outlined in black, Goals which are being monitored and for which no violations have been detected are outlined in green unless a violation has been detected in which case it is outlined in red. A screen shot of the goal monitor display is shown in figure 4.20.

When no goal is selected, a summary of all the goal violations that have been detected is shown below the goal model. This shows the name of the goal violated, the time at which the goal was instantiated and the time at which the violation occurred.

When a goal is selected, all violations of that particular goal are displayed. This display shows the instantiation and violation times and additionally shows the values of all the parameters of the goal instance which caused the violation.

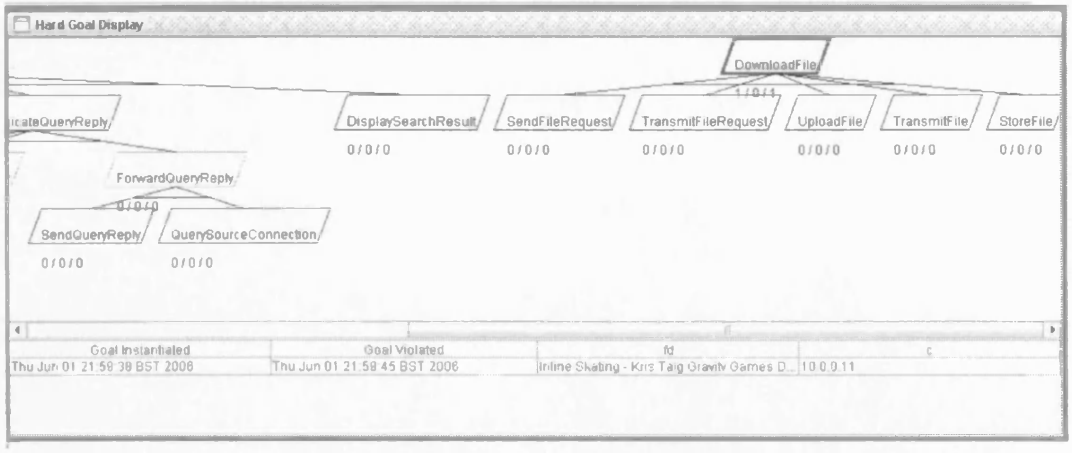



Figure 4.20: Output from monitoring the goal 'Download File'

## 4.5 Summary

This chapter has described the monitoring framework for monitoring hard goals which are specified using KAOS goal-oriented requirements specifications and temporal logic. There are various issues which need to be considered in designing a monitoring framework such as how to instrument the system, how to deal with distributed components and the performance impact of monitoring code on the target system. The choices which were made for the monitoring framework were described and justified with reference to the types of system which it is intended that the monitoring framework should be used with.

A major part of the monitoring framework is the monitor server which is responsible for evaluating whether goals are satisfied at run time. The monitor operates entirely at the requirements level and has no knowledge of the implementation of the system. The monitor uses a live instance of the KAOS object model of the system which is updated as changes occur at run time. This separates the monitor from any knowledge of the implementation of the target system.

A requirements monitoring framework has to relate events which are detected at run time, described at the implementation level, to events which can be understood in terms of the requirements specification. This is done in the monitoring framework by translating implementation level events into events which represent changes in the KAOS object model of the monitored system. Implementation level events are captured using AspectJ pointcuts and then translated into changes to the KAOS object model, in advice which is associated with those pointcuts.

The monitoring framework provides two methods for instrumenting the monitored system. The first is to write AspectJ code directly, which translates events from implementation to requirements level. The second is to write a mapping which describes how KAOS entities and relationships are related to methods, attributes and classes at the implementation level. The mapping is then used to automatically generate AspectJ code which instruments the system. The mapping approach is the preferred approach as the it is usually more concise and it makes the relationship between requirements and implementation levels explicit. Unfortunately, the mapping language is not able to handle all situations so the option to write instrumentation code directly in AspectJ is retained. This provides greater flexibility but loses some of the benefits of using the

mapping language.

# Chapter 5

# Monitoring Soft Goals

Soft goals are goals for which formal criteria for satisfaction cannot be established. An example is searching for files using a file sharing network. If the user searches for songs by a particular artist then the user wants the search results to be relevant results on servers which will enable the file to be downloaded quickly. It is not possible to formally define what constitutes a successful search in these circumstances. In practice the user may need to compromise between different soft goals. Soft goals are considered to be satisfied when sufficient evidence for their satisfaction exists that the stakeholders in those goals are convinced that they are satisfied.

Soft goals can be handled during requirements engineering by relating soft goals to hard goals which impact positively or negatively on the satisfaction of the soft goals. For example, a soft goal relating to security could be supported by goals requiring the implementation of passwords and encryption. The same hard goals might hinder soft goals requiring usability and performance. These relationships between hard goals and soft goals provide evidence, at the requirements stage, that soft goals will be satisfied or not.

While analysis during requirements engineering can help developers to implement soft goals, run-time monitoring is useful to ensure that soft goals are actually satisfied once the system is deployed. This is particularly necessary when the environment in which the system operates is subject to change which could result in soft goals failing in the same way that hard goals are subject to failure in those circumstances. Even excluding changes in the environment, it is relatively hard to ensure that a system satisfies soft goals during development so run-time monitoring is also useful in this case. It is also often the case that while a soft goal is judged to be satisfied by the stakeholders in the system, it would still be desirable to satisfy it to a greater extent if possible. Run-time monitoring of soft goals tells the developers and stakeholders in the system to what extent the system is satisfying soft goals.

Monitoring is used to provide evidence which will allow stakeholders to determine whether a soft goal has been satisfied in the deployed system and to determine if at some time in the future the system is no longer satisfying the goal.

To monitor something, it is necessary to formally define what is to be monitored. Soft goals cannot be formally defined so it is necessary to monitor something which can be formally defined. The role of monitoring is then to provide evidence which will allow stakeholders to determine whether a soft goal has been satisfied or not. This is done by formally defining metrics which are indicative of the satisfaction of soft goals. A soft goal metric is a value which will tend towards a higher (or lower) value the better a soft goal is supported by the system. These metrics are then evaluated as the system

runs and the results displayed in such a way that stakeholders and developers can use then to determine whether the associated soft goals are satisfied.

Soft goal metrics must be formally defined so that they can be monitored at run time. The monitoring framework uses the existing KAOS model of the system to assist in the specification of soft goals. This has the benefit that the amount of extra instrumentation that needs to be developed is reduced, as instrumentation already exists for those parts of the KAOS model which are already monitored. It may still be necessary to write additional instrumentation if the soft goals use parts of the KAOS model which are not otherwise monitored but it is still beneficial that the instrumentation process works in the same way as for hard goals.

There are three stages involved in the specification of soft goal metrics. *Goal instance metrics* are properties of individual goal instances which provide additional information about those goals beyond the binary success or failure results; for example, the time taken to download a file or the total amount of data that has been downloaded by a particular client. Goal instance metrics are typically related to non-functional properties which can be evaluated on an instance by instance basis. These metrics can either be built-in or user defined. The built-in metrics provide generic information which makes sense for any goal instance, such as how long a goal instance took to satisfy. User defined metrics provide additional information for particular goal types. They are defined by the developer in terms of the KAOS object model of the system they apply to. For example, for the goal 'Achieve[Download File]' from the Limewire example, the amount of data downloaded by each instance of the goal could be defined as a goal instance metric.

*Goal aggregate metrics* are defined using goal instance metrics, either built-in or user defined, by aggregating the values of metrics from many goal instances. A goal aggregate metric can aggregate the results of one goal instance metric or several. These goal instance metrics can be from a single goal type or from different goal types. Goal aggregate metrics allow the overall ability of a system to satisfy non-functional requirements to be monitored. This is important as soft goals are not concerned with individual hard goal instances but with the general behaviour of a system over a period of time. In addition, not all non-functional requirements make sense when applied to individual goal instances. For example, reliability requirements are non-functional requirements which cannot be defined meaningfully for a single goal instance. A single hard goal instance can either be satisfied or fail. Reliability requires that the system consistently satisfies instances of a particular goal type and so can only be measured by aggregating many goal instances.

The third part of specifying a monitor for a soft goal is specifying a display for the soft goal metric so that the users of the monitoring framework can evaluate the performance of the system with respect to the soft goal. A display is specified by selecting a gauge and configuring it to display a soft goal metric. Gauges include simple numeric displays of the value of a metric, bar charts of the distribution of values of a metric and line graphs which show the values of a metric over a period of time. Both goal instance metrics and goal aggregate metrics can be used as input to a gauge. Generally different types of gauges are suitable for different types of metric. Goal aggregate metrics have a single value which is modified by each update while goal instance metrics have a completely new value generated by each update.

It is not always necessary to define goal aggregate metrics. Goal instance metrics

| Name | Type | Valid | Explanation |
|---|---|---|---|
| instantiationTime | int | instantiated | Time at which goal was instantiated. |
| satisfactionTime | int | satisfied | Time at which goal was satisfied. |
| failureTime | int | failed | Time at which goal failed. |
| satisfactionPeriod | int | satisfied | Time between goal instantiation and satisfaction. |
| failurePeriod | int | failed | Time between goal instantiation and failure. |
| satisfied | boolean | instantiated | Whether the goal instance has been satisfied. |
| failed | boolean | instantiated | Whether the goal instance has failed. |

Table 5.1: Built-in goal instance metrics.

can be used directly as input to soft goal metric displays. As the displays will make use of many goal instances to generate their output, the display provides a form of aggregation in itself. There are still, however, cases where goal aggregate metrics are useful.

Section 5.1 describes how goal instance metrics are formally specified and evaluated. Section 5.2 describes how goal aggregate metrics are specified, making use of goal instance metrics. Section 5.3 describes how the results of monitoring soft goal metrics are displayed to the users of the monitoring framework.

## 5.1 Goal Instance Metrics

There are two types of goal instance metric; built-in and user defined. Built-in metrics are automatically created for all goals. They represent important properties related to the instantiation, satisfaction and failure of goals which cannot be described using the language for user-defined metrics as they are not based on the object model. User defined metrics are developed for individual goal types and are formally defined using the KAOS object model of the system to describe properties which are specific that goal type.

### 5.1.1 Built-in Metrics

The built-in goal instance metrics are shown in table 5.1. Each goal instance metric has a name which is used to refer to it in the specification of goal aggregate metrics. Each metric has a particular data type, which for built-in metrics is either integer or boolean.

A goal instance metric is undefined until the goal reaches a certain state at which the value of the metric can be calculated. If the metric is valid when the goal is instantiated then the metric always has a value which is updated when the goal is satisfied or fails. Otherwise it is not valid unless the goal instance reaches either the satisfied or failed state.

The first three built-in metrics in the table are set to the time at which the goal enters the instantiated, satisfied and failed state. The metrics 'satisfactionPeriod' and 'failurePeriod' are set to the time between when the goal was instantiated and when it entered the satisfied or failed state. Finally, the 'satisfied' and 'failed' metrics are

boolean values which are false when the goal is instantiated and are set to true when the goal enters the relevant state. These metrics are used to build goal aggregate metrics which count how many instances of a goal type are satisfied or fail.

### 5.1.2 User Defined Metrics

User defined instance metrics are based on the values of attributes of entities in the KAOS object model. A user defined goal instance metric is linked to a particular hard goal and is calculated whenever that goal enters a valid state for the metric. Part of the specification of a user defined metric is to decide what states the metric should be calculated in. These states are the usual instantiated, satisfied and failed states that apply to all goal instances.

The simplest user defined metric is simply the value of an attribute belonging to one of the entities which are referred to in the goal specification. The value of the metric is not updated whenever the value of the attribute changes, only when the goal the metric is linked to enters a state for which the metric is valid. For more complex metrics this becomes important as it stops the value of the metric from being continuously recalculated whenever an attribute changes which would involve a complex SQL query for each change.

More complex metrics can be defined using standard mathematical functions such as 'average', 'standard deviation' and 'max' to define a goal instance metric based on the values of many entity attributes. Several sets of attribute values can be used where each set contains values for a particular attribute of an entity type. All entities which match given conditions are used in the calculation of the goal instance metric.

A condition can require that only entities which satisfy a given relationship are considered in the calculation of the metric. To satisfy the condition, all role labels in the relationship which correspond to labels in the goal definition must have the same values assigned to them as the labels in the goal specification. A condition can also be based on the value of an attribute. In this case the condition is satisfied if the value of the attribute satisfies a binary comparison operator.

To try and better define goal instance metrics, some formal definitions of these metrics are now introduced. This serves to provide a more precise definition of the semantics of the language.

A soft goal instance metric is defined by a function $F(A)$ where $A$ is a set of attributes. $F$ is one of the functions 'average', 'standard deviation', 'sum', 'max', 'min' and 'count'. All of these are standard mathematical functions, except for 'count' which simply returns the size of the set of attributes $A$. The set of attribute values $A$ is defined by a set of conditions $C$ so that the value of an entity attribute $e.a$ is contained in $A$ if all the conditions are satisfied:

$$\forall e : E$$
$$e.a \in A \Leftrightarrow \forall (c : C) : c \text{ is true}$$

where $E$ is the set of all entities of a particular type. By specifying no conditions ($C$ is the empty set), all entities of a certain type are included in the calculation of the metric.

If conditions are used, they can be one of three types. The first type of condition requires that a specific entity in the goal instance, $g$, that the metric belongs to, refers to the same entity as $e$. An entity in the goal instance is referred to by a label and the notation $g[l]$ means the entity used in $g$ referred to by label $l$. This type of condition is

then formally specified as:

$$c_1(l, e) = \begin{cases} \text{true} & \text{if } g[l] = e \\ \text{false} & \text{otherwise} \end{cases}$$

This results in a single entity matching the condition and so the goal instance metric is just the value of a particular attribute. The function $F$ can be ignored here by treating it as a 'sum' function with one value.

The second type of condition requires that the entity $e$ should be a role in a particular relationship and that all other roles in the relationship which are assigned labels used in the goal $g$ should have the same entities associated with those labels.

$$c_2(r, e) = \begin{cases} \text{true} & \text{if } r(g[l_1], \ldots e, \ldots g[l_n]) \\ \text{false} & \text{otherwise} \end{cases}$$

where the relationship $r$ has labels $l_1 tol_n$ and one of those labels refers to the entity $e$.

Finally, the third type of condition requires that an attribute of the entity $e$ satisfies a certain comparison function $b$ where $b$ is one of $=, <, >, \leq, \geq, \neq$. The condition is defined as:

$$c_3(e.a, b, v) = \begin{cases} \text{true} & \text{if } b(e.a, v) \\ \text{false} & \text{otherwise} \end{cases}$$

where $v$ is some constant value.

In practice, a goal instance metric either uses a single condition of type $c_1$ or it uses a set of conditions of types $c_2$ and $c_3$.

## Syntax

Goal instance metrics are specified using an XML language. In particular, goal instance metrics are implemented by using XSLT to transform goal instance metrics written in this language to SQL queries.

Goal instance metric specifications are incorporated into a goal specification which also includes the temporal logic definition of the goal. This makes sense as the goal instance metric specification needs to make reference to the labels used in the goal specification.

A goal instance metric specification is a representation of the mathematical description above. The syntax is based on this description although it does not represent it literally but rather attempts to provide a usable representation for developers.

The syntax of the specification language for soft goal metrics is described by the following XML document type definition:

```
1  <!ELEMENT Goal (Antecedent, Consequent, Value*)>
2
3  <!ELEMENT Antecedent ANY>
4  <!ELEMENT Consequent ANY>
5
6  <!ELEMENT Value (Function|Attribute)>
7  <!ATTLIST Value label CDATA #REQUIRED
8                  type (int|float|string|boolean)
9                  trigger (instantiated|satisfied|failed)>
10
```

```
11  <!ELEMENT Function (Attribute)+>
12  <!ATTLIST Function name (avg|std|sum|min|max)>
13
14  <!ELEMENT Attribute (Conditions)*>
15  <!ATTLIST Attribute
16      label CDATA #IMPLIED
17      type CDATA #REQUIRED
18      attribute CDATA #REQUIRED
19      attributeType (int|float|string|boolean)>
20
21  <!ELEMENT Conditions (Relationship)+>
22
23  <!ELEMENT Relationship (Variable)*>
24  <!ATTLIST Relationship name CDATA #REQUIRED>
25
26  <!ELEMENT Variable EMPTY>
27  <!ATTLIST Variable label CDATA #REQUIRED
28                     role CDATA #REQUIRED
29                     type CDATA #REQUIRED>
30
31  <!ELEMENT BinaryRelation (Attribute, Constant)>
32  <!ATTLIST BinaryRelation relation (eq|ne|gt|le|gte|lte)>
33
34  <!ELEMENT Constant EMPTY>
35  <!ATTLIST Constant type (string|int|float|bool)>
```

Goal instance metrics are contained in 'Value' elements within the 'Goal' element which the metric belongs to. The 'Value' element can contain either an 'Attribute' element, in which case the value of the metric will be the value of that attribute, or a 'Function' element, in which case the value of the metric is calculated from the child elements of the 'Function' element. Each 'Attribute' element can contain conditions inside a 'Conditions' element. Conditions can either be 'Relationship' conditions or 'BinaryRelation' conditions.

An 'Attribute' element actually represents all attributes of the same name which belong to any instance of a particular entity. The conditions determine which instances are included in the calculation. An 'Attribute' element specifies a label which is used in evaluating conditions. If the label is used in the specification of the goal then only the entity instance which corresponds to that label is used in the calculation. This is a $c_1$ type condition in the formal definition above. If the label is not present in the goal specification then all entities which match the conditions contained in the 'Attribute' entity are used in the calculation.

The conditions also refer to labels. If the label is from the goal specification then the label used in the condition must have the same value as the goal instance if the attribute, specified by the enclosing 'Attribute' element, is to be included in the calculation of the metric. If the label matches the label of the enclosing 'Attribute' then the entities referred to by these labels must also match. If the label is not referred to else where then the any entity is allowed.

## Example Specification

To illustrate the previous explanation, an example is provided here. An example of a goal instance metric defined for a goal in the Limewire case study is the metric 'totalDownloaded'. This metric is evaluated whenever the goal 'Achieve[Download File]' is satisfied. The value of the metric is the sum of the sizes of all the files which have been downloaded by the client which has just completed a download, including the one which has just been downloaded.

The goal instance metric is specified by the following XML fragment:

```
<Goal name="DownloadFile" type="achieve">
  <Antecedent>
    <Relationship name="RequestingFile">
      <Variable role="requestedBy" label="c" type="Client"/>
      <Variable role="requestedFile" label="fd"
                                     type="FileDescriptor"/>
    </Relationship>
  </Antecedent>
  <Consequent>
    <And>
      <Relationship name="SavedFile">
        <Variable role="savedBy" label="c" type="Client"/>
        <Variable role="savedFile" label="f" type="File"/>
      </Relationship>
      <Equals parameterType="string">
        <Attribute label="f" type="File" attribute="name"/>
        <Attribute label="fd" type="FileDescriptor"
                              attribute="name"/>
      </Equals>
    </And>
  </Consequent>


  <Value label="totalDownloaded" type="int"
         trigger="satisfied">
    <Function name="sum">
      <Attribute label="df" type="File" attribute="size"
                 attributeType="int">
        <Conditions>
          <Relationship name="SavedFile">
              <Variable label="c" role="savedBy"
                        type="Client"/>
              <Variable label="df" role="savedFile"
                        type="File"/>
          </Relationship>
        </Conditions>
      </Attribute>
    </Function>
  </Value>
</Goal>
```

This XML specification corresponds to the following formal specification:

$$\forall g : \text{DownloadFile where } g.\text{satisfied}$$
$$\text{totalDownloaded}(g) = sum(f_1.\text{size}, f_2.\text{size}, \ldots f_n.\text{size})$$
$$\text{where } f \in \{f_1, f_2, \ldots f_n\} \Leftrightarrow \text{SavedFile}(f, g.c)$$

The whole goal instance metric specification is contained inside the element for the goal 'Achieve[DownloadFile]' along with the formal specification of that goal. The temporal logic specification of the goal 'Achieve[Download File]' is found on lines 2–21. The 'Value' element, on line 24, contains the specification of the goal instance metric. The value itself has a label which can be used to refer to it in the specification of aggregate metrics and soft goal gauges. The value also has a type, integer in this case, and a trigger condition which determines when the goal instance metric should be evaluated. In this case the trigger has the value 'satisfied' which means the metric will be evaluated for each instance of the goal 'Achieve[DownloadFile]' when the instance is satisfied. If the goal fails then the metric will never be evaluated for that instance. Triggers with the value 'failed' work in a similar way but are evaluated if a goal instance fails. The third type of trigger is the 'instantiated' trigger which is evaluated once when the goal is instantiated and again when it is satisfied or fails.

Inside the 'Value' element is a 'Function' element (line 26) which specifies a function which should be applied to the set of values returned by its child elements. In this case the function is the 'sum' function which obviously adds all the values together.

The function element contains an 'Attribute' element in line 27. The 'Attribute' element specifies an attribute, in this case the 'size' attribute of the 'File' entity. This element specifies that a set of values should be passed to the parent function corresponding to the value of the attribute for the entity instances which satisfy the conditions contained in the 'Conditions' element.

The attribute has the label 'df' (for downloaded file). Although the goal specification refers to 'File' entities, it does so using the label 'f' so in this case the label is not used in the goal specification. The metric will thus be calculated using the 'size' attribute of all 'file' entities which match the specified conditions.

In this example there is a single condition, on lines 29–36, which specifies that the relationship 'SavedFile' should exist between all the 'File' entity instances used in the calculation of the metric and the 'Client' agent referred to by the label 'c'. The role 'downloadedFile' uses the same label as the enclosing 'Attribute' indicating that the relationship should be true for any entity which is used in the calculation. The label 'c' is used in the specification of the goal to refer to the 'Client' entity which is performing the download. This means that only instances of 'SavedFile' in which the role 'downloadedBy' is filled by this 'Client' entity should be tested against the attribute.

## Evaluation of Goal Instance Metrics

Goal instance metrics are evaluated as the monitored system runs. Whenever a goal is instantiated, satisfied or fails, all the relevant goal instance metrics which are associated with that goal are evaluated by querying the requirements object model, which is stored in a database, using SQL. The query used to evaluate a goal instance metric is generated automatically from the specification of that goal instance metric. This generation is done using XSLT to transform from the XML specification to an SQL query. The XSLT transform which carries out this process is included in appendix C.

As an example, the SQL query which is generated for the goal instance metric 'totalDownloaded' is:

```
1  <Value label="totalDownloaded" type="int"
2         trigger="satisfied" goal="DownloadFile">
3      <BoundLabel>c</BoundLabel>
4      <ValueQuery>
5  SELECT SUM(value) AS val FROM attribute, entity
6      WHERE name='size' AND entity.type='File'
7      AND entity.id = attribute.entity_id
8      AND new!='TRUE'
9      AND entity_id IN
10     (SELECT entity_id FROM role_entity, relationship
11         WHERE relationship.type='SavedFile'
12         AND relationship.id = role_entity.relationship_id
13         AND role='savedFile'
14
15         AND relationship.id IN
16         (SELECT relationship_id FROM role_entity
17             WHERE role='savedBy'
18             AND entity_id=?
19         )
20     )
21     </ValueQuery>
22 </Value>
```

The query is contained inside an XML document which contains additional information about the query. A query is generated for each goal instance metric. Each query is contained in a 'Value' element which specifies the label of the goal instance metric, the type of the metric (i.e. integer, float etc.), when the metric should be evaluated and the goal which the goal instance metric belongs to.

The 'Value' element contains two types of element. 'BoundLabel' elements indicate which labels in the specification of the goal instance metric have the same values as labels in the goal specification itself. These labels will already be bound to a value when the goal instance metric is evaluated. Unbound labels are labels which are used in the specification of the goal instance metric but not in the goal specification. These labels may potentially have many possible values when the goal instance metric is evaluated and the value of the metric is found by aggregating the values calculated for each possible value of the unbound labels.

Bound label elements are used at run time when an SQL query is evaluated. The values bound to these labels for a particular goal instance are inserted into the query to replace the unspecified values (the '?' symbols in the query). The 'BoundLabel' elements occur in the order that these values should be inserted into the SQL query. When a goal instance metric is evaluated, the monitor goes through the unspecified values in order and assigns the values bound to these labels in the order they appear.

The actual SQL query which calculates the value of the goal instance metric is contained inside the 'ValueQuery' element and the query itself starts on line 5. This is a fairly complex query but the relationship to the specification of the goal instance metric is reasonably simple. Line 5 calculates the sum of the values returned by the query and uses the name 'val' to refer to the sum. The query uses the 'attribute' and

'entity' tables to evaluate the query. This part of the query roughly corresponds to the 'Function' element of the goal instance metric specification.

Line 6 specifies that the name of the entity attribute which should be used in the calculation which in this case is the 'size' attribute of the 'File' entity. Line 7 joins the entity and attribute tables so that the name of the entity type can be retrieved from the entity table.

Line 8 specifies that 'new' attributes should not be included in the calculation. If an attribute has the value 'new' set to true then the updated value occurred after the time that the goal instance metric refers to and so it should not be used in the calculation. When attribute values are processed the 'new' attribute is set to false and the old value deleted so that the updated value becomes eligible for inclusion in goal instance metrics calculations from that time onwards. Lines 6 to 8 correspond to the 'Attribute' element of the goal instance metric specification.

The rest of the query corresponds to the contents of the 'Conditions' element of the goal instance metric specification, which in this case is a single relationship. Lines 9 and 10 restrict the entities which are used in the calculation of the metric to those which satisfy additional conditions contained in the sub-query. Lines 11 and 12 corresponds to the 'Relationship' element from the specification and ensures that the entity is tested against instances of the 'SavedBy' relationship. Line 13 ensures that the entity being tested is tested against the role 'savedFile', as the label from the 'Attribute' element and the label of that role are the same. Finally, lines 15 to 19 correspond to the 'savedBy' role in the 'SavedFile' relationship. The label on this role 'c' is a bound label which matches a label in the specification of the goal 'DownloadFile'. The name of the role is identified on line 17 and on line 18 the value bound to the label needs to be inserted into the query. This is done at run time for each goal instance individually as the value of the bound label can vary from instance to instance.

## 5.2 Goal Aggregate Metrics

Goal aggregate metrics are soft goal metrics which are calculated by combining goal instance metrics from many goal instances. A Goal aggregate metric can be defined over all instances of a particular goal or instances of more than one goal type. Goal aggregate metrics make use of an aggregation function to calculate values from a number of goal instances. Obvious examples are functions such as 'average' and 'standard deviation'. Some additional functions which are specific to goal monitoring are provided such as 'Count', 'Interval' and 'Rate', which measures the frequency at which events occur. The values returned by these aggregation functions can be combined using standard arithmetic operators (e.g. +, -, *, /) to calculate the overall value of the goal aggregate metric.

### 5.2.1 Formal Definition of Aggregate Functions

An aggregate function is a function defined for a set of goal instances which is time dependent. Consider a set of goal instances, $G$, with instances $\{g_0, g_1, \ldots\}$, all of which have a goal instance metric $v$ defined for them. The set $G_{t_0,t} = \{g_m, \ldots g_n\}$ is a subset of $G$ where $g_m$ is the first goal instance after $t_0$ for which $v$ was set and $g_n$ is the last goal instance for which $v$ was set before time $t$. When an aggregate metric is evaluated, the time $t$ is the current time and the time $t_0$ is either a fixed time in the past, in which the time period over which the metric is evaluated is variable, or an earlier time defined relative to the current time, in which case the time period is fixed. An

aggregate function is then a function $F(G_{t_0,t}, v)$.

The monitoring framework provides a number of aggregate functions. The 'Avg' function is defined as:

$$\text{Avg}(G_{t_0,t}, v) = \begin{cases} 0 & \text{if } G_{t_0,t} = \emptyset \\ \frac{g_m.v + \ldots + g_n.v}{n-m+1} & \text{otherwise} \end{cases}$$

so that it calculates the average of the values of the goal instance metrics for all the goal instances considered. The functions max, min and sum are similarly defined in the normal mathematical way.

The 'Count' function is equal to the number of goal instances for which a boolean valued goal instance metric is true. Its main purpose is to count the number of instances of a goal which have been satisfied or have failed, using the 'satisfied' and 'failed' built-in goal instance metrics. The aggregate function is defined as:

$$\text{Count}(G_{t_0,t}, v) = |S| \text{ where } S \subseteq G \text{ and } g_i \in S \Leftrightarrow g_i.v$$

The 'Rate' function is used to calculate the rate at which some event is occurring. It can be used, for example, to find the rate at which instance of a goal are being satisfied. It is defined as:

$$\text{Rate}(G_{t_0,t}, v) = \frac{\text{Count}(G_{t_0,t}, v)}{t - t_0}$$

The 'Interval' function calculates the difference between the most recent goal instance metric to be defined and the previous one. It is useful in combination with built-in goal instance metrics such as 'satisfactionTime' and 'failureTime' to calculate the intervals between goal satisfaction or failure. No start time has to be set for this function as it only uses the time between the two most recent goal instances in its calculation. The 'Interval' function is defined as:

$$\text{Interval}(G_{0,t}, v) = \begin{cases} 0 & \text{if } |G_{0,t}| \leq 1 \\ g_n.v - g_{n-1}.v & \text{otherwise} \end{cases}$$

### 5.2.2 Goal Aggregate Metric Syntax

The soft goal specifications are represented using an XML language which can be understood by the monitoring system. The syntax of this language is defined by the document type definition for the language:

```
1  <!ELEMENT Count (Value)*>
2
3  <!ELEMENT Rate (Value)*>
4  <!ATTLIST Rate samplePeriod CDATA #IMPLIED>
5
6  <!ELEMENT Interval (Value)*>
7
8  <!ELEMENT Avg (Value|Avg|Count|Interval|Rate|
9                 Sum|Ratio|Diff|Product)*>
10
11 <!ELEMENT Sum (Value|Avg|Count|Interval|Rate|
12                Sum|Ratio|Diff|Product)*>
```

```
13
14  <!ELEMENT Product (Value|Avg|Count|Interval|Rate|
15                     Sum|Ratio|Diff|Product)*>
16
17  <!ELEMENT Ratio
18      ((Avg|Count|Interval|Rate|Sum|Ratio|Diff|Product),
19       (Avg|Count|Interval|Rate|Sum|Ratio|Diff|Product))>
20
21  <!ELEMENT Diff
22      ((Avg|Count|Interval|Rate|Sum|Ratio|Diff|Product),
23       (Avg|Count|Interval|Rate|Sum|Ratio|Diff|Product))>
24
25  <!ELEMENT Value EMPTY>
26  <!ATTLIST Value name CDATA #REQUIRED
27                  goal CDATA #REQUIRED>
```

There are three types of elements which are used to described a goal aggregate metrics. Aggregate functions are specified by the elements 'Count', 'Rate', 'Interval', 'Avg', 'Sum' and 'Product'. All of these elements can contain 'Value' elements which specify goal instance metrics which should be used in the calculation of the aggregate functions. The 'Value' element specifies the name of a goal and the name of a goal instance metric. The aggregate function is calculated over all instances of the goal. Several 'Value' elements can be contained inside a single aggregate function element if desired, in which case the function is calculated for all instances of the goals specified by all the 'Value' elements.

The results of the aggregate functions can be combined using the 'Diff' and 'Ration' functions to calculate the ratio or difference of values. The 'Avg', 'Sum' and 'Product' functions can also be used in this way as well as their use as aggregate functions for goal instance metrics.

### 5.2.3 Examples of Goal Aggregate Metrics

There are a number of examples of soft goal aggregate metrics which can be defined for the Limewire example. The use of the 'Interval' operator is demonstrated by the soft goal 'Min[Routing Failures]' from the Limewire example. The informal definition of this goal is:

> The number of times that a client fails to correctly route a query reply message to the source of the corresponding query should be minimised.

This goal is related to the hard goal 'Maintain[Query Source Connection]' which requires that when a client receives a reply to a previously routed query the client should still be connected to the peer which forwarded the query.

The goal 'Maintain[Query Source Connection]' is actually rather idealised as it is perfectly legitimate for a user to disconnect from a peer between forwarding a query and query replies coming back. In such cases, any messages which should have been routed to that peer are simply dropped. Similarly, the application tolerates failures in the network which cause a peer to become disconnected by dropping messages for that peer and connecting to a new peer. These actions are made possible by the robustness of the Gnutella protocol in which none of the messages are critical to the operation of the system. Dropping messages only affects the overall quality of service delivered by

the system. Rather than specifying non-idealised hard goals, it is more useful from the perspective of monitoring to specify soft goals related to these idealised goals which can be monitored.

A routing failure occurs whenever the goal 'Maintain[Query Source Connection] fails. There are several metrics which could be used to evaluate the soft goal 'Min[Routing Failures]'. These include a simple count of the number of failures and the rate at which failures occur. The measure which is used here is the average interval between failures. The soft goal is specified by the following XML fragment:

```
1  <Avg>
2    <Interval>
3      <Value goal="QuerySourceConnection" name="failureTime"/>
4    </Interval>
5  </Avg>
```

The value of the 'Interval' element is the time between the two most recent failures of the goal 'Maintain[Query Source Connection]'. The 'Avg' element then results in the average interval being calculated. The metric represented by the 'Interval' element is described by the following specification:

$$\text{Interval}(G_{0,t}, \text{failureTime}) \text{ where } g \in G \text{ iff } g \in \text{QuerySourceConnection}$$

An example of the use of the rate operator is found in the specification of a metric for the soft goal 'Min[Search Communication Overhead]' in the Limewire example. This goal is informally defined as:

> The bandwidth used to communicate the searches of other users should be minimised.

In this case there are three hard goals which impact negatively on this soft goal. These are:

- Achieve[Forward Query]
- Achieve[Respond To Query]
- Achieve[Forward Query Reply]

All these goals involve the communication of queries from other users and the responses to those queries. While these messages are essential to the operation of the Gnutella network, from the point of view of an individual user of the network they add no value so should be minimised. The metric for this soft goal is calculated by combining the impact of these three hard goals.

For the purposes of this example, it is decided to measure the bandwidth in terms of the number of messages which are routed by the client. Another approach would be to define goal instance metrics which measure the actual size of each message (along with appropriate instrumentation to discover the size of each message). As the messages should all be approximately the same size, the results should give more or less similar results so the first, easier approach is adopted.

Each of the three goals which result in communication overhead cause a single message to be sent if they are satisfied. The communication overhead is specified in terms of messages sent per minute using the rate operator.

This soft goal metric is specified using the following XML fragment:

```
<Rate samplePeriod="60000">
    <Value goal="ForwardQuery" name="satisfied"/>
    <Value goal="RespondToQuery" name="satisfied"/>
    <Value goal="ForwardQueryReply" name="satisfied"/>
</Rate>
```

This specifies that the value of the soft goal metric should be the rate at which instances of any of the named goals are satisfied. The three 'Value' elements specify what the three goals are at that the metric should be evaluated when these goals are satisfied. The sample period attribute of the 'Rate' element specifies that the rate should be calculated by considering goals which were satisfied in the last minute.

The formal definition of this soft goal metric, which is represented by the above XML fragment is:

**Min[Search Communication Overhead]**
Rate($G_{t-60000,t}$, satisfactionTime) where $g \in G$ iff
$g \in$ ForwardQuery $\cup$ RespondToQuery $\cup$ ForwardQueryReply

## 5.3 Display of Soft Goal Metrics

The final stage in run-time monitoring is to provide feedback to the user about the execution of the system. The monitoring framework described in this thesis provides visual feedback to the user. This takes the form of a number of *gauges*, each of which displays the value of one or more soft goal metrics. Three gauges have been developed as part of the framework, which provide the most obvious types of feedback for the soft goals used in the Limewire and NGDS systems. The framework also allows additional gauges to be developed easily using a plug-in architecture. Any gauge class which is installed in the plug-in folder is automatically loaded and can be referred to by name in the specification of the display for a particular soft goal. The gauges which have been developed are the distribution gauge, the history gauge and the min-max gauge.

A distribution gauge shows a distribution of values using a bar chart of the frequency with which a soft goal metric reports different ranges of values. The range of possible values is split into a number of divisions with each division covering an equal range. The y-axis of the chart shows the relative number of values which fall into each division as a percentage of the total number of values. Distribution gauges are a good choices for showing the range and distribution of values of a particular goal instance metric.

A history gauge displays the value of a goal instance metric over time. This gauge allows several metrics to be displayed using the same gauge with each metric being represented by a different line. This type of gauge is particularly suitable for displaying the value of a goal aggregate metric as it changes over time.

A min-max gauge displays the current value of one or more soft goal metrics as bars, along with indicators of the minimum and maximum values which each metric has reached in the past. It is suitable for showing the current value of a goal aggregate metric, without historical information (other than the min and max values). This may be a better choice in some circumstances, particularly when a large number of different metrics need to be compared with each other.

### 5.3.1 Specification of Displays

The display for a soft goal metric is specified manually as part of the same document in which goal aggregate metrics are defined. The following DTD describes the additional elements which are used to specify a display:

```
1  <!ELEMENT Project (Display)*>
2  <!ATTLIST Project name CDATA #REQUIRED>
3
4  <!ELEMENT Display(Value|Avg|Count|Interval|Rate|
5                    Sum|Ratio|Diff|Product|GaugeParameter)*>
6  <!ATTLIST Display title CDATA #REQUIRED
7                    class CDATA #REQUIRED>
8
9  <!ELEMENT GaugeParameter EMPTY>
10 <!ATTLIST GaugeParameter name CDATA #REQUIRED
11                          value CDATA #REQUIRED>
12
```

The 'Project' element is the root element of the document. It gives a name to the monitoring project and contains all the display specifications which are specified using 'Display' elements. The 'Display' element specifies a title for the display and the name of the class which implement it. The 'class' attribute of the 'Display' element specifies the name of the Java class which implements the gauge which should be used to display the monitor result. The named class is loaded at run time using a Java class loader which allows new gauge types to be added to the framework easily. The 'Display' element can contain 'GaugeParameter' elements which allow values to be passed to the gauge class This allows the developer to configure the gauge so that is displays the correct information for the metric which is displayed. For example, the gauge parameters could modify the labels on the axes or the range of values along an axis. The allowable parameters are specific to each gauge type. Gauge parameters should have default values specified inside the display class that will be used if the parameter is omitted.

Each 'Display' element also contains one or more specifications of goal aggregate metrics. A goal instance metric can also be displayed directly by placing a 'Value' element directly inside the 'Display' element. Although aggregate metrics and goal instance metrics can be treated identically by gauges, it is important to remember that they actually represent different thing conceptually and so the types of displays that are suitable may be different. Each time a goal instance metric passes a value to a gauge, the value is a completely new value which is unrelated to previous values of the metric. An appropriate type of gauge for goal instance metrics is a distribution gauge which shows the distribution of results as a bar chart. The values passed by an aggregate metric are updates to the previous value of the metric. An example of a suitable gauge for this type of metric is a history gauge which shows the history of the gauge over time.

An example of a display specification from Limewire is the following:

```
1  <Project name="limewire">
2    <Display title="Minimise Download Time"
3             class="DistributionGauge">
4      <GaugeParameter name="xLabel"
5                      value="Download Time/min"/>
```

| Parameter | Type | Default | Explanation |
|---|---|---|---|
| divisions | long | 10 | Number of divisions to use |
| interval | long | 1000 | Size of each division |
| labelFreq | long | 2 | Frequency at which divisions should be labelled |
| xLabel | String | "time / s" | Label on the x-axis |
| yLabel | String | "y-axis" | Label on the y-axis |
| xScaleFactor | long | 1000 | The value of x-axis labels are divided by this amount |
| rescale | boolean | true | True if the size of the divisions should be recalculated if a value is out of range |

Table 5.2: Possible parameters for the distribution gauge.

```
6       <GaugeParameter name="interval" value="60000"/>
7       <GaugeParameter name="xScaleFactor" value="60000"/>
8       <Value name="satisfactionPeriod" goal="DownloadFile"/>
9     </Display>
10  </Project>
```

This specification tells the monitoring system to display a gauge showing the distribution of download times for files which are successfully downloaded. The 'Display' element is used to describe how the soft goal metrics of its child elements should be displayed. The 'title' attribute of this element provides a title for the display. The 'class' attribute specifies which gauge class should be used to display the metrics. In this example, the class selected is the 'DistributionGauge' class. The gauge displays the distribution of download times for different file downloads. It normally takes a few minutes to download a file so the display will show what percentage of files are downloaded in one minute, two minutes and so on up to ten minutes.

The 'Display' element contains a number of 'GaugeParameter' elements as well as elements which represent soft goal metrics. The 'GaugeParameter' elements are used to set up the gauge selected by the 'Display' element. The allowable parameters depend on which gauge is used as each gauge has its own set of parameters which it understands which allows new gauges to have whatever parameters the developer decides are necessary. The parameters which the distribution gauge understands are shown in table 5.2. Each parameter is referred to by its name, which is specified in the 'name' attribute of the 'GaugeParameter' element. The value of the parameter is specified by the 'value' attribute. Each gauge parameter has an allowable range of values, indicated by the type entry in the above table. If a parameter is not specified for a particular display, the value of that parameter is set to the default value shown in the table. In the example, only three parameters are set; the rest will be automatically set to the default values. The first of these three parameters, the 'xLabel' parameter, sets the label on the x-axis of the gauge to indicate that it shows the download time. The 'interval' parameter sets each interval on the x-axis to a size of one minute (or sixty thousand milliseconds). The 'xScaleFactor' parameter tells the gauge to scale the labels on the x-axis so that the labels will show minutes rather than milliseconds.

The soft goal metric to be monitored by the display is specified by the 'Value' element in this example. This element tells indicates that the gauge should take the

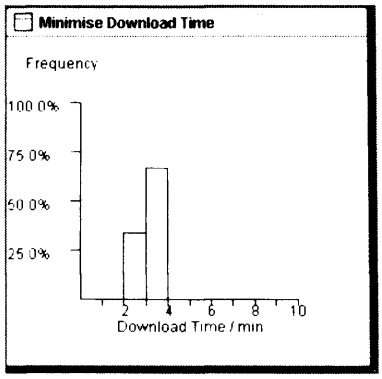



Figure 5.1: An example of a distribution gauge.

'satisfactionPeriod' goal instance metric of the goal 'DownloadFile' as input. This will result in the gauge displaying the distribution of the time taken to download files as this corresponds to the time taken to satisfy the goal 'DownloadFile'. The typical output from this gauge is shown in figure 5.1.

An example of the use of a history gauge is shown in the following example:

```
<Display title="Routing Overhead" class="HistoryGauge">
    <GaugeParameter name="yLabel" value="Messages / min"/>
    <GaugeParameter name="yMax" value="30"/>
    <GaugeParameter name="yInterval" value="5"/>
    <GaugeParameter name="period" value="180000"/>
    <GaugeParameter name="xInterval" value="60000"/>

    <GaugeParameter name="label1" value="all messages"/>
    <Rate samplePeriod="60000">
        <Value goal="ForwardQuery" name="satisfied"/>
        <Value goal="RespondToQuery" name="satisfied"/>
        <Value goal="ForwardQueryReply" name="satisfied"/>
    </Rate>

    <GaugeParameter name="label2" value="forward query"/>
    <Rate samplePeriod="60000">
        <Value goal="ForwardQuery" name="satisfied"/>
    </Rate>

    <GaugeParameter name="label3" value="respond to query"/>
    <Rate samplePeriod="60000">
        <Value goal="RespondToQuery" name="satisfied"/>
    </Rate>

    <GaugeParameter name="label4"
                    value="forward query reply"/>
    <Rate samplePeriod="60000">
        <Value goal="ForwardQueryReply" name="satisfied"/>
    </Rate>
</Display>
```

| Parameter | Type | Default | Explanation |
|---|---|---|---|
| period | long | 60000 | Period of time to display history for in ms |
| xLabel | String | "time / s" | Label on the x-axis |
| yLabel | String | "y-axis" | Label on the y-axis |
| xInterval | long | 20000 | Interval between ticks on x-axis |
| yInterval | float | 1.0 | Interval between ticks on y-axis |
| yMin | float | 0.0 | Minimum value on y-axis |
| yMax | float | 5.0 | Maximum value on y-axis |
| rescale | boolean | true | True if y-axis should be rescaled if a value is out of range |
| label[n] | String | "" | Label for individual line |

Table 5.3: Possible parameters for the history gauge.

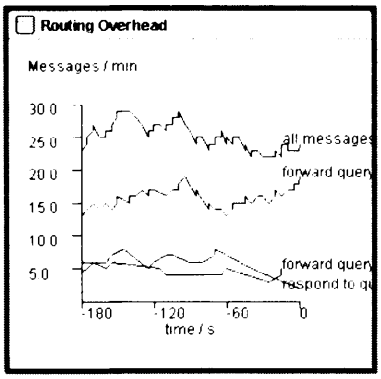



Figure 5.2: An example of a history gauge.

The example specification shows the number of search messages passed by a Limewire client over time. This is useful to know as passing large number of search messages can impair other aspects of network performance, particularly the speed of file downloads. The display should show four different lines, one representing the overall rate at which messages are being passed by the client and the others representing the rate of different types of messages.

The parameters which a history gauge understands are shown in table 5.3. The most important parameter is the 'period' parameter which determines how far into the past the gauge should display values. If the 'rescale' parameter is set to true then when a metric exceeds the maximum value on the y-axis, the axis will be modified to fit the new, higher, value. The 'label[n]' parameters are used to specify labels for the lines for individual metrics.

The goal aggregate metrics to be displayed by the history gauge are specified inside the 'Display' element. There are four goal aggregate metrics which calculate the rate at which different goals relating to search message passing are satisfied. Typical output for this display is shown in figure 5.2.

An example of the use of a min-max gauge is the following specification:

```
1  <Display title="Routing Overhead Min-Max"
2           class="MinMaxGauge">
```

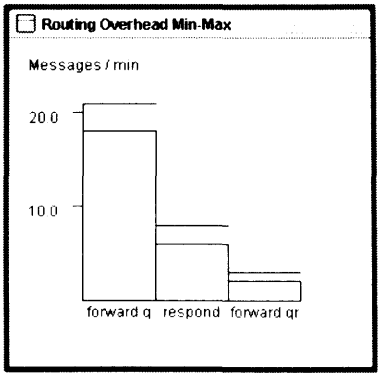



Figure 5.3: An example of a min-max gauge.

```
3       <GaugeParameter name="yInterval" value="10"/>
4       <GaugeParameter name="yLabel" value="Messages / min"/>
5
6       <GaugeParameter name="label1" value="forward q"/>
7       <Rate samplePeriod="60000">
8           <Value goal="ForwardQuery" name="satisfied"/>
9       </Rate>
10      <GaugeParameter name="label2" value="respond"/>
11      <Rate samplePeriod="60000">
12          <Value goal="RespondToQuery" name="satisfied"/>
13      </Rate>
14      <GaugeParameter name="label3" value="forward qr"/>
15      <Rate samplePeriod="60000">
16          <Value goal="ForwardQueryReply" name="satisfied"/>
17      </Rate>
18  </Display>
```

The example display specification will created a min-max gauge which displays the rate at which different types of Gnutella messages are being passed by a client as in the previous example of the history gauge. The current value of each metric is displayed in a bar, along with an indication of the maximum value it has achieved. As the rate starts off at zero, the minimum value is not relevant in this example as it will always be zero. An example screen shot of this display is shown in figure 5.3.

The allowable parameters for a min-max gauge are shown in table 5.4. These parameters work in a similar way to the parameters for the previous two gauge types. The parameters allow the y-axis label, maximum y-axis value and space between tick marks to be set. Additionally, the y-axis can be set to be rescaled if the largest value of one of the bars is greater than the initial maximum value by setting the 'rescale' parameter.

### 5.3.2 Development of Additional Gauge Types

The monitoring framework allows developers to easily add new gauge types to the framework. This allows gauges to be developed which are specific to particular domains so that information from run-time monitoring can be effectively communicated.

A gauge class must implement the `Gauge` interface:

| Parameter | Type | Default | Explanation |
|---|---|---|---|
| yLabel | String | "y-axis" | Label on the y-axis |
| yMax | float | 5.0 | Maximum value on y-axis |
| yInterval | float | 1.0 | Interval between ticks on y-axis |
| rescale | boolean | true | True if y-axis should be rescaled if a value is out of range |

Table 5.4: Possible parameters for the min-max gauge.

```
1  public interface Gauge {
2      public void init(SoftGoalMonitor[] monitors,
3                       Map paramMap);
4      public JPanel getGaugePanel();
5  }
```

The `init` method is called when the gauge is created. The `monitors` parameter tells the gauge what metrics to display and the `paramMap` parameter tells it what the parameters passed by the 'GaugeParameter' elements are. The `getGaugePanel` method should return an object which is responsible for displaying the gauge.

Normally gauge classes will extend the `AbstractGauge` class which handles the creation of the display area for the gauge and attaches a listener to the monitor for the soft goal metric so that the gauge is informed of value changes. The interface of the `AbstractGauge` class is:

```
1  abstract public class AbstractGauge implements Gauge {
2      public abstract void drawGauge(int w, int h,
3                                     Graphics2D g);
4      public abstract void updateValues(Number value,
5                                        int monitorIndex,
6                                        long time);
7
8      public void init(SoftGoalMonitor[] monitors,
9                       Map paramMap) { ...  }
10     public JPanel getGaugePanel() { ....  }
11     protected String getStringParam(String key,
12                                     String def) { ...  }
13     protected boolean getBooleanParam(String key,
14                                       boolean def) { ...  }
15     protected int getIntegerParam(String key,
16                                   int def) { ...  }
17     protected long getLongParam(String key,
18                                 long def) { ...  }
19     protected float getFloatParam(String key,
20                                   float def) { ...  }
21 }
```

The `AbstractGauge` class implements the `init` and `getGaugePanel` methods and adds some methods which allows the gauge class to access the gauge parameters in a convenient manner. This class also adds to abstract methods which need to be implemented by the gauge implementation. The `drawGauge` method

is called whenever the gauge needs to be drawn and should render the gauge. This method is passed a `Graphics2D` object which allows the gauge window to be rendered to as well as the width and height of the gauge. The second abstract method is the `updateValues` method which is called whenever one of the metrics being monitored changes. The parameters tell the gauge the new value of the metric, which metric has changed and the time the change occurred. This method should store the new state of the gauge so that the change can be rendered by the `drawGauge` method.

## 5.4 Summary

This chapter has discussed the specification of monitorable metrics which are indicative of the satisfaction of soft goals. These metrics are monitored at run time and the results displayed to users of the monitoring framework so that it can be determined whether these goals are satisfied by a running system in a deployment environment.

The specification of soft goal metrics involves two types of metrics: goal instance metrics and goal aggregate metrics. The former are evaluated for individual goal instances while the latter are evaluated over sets of goal instances. This approach allows the specification of complex soft goal metrics in a structured manner and provides a great deal of flexibility in the types of metrics which can be specified.

The formal specification of soft goal metrics makes use of the KAOS object and goal models. Goal instance metrics make use of the KAOS object model while goal aggregate metrics make use of the KAOS goal model and the goal instance metrics which have been defined. This means that no additional instrumentation is required as long as soft goal metrics make use of entities, relationships and goals which are already monitored. Even if the specifications use parts of the KAOS models which are not already monitored, the instrumentation process is the same as for hard goals so existing processes can be used.

Goal instance metrics are implemented as SQL queries. This has made them relatively easy to implement using XSLT to generate the query from the specification. This was particularly valuable during development of the specification language for goal instance metrics as it made it possible to rapidly make changes to the language and implement those changes in SQL. This also means that it should be relatively easy to change or extend the specification language if it should become necessary. The downside of this approach is that user defined goal instance metrics are only usable if an SQL database is used to store the KAOS object model during monitoring. While there is no reason why an implementation of the goal instance metric specification language could not be written for a KAOS object model implemented as Java objects, it would have been more time consuming and harder to modify as the language was developed. Now that a reasonably stable version of the language has been developed and implemented for the object model stored in a database, an implementation for an object model implemented in Java is more feasible. Built in goal instance metrics and goal aggregate metrics which use them are still usable with either implementation of the KAOS object model.

# Chapter 6

# NGDS Case Study

The monitoring framework which has been described in the previous chapter was evaluated using a case study to determine whether the framework achieves the objectives set out in chapter 1. This was done by implementing monitoring for a workforce scheduling system.

The system used for this case study is a prototype workforce scheduling system being developed by BTexact called the Next Generation Dynamic Scheduler(NGDS). The purpose of the system is to allocate BT's field technicians to jobs, taking account of the varying lengths of jobs and the travel time between jobs. A job may need to be split over a break if it is too long. Some jobs may also need to be performed in parallel with a related job at another location.

The system is an improved version of the existing dynamic scheduling system which is already deployed by BT. This system runs at the start of the day to create a schedule for the day and at regular intervals throughout the day to update the schedule as new jobs are added or existing jobs slip back in the schedule. The NGDS system improves on the existing system by providing much greater flexibility in the algorithm used to generate the schedule, allowing it to be customised to the specific environment in which it operates and to be easily modified if that environment changes.

This system was chosen for the case study because it is a real system which is relatively large and complex. The soft goals which were monitored were based on the needs of the developers of the system to understand the operation of the scheduling system as well as the customer's need to get an overall picture of the operation of the scheduling system. The soft goals which are monitored are thus related to real requirements.

The NGDS system uses a series of algorithms to construct a schedule containing tasks which are either jobs or breaks. Initially, tasks are inserted into the schedule one at a time. For each task, the scheduler tries to find the best available position for the task so that tasks are not late or are as close as possible to their desired completion time. The scheduling algorithm must also obey functional requirements such as allowing sufficient travel time between jobs at different locations and correctly scheduling parallel jobs. Jobs are typically inserted into the schedule in three stages. First jobs which need to be performed at multiple locations by multiple technicians in parallel are inserted into the schedule. Secondly, jobs which will take a long time and need to be split over breaks are inserted. Finally, the remaining jobs are inserted. The schedule is constructed in this order so that the most difficult jobs to schedule are inserted first, followed by simpler jobs. Having constructed a schedule, the system then tries to optimise it using one of a choice of local search algorithm which rearrange items in the schedule

to try and improve the quality of the schedule. For example, a good schedule should minimise the time technicians spend travelling by assigning jobs which are closely spaced geographically to the same technician where possible. The schedule should also try and minimise waiting time for customers and take account of the priority given to different types of jobs. Local search algorithms have a number of parameters which can be modified to provide fine tuning of the algorithm.

Because the schedule runs at regular intervals throughout the day, there is a time limit on how long the scheduler can run. The scheduler is called every ten minutes so the time limit is a few minutes.

The goals which were considered most interesting to monitor by the developers of the system were soft goals related to the quality of the search results produced. The functional aspects of the system are fairly straightforward as the main functional goal is to construct a schedule which satisfies certain conditions such as allowing sufficient travel time between jobs. The main area of interest was to determine what effect each search stage has on the quality of the schedule using various measures.

There are three types of metric which are of interest. The first are metrics associated with the assignment of tasks. The second are metrics associated with the technicians who carry out the tasks. The third type of metric is involves the movement of tasks which happens when a schedule is optimised and the number and types of moves that an algorithm performs.

For each task there is a quality of service value which is already defined in the system. This takes into account lateness in scheduling the task as well as complete failure to schedule the task to any technician, taking into account the priority of the task. There is also a value for the travel time taken for the technician to reach the location at which the task needs to be carried out.

Technician metrics include the number of tasks assigned to an individual technician, the amount of time technicians spend travelling and the number of technicians which have not been assigned any tasks. Generally, the more tasks assigned to each technician, the more efficiently the technicians are being used. The amount of time each technician spends travelling indicates how well the algorithm is routing technicians to jobs which are closely spaced geographically. Technicians which have not been assigned any jobs suggests that the algorithm may not be assigning tasks efficiently unless there are very few jobs to assign.

The task and technician metrics are goal instance metrics which are associated with goals which require that each search stage completes. These metrics can be combined over many executions of the scheduler to build up a picture of the performance of each search stage.

Once a schedule has been constructed, it is optimised by rearranging tasks so that the quality of service increases. NGDS uses two types of moves to do this. Insertion moves insert a task before or after another task in the schedule. Swap moves exchange the positions of two tasks in the schedule. It is useful to measure the number of each type of move performed in each search stage and their relative proportions.

Generally, there are likely to be several assumptions about the environment which are made in selecting an algorithm and configuring its parameters. Assumptions include the expected number of technicians and tasks, the relative proportions of high and low priority tasks and the relative numbers of parallel and long jobs. As the system is designed with a lot of flexibility in the algorithms, some of these assumptions are made

when the system is configured rather than when the system is implemented. These assumptions are also soft so, for example, the expected proportions of high and low priority tasks is a vague amount rather than a precise figure.

The rest of this chapter proceeds through the process for monitoring a system using the monitoring framework. Section 6.1 outlines the objectives of the case study. Section 6.2 describes the formal definition of hard goals for the NGDS system. Section 6.3 formally defines soft goal metrics corresponding to the soft goals described above. Section 6.4 shows how the NGDS system is instrumented using AspectJ. Section 6.5 presents the results of the case study.

## 6.1 Objectives

This case study aims to validate the contributions claimed in section 1.5 by using the features of the monitoring framework to monitor the NGDS system and by assessing how successful these features were.

There are three areas which are assessed by the case study: performance, instrumentation and soft goal specification.

### 6.1.1 Performance

The most important performance issue is the affect instrumentation code has on the performance of the monitored system. The performance overhead must be low enough that monitoring can be used as part of the normal operation of the system, so that failures due to changes in the environment of the monitored system can be detected.

The performance overhead will depend on the configuration of the monitoring system. The performance of the monitoring system is likely to vary depending on which implementation of the requirements instance model is used; the database implementation or the implementation using Java classes. The performance is also likely to depend on the characteristics of the network connection between the monitored system and the monitor server. The instrumentation overhead is likely to be lower if the connection is faster.

Evaluating the performance of the monitoring framework thus requires performance data to be collected for a number of different configurations. Unfortunately, the performance of the monitoring framework will likely vary greatly from application to application so the results from this case study will not be able to predict the performance overhead due to monitoring in other applications. The results should however be able to provide some insight into the relative performance of the different configurations of the monitoring system and show that monitoring in the deployment environment is at least feasible using the monitoring framework.

### 6.1.2 Instrumentation

The second objective of the case study is to show that the system can be effectively instrumented using the monitoring framework, allowing goals to be monitored. As there are two approaches to instrumentation within the framework (the mapping approach and instrumentation using AspectJ directly), it is necessary to determine how these two approaches compare in practice. The case study should identify how much instrumentation can be implemented using the mapping language and how often it is necessary to fall back on AspectJ. It should also determine whether the amount of code which has to be written is reasonable.

Ensuring the correctness of instrumentation code is largely left to the instrumentation developer so demonstrating correctness is not an objective of the case study other than to show that given correct instrumentation, the monitoring framework correctly processes the generated instrumentation messages.

### 6.1.3 Soft Goal Specification

The final objective is to demonstrate the use of the soft goal specification language to formally specify metrics, to show that the language is suitable for specifying metrics related to real soft goals. This suitability needs to be considered in terms of whether it is possible to express those metrics which are desired, how easy the metrics are to specify and how much additional instrumentation needs to be written so that those metrics can be monitored.

### 6.1.4 Utilisation of Monitoring Results

The case study should demonstrate that the monitoring framework can be used to detect when failures have occurred due to changes in the environment of the NGDS system. In the case of soft goals, this means determining whether the goals have failed using the information provided by soft goal displays. These displays should allow users of the monitoring framework to decide when the system is failing to satisfy a soft goal and give some indication of why that failure is occurring.

## 6.2 Formally Define Hard Goals

The goal refinement for the hard goals of the NGDS system is shown in figure 6.1 along with the agent responsibility links for those goals. The top level goal is 'Achieve[Construct Schedule]' which requires that the system should try to construct a schedule. This goal has two sub goals. The first, called 'Achieve[Assign Jobs]', corresponds to the initial phase of creating a schedule in which as many jobs as possible are placed in the schedule. The second goal, called 'Achieve[Optimise Schedule]', required that a local search algorithm should be run on the schedule to optimise it with respect to the non-functional requirements for the schedule. The goal 'Achieve[Assign Jobs]' has three sub-goals which require parallel jobs, long jobs and normal jobs to be inserted into the schedule. Jobs are placed in this order as parallel jobs have the greatest constraints on them as they must be scheduled at the same time as another job and long jobs are harder to place than normal jobs as they require a large gap in a schedule.

All of these goals are assigned to agents which are responsible for running these algorithms, which inherit from the 'Algorithm' agent. In the NGDS system, these agents all run on the same system but they are modelled as separate agents due to their importance at the requirements level.

All the leaf goals in this graph are assigned to agents in the system so they are all requirements not assumptions. As a result, it is not particularly interesting to monitor these goals for their own sake. It is however necessary to monitor these goals so that soft goals can be defined using them. These soft goals are of interest as they are affected by the environment, specifically by the types of schedules that are submitted to the system. The actual hard goals are fairly trivial to specify as there are so few limits on what behaviour is allowed by the scheduler. The only real requirements is that once a search has started it should eventually finish, although this is sufficient to allow specification of soft goals which depend on the hard goals as it makes the start and end of a search explicit in the requirements model.

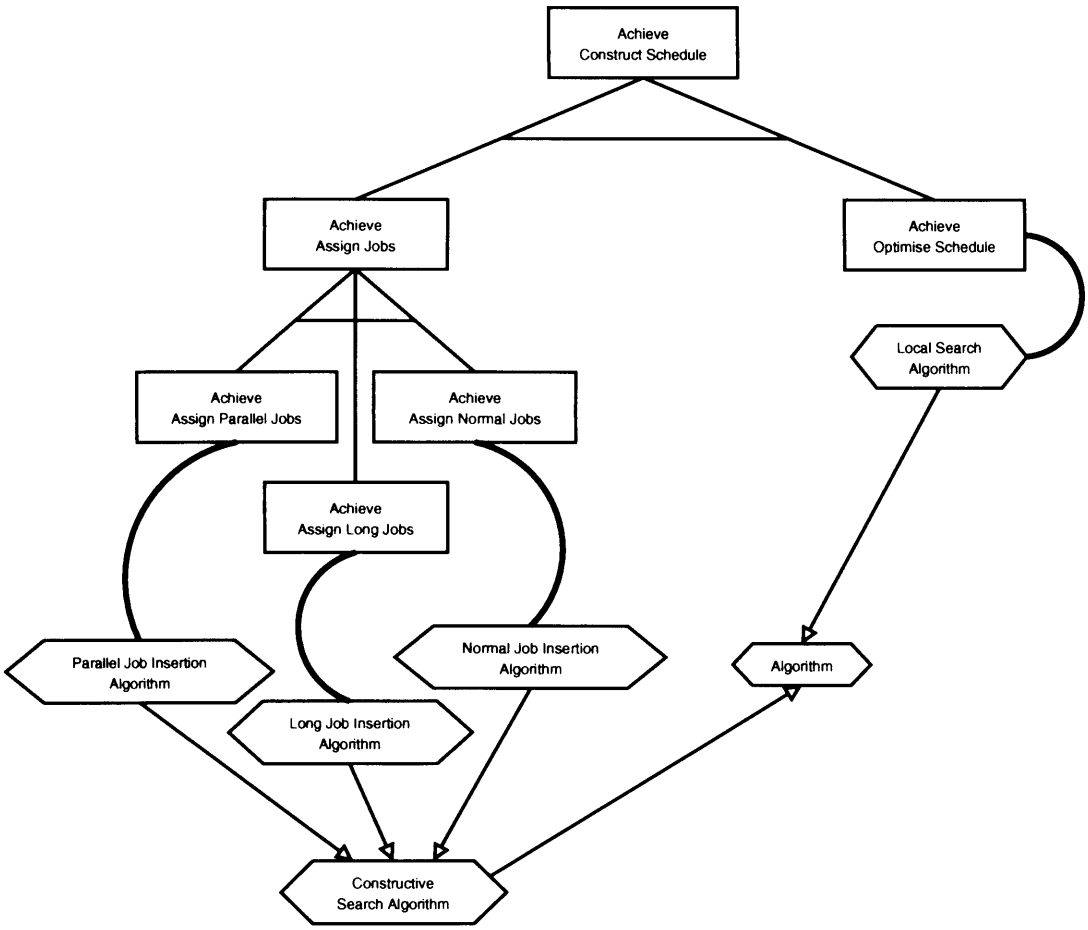



Figure 6.1: Goals refinements and agent responsibilities for the NGDS system.

The formal definition of the leaf goals are as follows:

**Goal**: Achieve[Assign Parallel Jobs]
$\forall$ a:ParallelJobInsertionAlgorithm, s:Schedule
RunningParallelJobInsertionAlgorithm(a, s) $\Rightarrow$
$\Diamond_{\leq 10}$ min$^{\neg}$ RunningParallelJobInsertionAlgorithm(a, s)

**Goal**: Achieve[Assign Long Jobs]
$\forall$ a:LongJobInsertionAlgorithm, s:Schedule
RunningLongJobInsertionAlgorithm(a, s) $\Rightarrow$
$\Diamond_{\leq 10}$ min$^{\neg}$ RunningLongJobInsertionAlgorithm(a, s)

**Goal**: Achieve[Assign Normal Jobs]
$\forall$ a:NormalJobInsertionAlgorithm, s:Schedule
RunningNormalJobInsertionAlgorithm(a, s) $\Rightarrow$
$\Diamond_{\leq 10}$ min$^{\neg}$ RunningNormalJobInsertionAlgorithm(a, s)

**Goal**: Achieve[Optimise Schedule]
$\forall$ a:LocalSearchAlgorithm, s:Schedule
RunningLocalSearchAlgorithm(a, s) $\Rightarrow$
$\Diamond_{\leq 10}$ min$^{\neg}$ RunningLocalSearchAlgorithm(a, s)

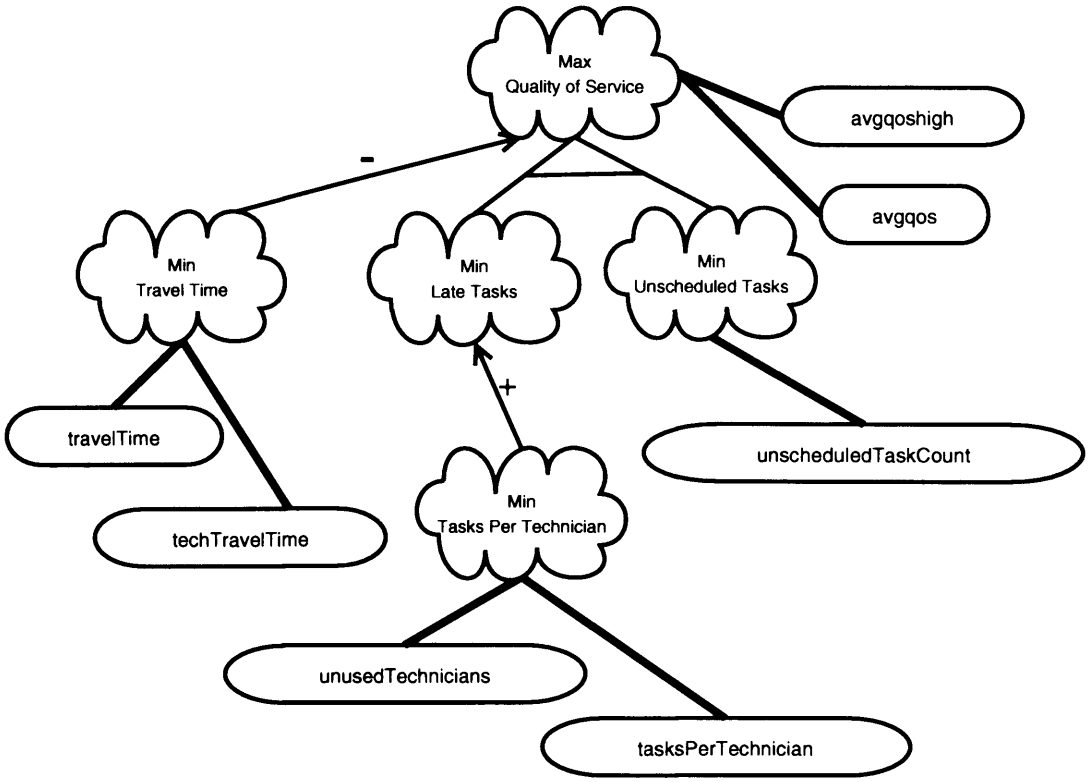



Figure 6.2: The soft goal model for NGDS system.

As can be seen, these definitions follow a similar pattern. Once each algorithm is running, it should finish running within a time limit. Because all the agents which represent the different algorithms extend the 'Algorithm' agent and because no additional attributes are monitored in the individual specialisations of the 'Algorithm' agent, it is possible to ignore the specialised agents and just monitor these goals with the labels 'a' representing a generic 'Algorithm' agent. This simplifies the monitoring problem as it means that several agents do not need to be represented.

## 6.3 Formally Define Soft Goal Metrics

There are a number of soft goals which are of interest in the NGDS system and are the focus of run-time monitoring in this case study. These soft goals, which are related to the quality of the schedule which is generated by the scheduler, are shown in figure 6.2. In this figure, soft goals are shown using the normal soft goal notation. The metrics associated with those soft goals are shown using a non-standard notation with the metrics in rounded rectangles and the associations with soft goals shown by the thick lines connecting them. It is possible for more than one metric to be associated with a single soft goal which provides the user of the framework with different ways of evaluating the satisfaction of a soft goal.

The basic measure of the quality of service of the system is the quality of service attribute, which the scheduler tries to maximise. This is represented by the soft goal 'Max[Quality of Service]'. A quality of service measure exists for each individual task in the schedule which indicates how well it has been scheduled, taking into account how late the task is or the failure to schedule the task. The quality of service for the schedule is calculated by combining the quality of service for the individual tasks in

the schedule. The metric 'avgqos' calculates the average quality of service of all the tasks in the schedule. The metric 'avgqoshigh' calculates the average for only those tasks with high priority. Metrics for only those tasks with medium or low priority are also obviously possible as well as any combination of task groups. This allows users of the framework to see the success of the system in satisfying the soft goals from the perspective of different groups of tasks which share specific properties.

The soft goals 'Min[Unscheduled Tasks]' and 'Min[Late Tasks]' together satisfy the goal 'Max[Quality of Service]'. The metric 'unscheduledTaskCount' is associated with 'Min[Unscheduled Tasks]' and has as its value the number of tasks in a schedule which are left unassigned by the schedule.

The soft goal 'Min[Tasks Per Technician]' requires that as few tasks as possible should be assigned to each technician. This soft goal contributes positively to the soft goal 'Min[Late Tasks]' as technicians which have fewer tasks assigned to them are likely to have better quality of service. This soft goal has two metrics associated with it. The metric 'tasksPerTechnician' calculates the average number of tasks which are assigned to each technician. The metric 'unusedTechnicians' counts the number of technicians which have not had any tasks assigned to them.

Another property of a schedule which is desirable is that the travel time between tasks can should be minimised, which is represented by the goal 'Min[Travel Time]'. This soft goal conflicts to some extent with the goal 'Max[Quality of service]' in that a schedule optimised to minimise travel time is likely to be different from one optimised for quality of service, although in many cases decreasing travel time is also likely to increase quality of service. To demonstrate this conflict, consider two tasks which need to performed in locations which are close to each other geographically. Quality of service can be maximised by having a different technician perform each task. Travel time will be minimised by having one technician perform both tasks but this will delay the second task. Two metrics associated with this soft goal are shown in the diagram. The metric 'travelTime' considers travel time from the perspective of a task by calculating the average time taken to reach each task. The metric 'techTravelTime' considers these metrics from the point of view of technicians by calculating the average time each technician has to travel to satisfy the schedule.

### 6.3.1 Define Goal Instance Metrics

To monitor the goal instance metrics described above, it is necessary to formally define them and express them in the XML language for specifying goal instance metrics. All of the goal instance metrics are associated with the four leaf goals in figure 6.1 and they are evaluated whenever an instance of one of these goals is satisfied.

The goal instance metric 'avgqos' is associated with the soft goal 'Max[Quality Of Service]' and is the average quality of service value of all tasks in a schedule after a search algorithm has finished running a search stage. It is defined as:

$$\begin{aligned} \forall\ g:\ & AssignParallelJobs \cup AssignLongJobs \cup \\ & AssignNormalJobs \cup OptimiseSchedule, \\ t:\ & Task \\ \text{avgqos}(g) &= \text{Avg}(t_1.qos, t_2.qos, \ldots t_n.qos) \\ \text{where } t \in \{t_1, t_2, \ldots t_n\} &\Leftrightarrow \text{InSchedule}(t, g.s) \end{aligned}$$

This definition uses the relationship 'InSchedule' which is true for a task that belongs to a particular schedule. It is true whether or not that task has actually been success-

fully assigned a technician and time slot as long as the task is one which should be assigned. This is an example of a relationship which is not used in the definition of the hard goals which are monitored but which appears in the definition of a soft goal metric. Instrumentation code must be written for these relationships in addition to the code necessary to monitor the hard goals. The 'Task' entity also appears only in the specification of soft goals and requires additional instrumentation code.

The value of the goal instance metric 'unscheduledTaskCount', which is associated with the soft goal 'Min[Unscheduled Tasks]', is the number of tasks which have not been scheduled at the end of a search stage. The metric is formally defined as:

$$\begin{aligned}&\forall\ g: AssignParallelJobs \cup AssignLongJobs \cup \\ &\qquad AssignNormalJobs \cup OptimiseSchedule, \\ &\quad t: Task \\ &\text{unscheduledTaskCount}(g) = \\ &\text{Count}(t_1.unscheduled, t_2.unscheduled, \ldots t_n.unscheduled) \\ &\text{where } t \in \{t_1, t_2, \ldots t_n\} \Leftrightarrow \text{InSchedule}(t, g.s) \wedge unscheduled = false\end{aligned}$$

The goal instance metric 'travelTime' is associated with the soft goal 'Min[Travel Time]' and measures the average time taken to reach a task in a schedule. The metric is defined as:

$$\begin{aligned}&\forall\ g: AssignParallelJobs \cup AssignLongJobs \cup \\ &\qquad AssignNormalJobs \cup OptimiseSchedule, \\ &\quad t: Task \\ &\text{travelTime}(g) = \text{Avg}(t_1.travelTime, t_2.travelTime, \ldots t_n.travelTime) \\ &\text{where} t \in \{t_1, t_2, \ldots t_n\} \Leftrightarrow \text{InSchedule}(t, g.s)\end{aligned}$$

The soft goal model also contains three metrics associated with technicians, which can be measured using three goal instance metrics. Associated with the soft goal 'Min[Travel Time]' is the goal instance metric 'techTravelTime' which measures the average travel time of each technician which is used by a schedule. It is formally defined as:

$$\begin{aligned}&\forall\ g: AssignParallelJobs \cup AssignLongJobs \cup \\ &\qquad AssignNormalJobs \cup OptimiseSchedule, \\ &\quad t: Technician \\ &\text{techTravelTime}(g) = \text{Avg}(t_1.travelTime, t_2.travelTime, \ldots t_n.travelTime) \\ &\text{where } t \in \{t_1, t_2, \ldots t_n\} \Leftrightarrow \text{UsableBySchedule}(t, g.s)\end{aligned}$$

The goal instance metric 'unusedTechnicians' is associated with the soft goal 'Max[Tasks Per Technician]' and simply counts the number of technicians which are available to a schedule but are not assigned tasks in that schedule. It is defined as follows:

$$\begin{aligned}&\forall\ g: AssignParallelJobs \cup AssignLongJobs \cup \\ &\qquad AssignNormalJobs \cup OptimiseSchedule, \\ &\quad t: Technician \\ &\text{unusedTechnicians}(g) = \\ &\text{Count}(t_1.allocatedTasks, t_2.allocatedTasks, \ldots t_n.allocatedTasks) \\ &\text{where } t \in \{t_1, t_2, \ldots t_n\} \Leftrightarrow \text{UsableBySchedule}(t, g.s) \wedge t.allocatedTasks = 0\end{aligned}$$

Finally, 'tasksPerTechnicians' is the goal instance metric associated with the soft goal 'Max[Tasks Per Technician]'. It measures the average number of tasks assigned to a technician. It is formally defined as:

$$\begin{aligned} &\forall \; g: \; AssignParallelJobs \cup AssignLongJobs \cup \\ &\qquad AssignNormalJobs \cup OptimiseSchedule, \\ &\qquad t: \; Technician \\ &\text{tasksPerTechnician}(g) = \\ &\text{Avg}(t_1.allocatedTasks, t_2.allocatedTasks, \ldots t_n.allocatedTasks) \\ &\text{where } t \in \{t_1, t_2, \ldots t_n\} \Leftrightarrow \text{UsableBySchedule}(t, g.s) \end{aligned}$$

It is also possible to define goal instance metrics which make use of the attributes of the 'Task' and 'Technician' entities to restrict the set of entities which are used in the calculation of the metric. For example, it is possible to calculate the average quality of service for only those tasks which have a high priority:

$$\begin{aligned} &\forall \; g: \; AssignParallelJobs \cup AssignLongJobs \cup \\ &\qquad AssignNormalJobs \cup OptimiseSchedule, \\ &\qquad t: \; Task \\ &\text{avgqoshigh}(g) = Avg(t_1.qos, t_2.qos, \ldots t_n.qos) \\ &\text{where } t \in \{t_1, t_2, \ldots t_n\} \Leftrightarrow \text{InSchedule}(t, g.s) \wedge t.\text{priority} = \text{'high'} \end{aligned}$$

Other attributes of the task metric which it makes sense to use in this way are 'jobType', 'commitType' and 'dueDate' attributes. It is also possible to use more than one of these attribute so that it is possible, for example, to calculate the average quality of service for tasks with high priority which are due to be completed by the end of the next day, although a new goal instance metric needs to be defined for each combination which is monitored.

### 6.3.2 Define Soft Goal Displays

The soft goal metrics can be displayed to the users of the monitoring framework so that they can assess the operation of the system with respect to the soft goals. The goal instance metrics described previously can be displayed using a history gauge to show the value of the metrics over time. An example of such a display specification is the following listing which defines a display which uses a history gauge to show the average quality of service value after each search completes.

```
1  <Display title="QoS on completion" class="HistoryGauge">
2      <GaugeParameter name="yMin" value="19"/>
3      <GaugeParameter name="yMax" value="23"/>
4      <GaugeParameter name="yInterval" value="1"/>
5      <GaugeParameter name="period" value="2400000"/>
6      <GaugeParameter name="xInterval" value="500000"/>
7      <GaugeParameter name="label1" value="qos"/>
8      <Value name="avgqos" goal="OptimiseSchedule"/>
9  </Display>
```

Similar displays can be defined for other goal instance metrics.

## 6.4 Instrument the Target System

Monitoring the NGDS system requires that instrumentation code is inserted into the system. There are two approaches to this within the monitoring framework, as described in chapter 4. The simplest approach, from the point of view of the developer, is to write a mapping between entities and relationships in the requirements model and corresponding classes, methods and attributes in the implementation of the system. This mapping is used to generate instrumentation code written in AspectJ. The second approach is to write the instrumentation in AspectJ directly. This approach is more flexible but also has a greater degree of complexity from the developers perspective.

The instrumentation of the NGDS system was performed using a combination of both of these approaches. This is reasonably simple to achieve as the two methods are complementary since mappings are used to generate AspectJ code which can interact with manually written AspectJ code as necessary. Instrumentation is written individually for each entity and relationship in the requirements model so the choice can be made individually for each entity and relationship as to whether to use a mapping or a instrumentation aspect to develop the instrumentation.

Instrumentation needs to be written for the four leaf goals in the goal graph in figure 6.1. Each of these goals has a relationship for which instrumentation needs to be written as well as for the entities 'Algorithm' and 'Schedule'. The relationship 'InSchedule' and the entities 'Task' and 'Technician' are also used in the definition of the soft goal metrics so these also require instrumentation code.

The instrumentation necessary for the entity 'Schedule' is very simple as this entity has no attributes and is associated directly with an implementation level object. The instrumentation for this object is generated from the following mapping:

```
1  <Mapping name="ngds">
2      <!-- Map the KAOS entity Schedule to the
3           Java class DS_Schedule -->
4      <Object name="Schedule"
5          objectElement="//UML:Class[@name='DS_Schedule']">
6      </Object>
7  </Mapping>
```

The aspect generated for the entity 'Schedule' creates an aspect instance for each instance of the Java class `DS_Schedule`. The monitor is told to create a new instance of the entity 'Schedule' as soon as the aspect instance is created. The entity is created without providing an identifier for the instance so a globally unique identifier is automatically generated. This mapping is used to generate the aspect shown below:

```
1  package monitor.ngds.mapping;
2
3  privileged public aspect ScheduleInstance
4             extends ScheduleType pertarget(
5             execution(com.bt.ngds.mdl.DS_Schedule.new(..))) {
6
7      pointcut initPointcut(com.bt.ngds.mdl.DS_Schedule
8                            targetClass) :
9          execution(com.bt.ngds.mdl.DS_Schedule.new(..)) &&
10         target(targetClass);
11
```

```
12      after(com.bt.ngds.mdl.DS_Schedule targetClass) :
13              initPointcut(targetClass) {
14          initInstance();
15      }
16  }
```

In this case, the generated aspect is quite a bit longer than the three line mapping, although a hand written aspect might be a bit more concise.

The mapping aspect for the 'Task' entity is the most complex aspect. The listing for this aspect is:

```
1   // Relates the Task entity to the implementation
2   // by extending the TaskType class.
3   // An aspect instance is created for each instance of
4   // DS_Task which is created.
5   public privileged aspect TaskInstance extends TaskType
6                   pertarget(execution(DS_Task.new(..))) {
7
8       // The KAOS Task entity represented by this aspect
9       // maps to this object.
10      private DS_Task task;
11
12      // When the DS_Task object is created, this advice
13      // sends a message to create a new Task entity
14      // instance.
15      after(DS_Task task) : execution(DS_Task.new(..)) &&
16                            target(task) {
17          initInstance();
18          this.task = task;
19      }
20
21      // Recalculates the values of the attributes of the
22      // task entity and informs the monitor of the new
23      // values.
24      public void resetAttributes() {
25          //Set QoS
26          qosUpdated(task.getQosCost());
27
28          //Set job type
29          if (task.isLongJob()) {
30              jobTypeUpdated("Long");
31          } else if (task.hasParallelJobParent() ||
32                     task.isParallelJobHead()) {
33              jobTypeUpdated("Parallel");
34          } else {
35              jobTypeUpdated("Normal");
36          }
37
38          //Set commit type
39          commitTypeUpdated(task.getCommitType());
40
```

```
41          //Set priority
42          ...
43
44          //Set due date
45          ...
46
47          //Set travel time
48          ...
49
50          //Set unassigned
51          ...
52      }
53  }
```

An instance of this aspect is created for each instance of the `DS_Task` class. When an instance of the aspect is created it stores the instance of `DS_Task` with which it is associated so that the values of attributes can be obtained from it when `resetAttributes()` is called. The system is instrumented in this way so that the attribute values are not reported to the monitor every time they change but are instead only reported when they are needed, which is whenever a search stage is complete. This approach is necessary for performance reasons, as otherwise a very large number of changes to these attribute values would occur during the execution of a search stage which are not necessary for the monitoring of the system. The need for this optimisation is the reason why it is not possible to use the mapping approach to provide instrumentation for these entities.

The aspect for the 'Technician' entity follows a similar pattern to that of the 'Task' entity. An aspect is instantiated for each instance of the `DS_Technician` class and the attributes are set by calling a `resetAttributes()` method on the object.

The aspect for the 'Algorithm' entity is:

```
1   // Relates the Algorithm entity to the implementation
2   // by extending the AlgorithmType class.
3   // An aspect instance is created for each instance of
4   // SingleSolutionMethod which is created.
5   public aspect AlgorithmInstance extends AlgorithmType
6           pertarget(execution (SingleSolutionMethod.new(..))) {
7
8       // Keep track of the number of swap and insert moves
9       // which the algorithm has performed.
10      private int insertMoves = 0;
11      private int swapMoves = 0;
12
13      // Tells monitor that a new instance of the Algorithm
14      // entity has been created whenever an instance of
15      // this aspect is created.
16      public AlgorithmInstance() {
17          initInstance();
18      }
19
20      // Pointcut which detects when a move has been performed
21      public pointcut successfulMove(Move move) :
```

```
22          execution(double HeuristicSolution.
23              performMove(Move)) &&
24          args(move);
25
26      // Whnever a move is eprformed, this advice updates the
27      // move counter.
28      before(Move move) : successfulMove(move) {
29          if (move instanceof InsertMove) {
30              insertMoves++;
31          } else if (move instanceof SwapMove) {
32              swapMoves++;
33          }
34      }
35
36      // Method which informs the monitor of the number
37      // of moves which have been performed.
38      public void resetAttributes() {
39          insertMovesUpdated(insertMoves);
40          swapMovesUpdated(swapMoves);
41      }
42
43  }
44
```

The algorithm entity corresponds to the `SingleSolutionMethod` class so an instance of this aspect is created for each instance of this class. The 'Algorithm' entity has attributes for the number of swap and insert moves which have been performed by the algorithm. The aspect contains a pointcut which matches whenever a move operation is performed on the schedule. This pointcut does not update the attribute immediately, as this would be an excessive drain on the performance of the scheduling system, but instead keeps a count as moves are performed and updated and provides a method which is called once the algorithm has completed. This allows the number of moves to be detected when an algorithm completes but not during a search.

The aspect for the relationship 'RunningParallelJobInsertionAlgorithm' handles both updates the status of this relationship and triggers the update of the attributes of the tasks, technicians and the algorithm instance. The listing is as follows:

```
1   // Relates the RunningParallelJobInsertionAlgorithm
2   // relationship to its implementation by extending the
3   // relevant appropiate helper class.
4   // An instance of this aspect exists during each execution
5   // of the runAlogrithm method.
6   public aspect RunningParallelJobInsertionAlgorithmInstance
7       extends RunningParallelJobInsertionAlgorithmType
8       percflow( execution(
9         void NGDSParallelJobInsertion.runAlgorithm())){
10
11      // Pointcut which matches when the runAlgorithm method
12      // is called and gets the implementation object
13      // associated the algorithm entity.
14      public pointcut
```

```
15        runAlgorithm(SingleSolutionMethod searchMethod) :
16        execution(void NGDSParallelJobInsertion.
17                  runAlgorithm()) &&
18        target (searchMethod);
19
20      before(SingleSolutionMethod searchMethod) :
21          runAlgorithm(searchMethod) {
22          DS_Schedule schedule = (DS_Schedule)searchMethod.
23              getHeuristicProblem().getProblem();
24
25          // Informs the monitor that a new isntance of the
26          // relationship has been created.
27          initInstance(
28              AlgorithmInstance.aspectOf(searchMethod),
29              ScheduleInstance.aspectOf(schedule));
30      }
31
32      // Advice whcih runs after execution of the algorithm
33      after(SingleSolutionMethod searchMethod) :
34          runAlgorithm(searchMethod) {
35
36
37          // Inform monitor of the current value of the
38          // attributes of all the Task entities in the
39          // schedule.
40          DS_Schedule schedule = (DS_Schedule)searchMethod.
41              getHeuristicProblem().getProblem();
42
43          for (int i = 0; i < schedule.getNTasks(); i++) {
44              DS_Task task = (DS_Task) schedule.getTask(i);
45              TaskInstance.aspectOf(task).resetAttributes();
46          }
47
48          // Inform monitor of the current value of the
49          // attributes of all the Technician entities in the
50          // schedule
51          for (int i = 0; i < schedule.getNResources(); i++) {
52              DS_WTM tech = (DS_WTM)schedule.getResource(i);
53              TechnicianInstance.aspectOf(tech).
54                  resetAttributes();
55          }
56
57
58          // Inform monitor of the current value of the
59          // attributes of the Algorithm entity
60          AlgorithmInstance.aspectOf(searchMethod).
61                              resetAttributes();
62
63          // Tell the monitor that this relationship instance
64          // has been destroyed.
```

```
65              destroyInstance();
66          }
67      }
68
69
```

The aspects for the other relationships: 'RunningLongJobInsertionAlgorithm', 'RunningNormalJobInsertionAlgorithm' and 'RunningLocalSearchAlgorithm', are similar to this one. An instance of this aspect is instantiated whenever the `runAlgorithm` method of the `NGDSParallelJobInsertion` class, which starts the parallel job insertion algorithm, is called. A relationship instance is created with the `initInstance` method immediately after the aspect is created. The two objects passed as roles to the method are the `SingleSolutionMethod` object which represents the algorithm and the `DS_Schedule` object which represents the schedule. After the search is complete, the relationship instance is destroyed with a call to `destroyInstance()`. Additionally, the task and technician and algorithm objects are updated at this stage by calling `resetAttributes()` on them. As these attributes are only updated when these methods are called, at the end of each algorithm's run, the value of these attributes is only actually correct at this time. This is a trade-off which provides better performance in the target system at the expense of more limited monitoring information. In practice, only the monitoring information which is provided is really necessary.

The soft goals use the relationships 'InSchedule' and 'UsableBySchedule' which identify which tasks need to be scheduled and which technicians are available to perform those tasks. The aspect for the relationship 'InSchedule' is:

```
1   // Relates the InSchedule relationship to the
2   // instances of the relationship which exist at
3   // the implementation level. This is done using
4   // a single aspect instance which uses introductions
5   // to associate an instance of the InSchedule type
6   // with each DS_Task instance.
7   public privileged aspect InScheduleInstance {
8       private InScheduleType DS_Task.instance;
9
10      public InScheduleType DS_Task.getInScheduleInstance() {
11          return instance;
12      }
13
14      // Ensures that the "after" advice in this aspect will
15      // be executed after that in TaskInstance so that
16      // the Task is created before the relationship in
17      // which it is a role.
18      declare precedence : InScheduleInstance, TaskInstance;
19
20
21      // Creates a new instance of the InSchedule
22      // relationship whenever a DS_Task object is created
23      after(DS_Task task) :
24          execution(DS_Task.new(..)) && this(task) {
```

```
25
26          task.instance = new InScheduleType();
27          task.instance.initInstance(
28              TaskInstance.aspectOf(task),
29              ScheduleInstance.aspectOf(task.getSchedule()));
30      }
31
32
33      // Destroys the InScheudle instance when the task is
34      // removed from a schedule.
35      after(DS_Task task) :
36          execution(void Schedule.removeTask(Task)) &&
37          args(task) {
38
39          task.instance.destroyInstance();
40      }
41  }
```

Because each task is assigned to a schedule as soon as it is created, the relationship instance should be created when a `DS_Task` object is created, although it is necessary to use 'declare precedence' to make sure that the aspect for the 'Task' entity executes before this one. It is generally the case that when two instrumentation aspects have advice that executes at the same time, advice associated with entities should execute before advice associated with relationships. This is because the relationship may have the entity as a role while the entity will not generally be affected by changes to the state of the relationship.

This aspect does not use a 'pertarget' clause, so there is only a single aspect instance. The information for each instance is added to the `DS_Task` class using the AspectJ inter-type declaration facility which allows additional methods and fields to be defined for existing classes. This is necessary because the pointcut at which this relationship is destroyed is defined at a method of the `DS_Schedule` class rather than the `DS_Task` class and so would not match for an aspect which is instantiated for each `DS_Task` instance.

The aspect uses an inter-type member declaration, at line 8, to add an instance of `InScheduleType` directly to the `DS_Task` class. At line 23, a new relationship instance is created whenever a new `DS_Task` object is created. This advice creates a new instance of the `InScheduleType` class and stores it in the `DS_Task` object which has been created. The relationship is destroyed when the advice at line 35 executes, when the `removeTask` method is called.

The aspect for the relationship 'UsableBySchedule' follows the same pattern as the aspect for the 'InSchedule' relationship.

## 6.5 Results

### 6.5.1 Performance

Instrumentation of source code will inevitably affect the performance of the monitored system. It needs to be determined for each application individually whether the performance penalty is acceptably small. When monitoring takes place during development, significant performance penalties can be tolerated. The aim of the monitoring framework is that monitoring should take place "on-line", as part of the normal operation of

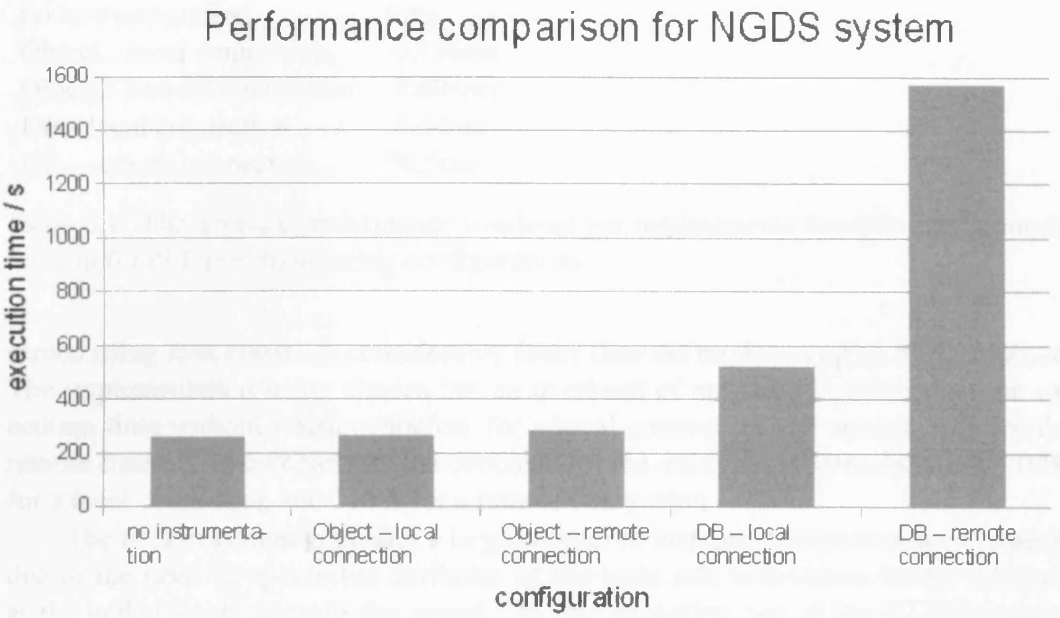



Figure 6.3: Average execution times for NGDS system in various configurations. From left to right: execution time without monitoring, monitoring over a local area connection with a Java implementation of the requirements instance model, over a wide area connection with the Java implementation, over a local area connection with the database implementation of the requirements instance model, over a wide area connection with the database implementation.

the system in the deployment environment, so the performance overhead must be much smaller.

The performance of the NGDS system while being monitored was evaluated by measuring the time taken for a complete run, in which a single schedule is created using the same input data each time. This measurement was made for a number of different configurations. In each configuration, the NGDS system was executed five times and the average execution time is presented in figure 6.3.

The first result shows the average time taken to execute the NGDS system in the absence of instrumentation code. This result can be compared with the time taken to execute the system when it is monitored. The monitor was run on a different machine than the target system and results were obtained using two different machines to run the monitor. In the first case, the monitor was run on a machine connected to the one running NGDS over a local area network. This provided a low low latency (less than one millisecond), high bandwidth, connection.In the second case, the monitor was run on a machine in a remote location, connected to the machine running NGDS over the Internet, with a latency of ten to twenty milliseconds and a bandwidth of around 1Mbit/s. The monitoring framework also provides two different architectures for storing the requirements instance model, as described in 4.2.2. Both of these architectures were tested, giving four four possible configurations for which results were obtained, corresponding to the different combinations of requirements instance model and network connection.

The results show that the architecture in which the object model instance is repre-

| | |
|---|---|
| no instrumentation | 0ms |
| Object - local connection | 0.196ms |
| Object - remote connection | 0.604ms |
| DB - local connection | 6.44ms |
| DB - remote connection | 38.5ms |

Table 6.1: The average performance overhead per requirements instance model modification for different monitoring configurations.

sented using Java classes is considerably faster than the model using an SQL database. The implementation using classes has an overhead of around 3%, relative to the execution time without instrumentation, for a local connection and around 10% for the remote connect. In comparison, the overhead for the database implementation is 102% for a local connection and 513% for a remote connection.

The NGDS system generates a large number of instrumentation messages, mainly due to the need to update the attributes of the tasks and technicians in the schedule at the end of each stage in the search. In one execution run of the NGDS system, 40747 modifications are made to the requirements model instance, of which 95% are modifications to attribute values. Using this information, it is possible to calculate approximate figures for the performance overhead incurred for each modification to the requirements instance model (table 6.1).

In this situation, the database implementation of the requirements instance model clearly has performance problems, particularly when using a slower connection rather than a high speed, local area connection. The implementation of the requirements instance model using Java classes does provide sufficiently good performance to be used in during normal operation of the system.

### 6.5.2 Instrumentation

The instrumentation of the goals for the NGDS system requires a total of 377 lines of code. This provides instrumentation for a total of four entities and six relationships. Of those lines of code, 129 are used to write instrumentation for the four entities and the other 248 are used to write instrumentation for the five relationships. This gives an average of 32.25 lines of code for an entity and 41.33 lines for a relationship.

The greatest concern which emerges is that writing the instrumentation aspects is quite time consuming, despite the fairly modest requirements in terms of actual code. The most difficult part of the process is writing the pointcuts which determine when the instrumentation executes. In practice, this requires looking at the source code of the implementation in addition to the documentation automatically generated by Javadoc. It might also be possible to work from detailed design documents if these are available.

As can be seen from the example instrumentation aspects presented, the aspects can become quite complex. For example, the 'RunningSearchStage' aspects includes loops over all tasks and technicians and the 'Task' aspect requires calling various methods on the task objects to determine the values of attributes. This sort of complexity is the reason why the approach of generating the translation aspects from a mapping between the requirements level and the implementation level is not able to capture all the mappings which are necessary. The approach works when the design of the implementation system is fairly simple. When a real world system of sufficient complexity is examined, cases emerge which are not covered by the mapping language.

In this case study, the mapping language was only able to be used for one of the entities, the 'Schedule' entity, and none of the relationships. This was mainly due to performance problems with the 'Task' and 'Technician' entities which also affected the four relationships related to running the search stages. These four relationships otherwise map to the implementation so that the relationship is true during the execution of the method `runAlgorithm`, which is one of the types of mapping supported by the mapping language. The relationships 'InSchedule' and 'UsableBySchedule' could not make use of the mapping language because they do not match the types of mappings supported by the mapping language as these relationships become true when one method is executed and false when a different method of a different class is executed.

### 6.5.3 Soft Goal Specification

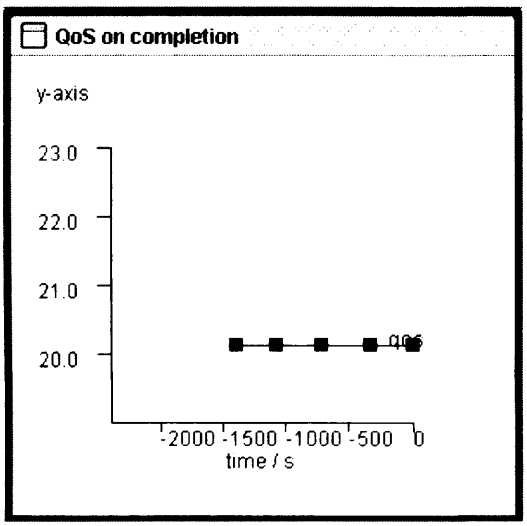
The case study demonstrated that the soft goal metrics in this example could be successfully specified using the soft goal specification language and that instrumentation could be developed to allow these metrics to be monitored. These specifications made use of goal instance metrics rather than goal aggregate metrics. The use of goal aggregate metrics were demonstrated in the Limewire example. These specifications required 81 lines of additional specification in XML for the six different metrics. The specification of these metrics made use of two additional relationships, two additional entities and 11 additional attributes which would not otherwise have to be monitored. The specification of the soft goal metrics was found to be relatively easy but implementing the additional instrumentation necessary to monitor the soft goal metrics was more time consuming.

### 6.5.4 Utilisation of Monitoring Results

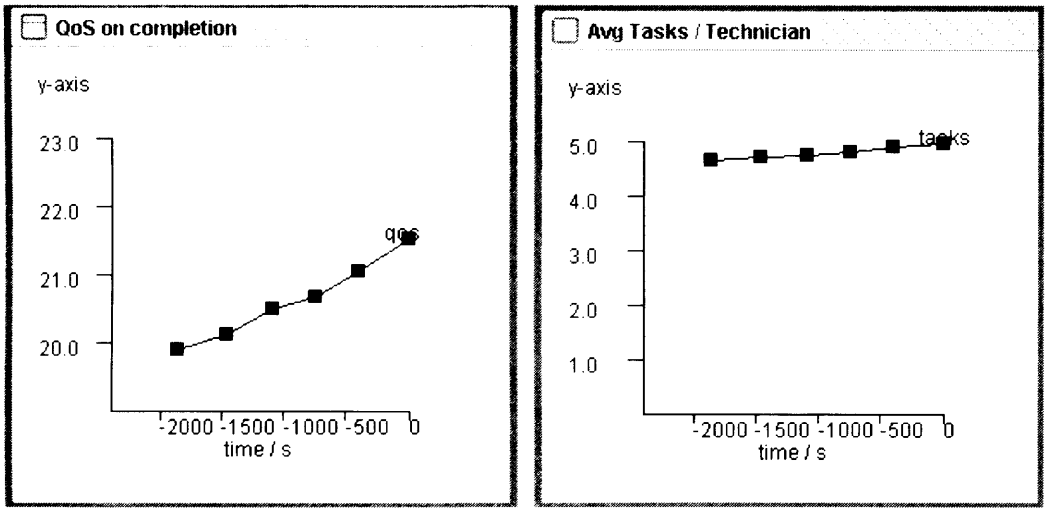
Soft goal displays were defined to demonstrate the display of soft goal metrics and how these metrics can be used to identify problems. An example scenario was used in which the number of tasks which need to be assigned by a schedule is gradually increased from 771 tasks to 851 tasks. The results of this are shown in figure 6.4. In figure 6.4(a) the number of tasks remains constant and the quality of service also remains constant. In figure 6.4(b), the number of tasks increases with time, resulting in a deterioration of quality of service (higher values indicate worse quality of service).

If the user of the monitoring framework decides that the quality of service is too poor, it might be necessary to investigate the reasons for the problem. In this case, the problem can be discovered by examining the display on the right side of figure 6.4(b), which shows the history of the soft goal metric 'tasksPerTechnician'. This shows that the average number of tasks assigned to a technician is increasing. As the graph of soft goals in figure 6.2 shows, the metric 'tasksPerTechnician' is related to the soft goal 'Min[Tasks Per Technician]', so the satisfaction of this soft goal is getting worse. This also results in a reduction of the satisfaction of 'Max[Quality Of Service]' as the soft goal 'Min[Tasks Per Technician]' is positively associated with the goal.

If the quality of service is considered to be too poor then something must be done to rectify the situation. As the problem is too few technicians, the most obvious solution is to increase the number of technicians. In this case, the environment is changed to ensure that the system continues to function. It is likely that this solution will not be possible so other solutions might need to be considered. It may be that there exists a scheduling algorithm which will give better results with limited numbers of technicians or which will emphasis high priority tasks by cancelling low priority tasks. If this is the




(a) With a constant number of tasks in the schedule, the quality of service remains the same.




(b) With an increasing number of tasks, the quality of service declines.

Figure 6.4: Display of the quality of service over time.

case then it is a straightforward modification to the system to select the new algorithm so that it is used for future scheduling operations. In this case, the system is modified so that it can continue to satisfy the requirements.

## 6.6 Summary

In this case study, the capabilities of the monitoring framework were evaluated by implementing monitoring for the NGDS system. The case study evaluated the performance of the monitoring framework, the success of the instrumentation mechanism and the use of the soft goal specification language.

The performance of the NGDS system was tested using the two implementations of the requirements instance model which are provided by the framework. The implementation making use of a Java classes to store this information was found to have a performance overhead of only a few percent. This means that it should be suitable for

most "on-line" monitoring situations. The database implementation was much slower, particularly when operating over a wide area network. This would seem to be mainly due to the large number of modifications to the requirements instance model which occur in this case study, combined with less efficient communication between the target system and the monitor. It had been anticipated that the number of modifications to the model would be relatively small but this does not seem to be the case in practice, at least in this example. A database implementation may still have advantages in some situations, such as providing greater reliability, but this has not been demonstrated.

It was found to be reasonably easy to instrument the NGDS system using AspectJ and all the necessary instrumentation could be written in a non-invasive manner, making use of the framework of support classes. The mapping language was of only limited use in this example. In part, this may be attributed to the need to optimise the instrumentation code. In other cases, this is due to the mapping language not supporting certain types of mapping.

The soft goal specification language was successfully used to formally specify soft goal metrics for the NGDS system. These metrics made use of quite complex goal instance metrics. Some additional instrumentation was required to allow these metrics to be evaluated.

Soft goal displays were defined which were able to show failure to satisfy soft goals at run time in an example scenario. By using a number of goal instance metrics associated with different soft goals, it is possible to investigate reasons why a system is failing to satisfy soft goals.

# Chapter 7

# Conclusions and Future Work

The aim of the work which has been presented in this thesis is to provide a framework for monitoring computer systems at run time to determine whether they satisfy a KAOS goal-oriented requirements specification. This framework covers both hard goals, formally defined using temporal logic, and soft goals, for which metrics are formally defined which can be used to evaluate the satisfaction of these goals at run time. The monitoring framework requires the monitored system to be instrumented so that it emits events at run time to a monitor server. The monitor server uses the events it receives to determine whether the monitored system has violated hard goals and to calculate the value of soft goal metrics.

Instrumentation of the monitored system is achieved using AspectJ, an aspect-oriented extension to the Java programming language, to obtain information about the execution of the system and to translate that information into events which represent changes in the requirements level instance model of the running system. The use of AspectJ for instrumentation allows instrumentation code to be kept separate from the code of the target system and allows instrumentation of any system programmed in Java, rather than of particular types of system such as those that use a specific architecture or middleware. As AspectJ advice has all the capabilities of the Java language, it is possible to describe complex translations from implementation level events to changes in the requirements instance model.

In addition to the approach using AspectJ directly, a mapping language which relates a KAOS object model to the implementation of a system can be used to generate the AspectJ code necessary to instrument a system, as long as the mapping from KAOS objects to implementation objects is relatively simple. Where the mapping language does not support a particular type of mapping, the mapping language cannot be used and the instrumentation developer must fall back on the AspectJ language. Similarly, the need to optimise instrumentation to minimise the performance overhead in the monitored system may prevent the use of the mapping language.

The monitor server detects failure of the monitored system to satisfy goals and calculates the value of soft goal metrics at run time, while the monitored system executes. This is done using the requirements instance model which represents the current state of the monitored system as a concrete instance of the requirements model, thus linking the run-time behaviour of the system back to the requirements specification. The requirements instance model is modified by instrumentation messages from the monitored system and whenever a changes occurs, goals which reference that part of the model are checked for failure and soft goal metrics are recalculated. The requirements instance model of the monitored system is stored either in a database or in memory

using Java objects. It is was not initially clear which was the better approach so two approaches are provided giving the developer flexibility to determine which implementation is most appropriate in a given situation. One of the aims of the evaluation was to asses the two implementations of the requirements instance model.

Soft goal monitoring requires that metrics are formally defined which are indicative of the satisfaction of soft goals. To allow this, a soft goal specification language has been developed. This language makes use of the KAOS goal and object models, which are used to defined hard goals, allowing simpler specification and reuse of existing instrumentation code. Soft goal metrics are visually displayed using configurable gauges so that the users of the monitoring framework can evaluate whether the soft goals associated with metrics are satisfied.

The suitability of this framework for run-time monitoring of KAOS requirements specifications has been demonstrated in a substantial case study using a work force scheduling system. The performance overhead of instrumentation code was evaluated in different configurations. The capabilities of the mapping language and direct instrumentation using AspectJ were also investigated.

This final chapter reviews the contributions which have been made by this thesis, evaluates how well the initial aims of the work have been satisfied and considers possible future work in this area.

## 7.1 Contributions and Results

The contribution made by this thesis is the development of a framework for monitoring a system against a goal-oriented requirements specification at run time which supports instrumentation of the monitored system to allow monitoring to take place. The framework supports monitoring for both hard and soft goals and provides graphical feedback so that satisfaction of soft goals can be determined by users of the framework. The remainder of this section offers some conclusions relating to the main features of the monitoring framework and compares these results with related work on monitoring.

### 7.1.1 Instrumentation

Instrumentation of the target system is a crucial part of run-time monitoring. The results of monitoring are only as good as the information which the instrumentation code provides to the monitor.

The monitoring framework described in this thesis uses AspectJ to implement instrumentation. The framework includes code to allow instrumentation to communicate with the monitor, hiding the details of this communication from the instrumentation developer. Helper code is automatically generated which allows the instrumentation developer to translate implementation level events to requirements level events by calling methods in the generated classes which correspond to changes in the requirements instance model. The developer does not need to worry about how the requirements instance model is represented in the monitor. The interface provided by the generated helper code ensures that the messages generated by the instrumentation code are valid changes to the requirements instance model which helps to ensure that the instrumentation code is correct. These helper classes also help to structure the instrumentation code with one aspect for each relationship or entity type.

A mapping language was also created which allows the instrumentation developer to express the relationship between the requirements and implementation levels. The mapping language an XML application as a concrete syntax and makes use of XPaths

to identify elements of the implementation. This mapping is used to automatically generate AspectJ instrumentation code.

These two approaches complement each other. The mapping approach allows a concise and explicit expression of the mapping between the requirements and implementation levels but is not able to express more complex mappings. The AspectJ approach provides complete freedom to express these mappings using all the capabilities of the Java programming language and AspectJ but at the expense of some clarity.

The two instrumentation approaches are demonstrated in the case study presented in chapter 6. In this case study, the mapping language was found to be of only limited usefulness. Most of the instrumentation was written directly in AspectJ, allowing more complex mappings to be specified and for the instrumentation code to be optimised so that the generated messages are minimised. In the Limewire example, it was possible to make much more use of the mapping language, in part because the performance demands were less significant as far fewer instrumentation messages were generated.

The performance overhead of the instrumentation code depends on both the efficiency of the instrumentation code and the speed of communication with the monitor but generally the communication speed will be the dominant factor. The speed of communication with the monitor depends on both the speed of the network connection and the implementation of the requirements instance model which is used. The database implementation of the requirements instance model was found to result in a considerably larger communication overhead due to the less efficient communication protocol.

### 7.1.2 Architecture

A general architecture for monitoring was developed, consisting of an instrumented system, a requirements instance model and monitors. The core of the architecture is the requirements instance model which stores an instantiation of the requirements model which corresponds to the state of the monitored system. The instrumentation code generates messages which inform the requirements instance model of changes which it should make to the model so that the model will reflect changes in the monitored system. Monitors use the requirements instance model to detect failures of hard goals and to evaluate soft goal metrics. The monitoring system provides two alternate implementations of the requirements instance model, one implemented using a database and the other using Java classes.

The database implementation represents KAOS relationships, entities and attributes as tables. The instrumentation code communicates directly with the database to update the model to reflect the state of the system. The monitor periodically checks the database to discover changes which have occurred in the model. This implementation is made more complex because the monitor is not informed of changes to the model as soon as they occur. It is therefore necessary to store each change which has occurred since the last time the database checked the model. So for example, when a relationship is destroyed, it is initially only marked as deleted and is not removed from the model until the monitor checks the database.

The other implementation of the requirements instance model uses Java classes which represent instances of KAOS relationships and entities. This approach allows the monitor to be immediately informed whenever the model changes. This implementation takes into account problems which might occur due to delays in communicating instrumentation messages in a distributed system, as discussed in section 4.1.4. This implementation attempts to sort messages into the correct order and ensure that they

are processed in that order although this still requires that the clocks of any distributed components are synchronised with a sufficiently high degree of accuracy.

These two implementations were evaluated using the case study. The database implementation was found to impose a significantly higher performance overhead on the monitored system than the implementation using Java classes. This appeared to be due to the less efficient communication between the monitor server and the target system when using the database implementation. However, the case study involved large number of changes to the requirements instance model over a short period of time. Other applications may involve far fewer changes, making the database implementation a viable option. The database implementation may have other benefits such as greater reliability but these have not been assessed. Currently, the most important benefit of the database implementation is that goal instance metrics are only supported on this implementation, although this is not an inherent limitation of the Java object implementation.

### 7.1.3 Monitoring Soft Goals

One of the objectives of the monitoring framework is to use monitoring to help evaluate whether soft goals are satisfied by a system. This was done by formally defining metrics which are indicative of the satisfaction of soft goals. By presenting the values of these metrics to the users of the monitoring system, they can decide whether the system has failed to satisfy the soft goals associated with the metrics. This allows users to make complex decisions about soft goals, using information which is not available to the monitoring system.

A language was developed for specifying soft goal metrics which is built on top of the KAOS goal-oriented requirements engineering language. This language allows the definition of two types of metric. Goal instance metrics are associated with a particular instance of a hard goal and are defined using the KAOS object model of the monitored system. Goal aggregate metrics are associated with a set of hard goal instances and are defined using the KAOS goal model and goal instance metrics which are defined for those goals. By combining these two type of specification, it is made easy to define complex metrics.

This approach was successfully used to specify soft goal metrics in the case study using the NGDS system. The case study established that the language for specifying soft goal metrics was able to formally specify the necessary soft goals and that this specification was reasonably clear and concise. The amount of code needed to instrument the system so that the soft goal metrics could be monitored was also found to be reasonable.

### 7.1.4 Display of Monitoring Results

The monitoring framework includes graphical displays which show the results of monitoring at run time. Hard goal failures are displayed using a goal tree which shows details about individual failures. The displays for soft goal metrics are configured by the developer. A number of default gauges are provided which can be customised as necessary or the developer can use the interface provided to write new gauges.

The framework provides a lot of flexibility in display of results in part because it is still not clear what type of feedback is useful to users and developers of a system in terms of run-time monitoring. The displays should, where possible, assist the developers in modifying the system so that failures will not occur in future. The final stage of

communicating monitoring information back to the developer is a difficult one and the problem requires further work.

### 7.1.5 Comparison with Related Work

The monitoring framework described in this thesis uses the KAOS goal-oriented requirements engineering approach which provides formal specification of goals in temporal logic. The use of a requirements engineering approach which incorporates formal specification is an advantage of this work. Other related work such as [Sankar 93, Chodrow 91, Kim 01, Havelund 01, Gates 01] do not integrate so well with an existing requirements engineering approach. A possible disadvantage of the approach in this thesis is that formal specifications chosen specifically for monitoring may be easier to use than KAOS, which is designed for specifying and analysing requirements, but it is advantageous to be able to use the same requirements specifications for both requirements analysis and monitoring.

An area of particular focus in this work has been to support instrumentation. This issue was not covered in any detail in [Fickas 95, Feather 98], which also use KAOS to specify requirements to be monitored. Support for instrumentation is not necessary in all cases, such as where monitoring is implemented using an object-oriented operating system[Dasgupta 86, Snodgrass 88], law-governed architecture[Minsky 96] or for monitoring web services[Robinson 03, Mahbub 04, Lazovik 04]. There is, however, value in a general approach to instrumentation which is not tied to a particular architecture, which is provided by the work in this thesis.

Some other approaches do support instrumentation to some extent. Work such as[Chodrow 91, Chen 03, Sankar 93] use source code annotations. This results in instrumentation code which is tangled with system code. Other approaches use source code modification[Liao 92] or byte code modification[Havelund 01] to separate instrumentation code from system code. None of these approaches are coupled to a comprehensive approach to requirements engineering, such as KAOS, as in this thesis. These approaches also lack the expressive power of AspectJ as an instrumentation mechanism.

The most interesting other piece of work on instrumentation for monitoring is the Java-MaC system[Kim 01]. In Java-MaC, a formal requirements specification is related to the implementation level through an intermediate language which defines conditions and events which depend on the state of the execution of the monitored system. This intermediate language allows the formal requirements specification to remain independent of the implementation details. The intermediate language is also used to automatically generate instrumentation by byte code modification. The work in this thesis uses a similar idea in using AspectJ as an intermediate language to translate implementation events to requirements level events. The relative disadvantages of Java-MaC are that the intermediate language is limited in its expressiveness and that the requirements specification is also fairly limited in scope.

In comparison with other work in this area, a weakness of the approach in this thesis may be the comparative complexity of the instrumentation process and the instrumentation code itself. This also makes it harder to ensure that instrumentation code is itself correct. In part this is because our requirements specification is richer and more complex than other approaches. In addition, keeping the instrumentation code separate from system code increases the complexity of the instrumentation code, as it is necessary to identify where in the system instrumentation code should be executed. This

extra complexity is necessary to support the separation of system and instrumentation code and to support the rich KAOS requirements specification.

Another important contribution made by this work is in monitoring of soft goals and in specifying metrics to allow monitoring of soft goals. This is a topic that has received relatively little attention. The work in this thesis builds on ideas from two sources. In [Robinson 03], aggregate monitors are included which detect situations such as repeated failures of a goal. In [Letier 04], the KAOS methodology is extended to more precisely specify soft goals. This is done not for monitoring but to improve analysis of soft goals at the requirements stage. Quality variables are defined, using natural language, and combined using objective functions which indicate quantities to be maximised or minimised. The work on monitoring soft goals in this thesis combines these two ideas so that aggregate metrics can be written which use goal instance metrics. The idea of quality variables is formalised, resulting in goal instance metrics which are defined using the KAOS object model.

In this thesis, the problem of displaying the results of monitoring to users of the monitoring framework was discussed. This is something which has had little attention. One place where this problem is discussed is in the ReqMon system[Robinson 03], where the idea of using visual gauges to display monitor output to human users is suggested. In this thesis, display of monitoring results for soft goals is handled using user defined gauges which can display soft goal metrics in a variety of ways. A few pre-defined gauges were developed and users can code additional gauges in Java. This leaves the user to determine how best to display results. What types of gauges are most useful remains an open question in this area which has not been solved by this thesis and is a possible direction for future work.

## 7.2 Critical Evaluation

The aims of the monitoring framework have generally been satisfied in that the features described previously have been successfully implemented and the system performs reasonably well. There are two areas of concern which may limit the effectiveness of the monitoring framework which are described here.

### 7.2.1 Correctness of Instrumentation

One problem with the approach described in this thesis is that it is difficult to guarantee correctness of the results which are produced by the monitoring system. The source of this problem is the actual instrumentation code. The output produced by the monitor server will only be as good as the information it is provided by the instrumentation code. If the mapping from requirements to implementation, whether explicit in the mapping language or implicit in AspectJ instrumentation code, is incorrect then the results produced by the monitor server may also be incorrect.

In practice, this process of mapping from requirements to implementation was found to be by far the hardest task in monitoring a system. It was also necessary to experiment to a degree before the correct mapping was found. This is likely to leave the developer with reduced confidence in the results produced by the monitor server as the possibility for errors is clear. Good documentation of the software engineering process, including requirements, architecture and design were found to be extremely helpful in instrumenting both the Limewire and the NGDS systems. Good tools were also helpful in developing instrumentation code, such as the ability of some software development environments to find all uses of a particular method. Perhaps tools specifically designed

to assist in the development of instrumentation code would give greater confidence in the correctness of instrumentation code as well as speeding up development.

Instrumentation can be incorrect due to it providing incorrect information or because instrumentation does not exist for some event where it should exist. The first problem might be dealt with by testing instrumentation code, in a similar way to unit tests. This would be done by executing the part of the target system where an instrumentation message should be generated and checking that the generated message matches the expected message. Some additional support from the monitoring framework would most likely be useful in implementing this. A particular problem is generating a valid execution of a particular part of the target system so that instrumentation can be checked. This would likely be much easier if unit tests already exist as the instrumentation test code could make use of these.

The problem of missing instrumentation is a challenging one. It is very difficult to prove that the existing instrumentation is complete. Testing is a possibility here although it would be necessary to work with more complex tests than unit tests. It might be possible to write a test for which some sequence of instrumentation messages is expected and compare this with the instrumentation messages which are actually generated. Missing instrumentation messages from the sequence could then be detected.

Another problem with instrumentation code is the danger that it may functionally alter the behaviour of the monitored system. The most serious case where this could occur is if instrumentation code calls methods which have side effects beyond returning a value, such as changing the state of an object or causing output operations to take place. It may be possible to perform some analysis of the instrumentation code which would help detect whether methods with side effects have been called or to detect such problems at run time during testing.

### 7.2.2 Scalability

No analysis of the scalability properties of the monitor server has been performed. As the monitoring framework uses a centralised server to gather information and evaluate satisfaction of requirements, scalability will probably be a concern if large distributed systems are monitored. This is not a problem which will exist for all systems which it might be desirable to monitor so the monitoring framework is still useful without a solution to this problem. In the NGDS case study, scalability is not a problem as only a single machine is monitored. Monitoring a Gnutella network, as in the Limewire example, makes scalability an issue. If a full scale Gnutella network were monitored, rather than a network of a few clients, then scalability would likely become a major issue.

One approach to solving the scalability problem might be to have a separate monitor server for each agent in the system which monitors the goals for which that agent is responsible. This does create greater complexity in instrumenting the system as it becomes necessary to determine which monitor server to send each message to. It may also be necessary to send some messages to multiple servers. It might be possible to reduce communication overhead by using existing communications to piggy-back information, as in [Sen 04].

### 7.2.3 Usefulness of Monitoring

The aim of this work is that developers of a system should be able to use the monitoring system to detect failures in the system caused by changes in the environment. This

alerts the developers of the need to modify the system so that it can continue to satisfy its requirements. The information provided by the monitoring system should assist developers in doing this.

The case study has shown that requirements monitoring can detect failures due to changes in the environment and that it is possible to use additional metrics, along with knowledge of the soft goal model, to determine the cause of these failures.

Once the cause of a failure is known, it becomes possible to consider possible ways of rectifying the problem. To prevent failures from occurring, the system can be modified so that it will satisfy requirements in the new environment or the environment can be modified so that it again conforms to the assumptions made by the system. In general, the best solution is likely to be to change the system as computer systems are normally easier to modify than the environment in which they operate.

It is left to the developers or administrators of the monitored system to determine what changes are necessary and how to implement them using their existing knowledge of the system and the problem domain. Knowledge of exactly what the failures are and why they have occurred should make this process easier.

It is certainly useful to alert developers or administrators of the failure of a system to satisfy requirements as it may not be obvious that such a failure has occurred. In particular, monitoring soft goals provides information which may not otherwise be easily available as it can tie together disparate information from different parts of the system. For example, individual users of a system may experience occasional performance problems with certain tasks but only by collating results from many users can it be determined which tasks have consistent performance problems which need to be addressed.

Making changes to a system to prevent failure due to changes in the environment requires a large amount of knowledge. Much of this knowledge is specific to the problem domain. It may not be explicitly represented in any of the design artifacts but may be general knowledge about the problem domain which should be known to developers and administrators of the monitored system.

Modifying a system at the implementation level is also likely to require information at a lower level than the requirements level information provided by the monitoring framework. Nonetheless, requirements level information is not useless in this context. Perhaps using requirements monitoring to discover when changes are necessary and then relating those requirements to architecture and design would help in this situation. Such links are already created to some extent between requirements and implementation as part of the instrumentation process so perhaps this information could be used in presenting the results of monitoring to the user of the monitoring framework.

This thesis has provided a demonstration of the feasibility of run-time requirements monitoring as a strategy for detecting failure of a system to satisfy requirements due to changes in the environment in which the system operates. There is still some way to go in demonstrating that requirements monitoring provides concrete benefits in the real world which justify the costs of implementation.

Demonstrating the usefulness of monitoring in the real world is a difficult problem. Doing so would involve implementing monitoring for a real system in its operating environment and using monitoring to detect failures which occur. Depending on the system being monitored, it could take a significant time before failures emerged, especially failures due to changes in the environment. Having detected failures, it would

then be necessary to take action to rectify them, making use of the information provided by monitoring where possible. Finally, it would be necessary to compare to benefits associated with monitoring to the costs, such as performance overhead and work involved in implementation, to determine in it is of overall benefit. Ideally, this process should be repeated for a number of different systems.

Realistically, before such a complex study is possible, further development of runtime requirements monitoring is probably necessary. Further case studies can provide evidence of usefulness and failures due to changes in the environment can be simulated. It might then be possible to examine how a system could be changed to prevent failures occurring again and what information would help developers or administrators carry out those changes.

## 7.3 Open Questions and Future Work

### 7.3.1 Improvements to Monitoring Framework

The monitoring framework could be extended in various ways to provide more functionality. An obvious extension would be to allow formalisms other than KAOS to be used for requirements specification. This would be relatively easy to achieve as long as an object model can be defined for the formalism. These models may be explicitly defined as part of the formal language or implicitly defined, in which case the model must be explicitly defined before the language can be used within the monitoring framework.

The mapping language was found to provide savings in terms of lines of code when it was able to be applied but these situations were limited. It is also beneficial because it makes the relationship between KAOS relationships and entities and the Java implementation explicit. The language could be further developed to provide additional types of mappings, hopefully extending its usefulness. It would also be desirable to include some capability within the language for caching changes to the model and then communicating them at an appropriate time as this was found to be a useful approach in the case study.

### 7.3.2 Architecture Specific Monitoring

The monitoring framework described in this thesis has problems guaranteeing correctness of instrumentation. Writing instrumentation code can also be time consuming. A different approach which overcomes these problems is to provide monitoring for a limited set of systems which share a common architecture. Instrumentation can then be defined at the architecture level using architectural knowledge to inform the placement of instrumentation at the interface between components in the system. Software is increasingly being written using middleware to help implement architectures and instrumentation code could be included in middleware to allow systems built using the middleware to be monitored without writing additional instrumentation code for each new system. The downside of this approach is that it only works for systems which are built using a particular middleware.

An example of this type of approach is monitoring web services. Because web services are specified at a high level and then implemented at a high level by an execution engine, instrumentation can be built into the execution engine in a way which can be used to monitor any web service which runs on the execution engine.

Further research in including requirements monitoring capabilities as part of middleware is a possible direction for future work.

### 7.3.3 Utilisation of Monitoring Results

There is still uncertainty as to what the best way to deal with the results of run-time requirements monitoring. The monitoring results can either be presented to the user, as in the monitoring framework in this thesis, or they can be used to automatically modify the target system to rectify problems or prevent them occurring in future.

The user of the monitoring framework may be able to deal with some problems by manually altering settings in the target system or by otherwise intervening to fix problems. Other problems may require rewriting parts of the system to prevent problems from occurring in the future. It is an open question how best to communicate monitoring results to allow the user to perform these tasks. The graphical gauges which form part of the monitoring framework are intended to provide this information although the it largely left to users of the framework to create their own gauges which are most suitable for a particular task. A possible direction for further work is to try and determine what sort of information is needed to modify a system and how to provide it through monitoring.

Using automatic modification of the target system is another possible approach. Combining this approach with run-time requirements monitoring is potentially very powerful but the target system has to be sufficiently flexible that it can be automatically modified to respond to the environment. This flexibility is not necessarily present in existing systems and it may not be easy to implement.

### 7.3.4 Mapping Requirements to Implementation

The mapping of requirements to the implementation of those requirements is of great importance to run-time requirements monitoring. The accuracy of the monitor is dependent on the quality of the instrumentation from which it obtains its information. The quality of the instrumentation is itself dependent on the quality of the requirements to implementation mapping, whether that mapping is used to automatically generate implementation or is used by the instrumentation developer to manually build instrumentation.

The approach taken in this thesis was to allow the instrumentation developer to retrospectively develop the mapping between requirements and implementation. This is done manually by the instrumentation developer. The mapping language and helper classes used when developing directly in AspectJ provide some assistance in formally defining these mappings.

A possible approach to this problem is to ensure traceability links between requirements and implementation are maintained during development of the target system. Traceability is itself a subject of research and the problem is far from solved[Gotel 94]. The quality of traceability from requirements to implementation is therefore likely to vary from system to system and the presence of such links cannot be assumed.

More extensive work on manually mapping implementation to requirements is a possible area for future work. An obvious start would be to create a tool to assist in writing the mapping. A good approach might be to automatically extract a UML / XMI model of the implementation and present it to the instrumentation developer. The other necessary input would be the KAOS requirements specification. The tool could then provide a graphical interface for relating elements of the requirements specification to the implementation. Further development might involve trying to automatically detect mapping between requirements and implementation, for example using similarity of name of elements in requirements and implementation. As such automatic mappings

will never be completely reliable, the graphical interface would still be necessary to check the automatic mappings and correct them as necessary. This leads to the possibility of an iterative process where the automatic mapper uses the developers correction to further improve the quality of the automatic mappings.

An approach to discovering mappings between requirements and implementation which might be useful is to make use of the relationships in the KAOS object model. By looking at relationships which exist in the KAOS object model, it should be possible to look for similar relationships which exist in the implementation. These relationships are likely to be obscured by implementation details (such as through the use of list and hash table objects in Java to model one to many relationships) but it might still be possible to match patterns of relationships in the KAOS object model to the implementation code.

# Appendix A

# Limewire Formal Specifications

**Achieve[Search For File]**
$\forall$ c1:Client, c2:Client, q:Query, f:File, fd:FileDescriptor
QueryRequested(q, c1) $\wedge$ InCommunicationRange(c1, c2) $\wedge$ SharingFile(c2, f) $\wedge$
MatchesFile(q, f) $\Rightarrow$ $\Diamond$ DisplayingResult(c1, fd) $\wedge$ ReferencesFile(f, fd)

**Achieve[Communicate Query]**
$\forall$ c1:Client, c2:Client, q:Query
QueryRequested(q, c1) $\wedge$ InCommunicationRange(c1, c2) $\Rightarrow$ $\Diamond$ ReceivedQuery(c2, c1, q)

**Achieve[Broadcast Query]**
$\forall$ c1:Client, c2:Client, q:Query
QueryRequested(q, c1) $\wedge$ Connected(c1, c2) $\Rightarrow$ $\Diamond$ SentQuery(c1, c2, q)

**Achieve[Transmit Query]**
$\forall$ c1:Client, c2:Client, q:Query
SentQuery(c1, c2, q) $\Rightarrow$ $\Diamond$ ReceivedQuery(c2, c1, q)

**Achieve[Forward Query]**
$\forall$ c1:Client, c2:Client, c3:Client, q:Query
ReceivedQuery(c1, c2, q) $\wedge$ Conencted(c1, c3) $\wedge$ c1$\neq$c2 $\wedge$
$\neg$ HopsLimitReached(q, c1) $\Rightarrow$ $\Diamond$ SentQuery(c1, c3, q)

**Achieve[Communicate Query Reply]**
$\forall$ c1:Client, c2:Client, c3:Client, c4:Client, q:Query, fd:FileDescriptor, f:File
QueryRequested(q, c1) $\wedge$ ReceivedQuery(c2, c3, q) $\wedge$
SharingFile(c2, f) $\wedge$ MatchesFile(q, f) $\Rightarrow$
$\Diamond$ ReceivedQueryReply(c1, c4, fd) $\wedge$ ReferencesFile(f, fd)

**Achieve[Respond To Query]**
$\forall$ c1:Client, c2:Client, q:Query, fd:FileDescriptor, f:File
ReceivedQuery(c1, c2, q) $\wedge$ SharingFile(c1, f) $\wedge$ MatchesFile(q, f) $\Rightarrow$
$\Diamond$ SentQueryReply(c1, c2, fd) $\wedge$ ReferencesFile(f, fd)

**Achieve[Tranmit Query Reply]**
$\forall$ c1:Client, c2:Client, fd:FileDescriptor
SentQueryReply(c1, c2, fd) $\Rightarrow$ $\Diamond$ ReceivedQueryReply(c2, c1, fd)

**Achieve[Forward Query Reply]**
∀ c1:Client, c2:Client, c3:Client, q:Query, fd:FileDescriptor
ReceivedQueryReply(c1, c2, fd) ∧ ReceivedQuery(c1, c3, q) ∧ ResponseTo(fd, q) ⇒
◊ SentQueryReply(c1, c3, fd)

**Achieve[Send Query Reply]**
∀ c1:Client, c2:Client, c3:Client, q:Query, fd:FileDescriptor
ReceivedQueryReply(c1, c2, fd) ∧ ReceivedQuery(c1, c3, q) ∧
Connected(c1, c3) ∧ ResponseTo(fd, q) ⇒ ◊ SentQueryReply(c1, c3, fd)

**Maintain[Query Source Connection]**
∀ c1:Client, c2:Client, c3:Client, q:Query, fd:FileDescriptor
ReceivedQueryReply(c1, c2, fd) ∧ ReceivedQuery(c1, c3, q) ∧
ResponseTo(fd, q) ⇒ ◊ Connected(c1, c3)

**Achieve[Display Search Result]**
∀ c1:Client, c2:Client, fd:FileDescriptor
ReceviedQueryReply(c1, c2, fd) ∧ QueryRequested(q, c1) ⇒
◊ DisplayingResult(c1, fd)

**Achieve[Download File]**
∀ c:Client, f:File, fd:FileDescriptor
RequestingFile(c, fd) ⇒ ◊ SavedFile(c, f) ∧ f.name = fd.name

**Achieve[Send File Request]**
∀ c1:Client, c2:Client, fd:FileDescriptor
RequestingFile(c1, fd) ∧ FileStoredBy(fd, c2) ⇒ ◊ SentFileRequest(c1, c2, fd)

**Achieve[Transmit File Request]**
∀ c1:Client, c2:Client, fd:FileDescriptor
SentFileRequest(c1, c2, fd) ⇒ ◊ ReceviedFileRequest(c2, c1, fd)

**Achieve[Upload File]**
∀ c1:Client, c2:Client, fd:FileDescriptor, f:File
ReceivedFileRequest(c1, c2, fd) ⇒ ◊ SentFile(c1, c2, f) ∧ f.name = fd.name

**Achieve[Transmit File]**
∀ c1:Client, c2:Client, fd:FileDescriptor, f:File
SentFile(c1, c2, f) ⇒ ◊ ReceivedFile(c2, c1, f)

**Achieve[Store File]**
∀ c1:Client, c2:Client, f:File
ReceivedFile(c1, c2, f) ⇒ ◊ SavedFile(c1, f)

## Appendix B

# Mapping Language DTD

```
<!ELEMENT Mapping (Relationship|Object)*>
<!ATTLIST Mapping name CDATA #REQUIRED>

<!ELEMENT Relationship (Role|Transition|StateMapping)*>
<!ATTLIST Relationship name CDATA #REQUIRED>

<!ELEMENT Transition (Role)*>
<!ATTLIST Transition
    position (before|after|around) #REQUIRED
    location CDATA #REQUIRED
    value CDATA #IMPLIED>

<!ELEMENT StateMapping (StateTransition)*>
<!ATTLIST StateMapping class CDATA #REQUIRED
                       attribute CDATA #REQUIRED>

<!ELEMENT State (Role)*>
<!ATTLIST State value (true|false) #REQUIRED
                valueObject CDATA #IMPLIED
                valueConst CDATA #IMPLIED>

<!ELEMENT Role EMPTY>
<!ATTLIST Role name CDATA #REQUIRED
               type CDATA #REQUIRED
               context CDATA #IMPLIED
               roleObject CDATA #IMPLIED
               objectID CDATA #IMPLIED>

<!ELEMENT Object (ObjectID, (Attribute)*)>
<!ATTLIST Object name CDATA #REQUIRED
                 objectElement CDATA #REQUIRED>

<!ELEMENT ObjectID EMPTY>
<!ATTLIST ObjectId object CDATA #REQUIRED>
```

```
<!ELEMENT Attribute EMPTY>
<!ATTLIST Attribute name CDATA #REQUIRED
                    attributeObject CDATA #REQUIRED>
```

## Appendix C

# Goal Instance Metric Query Generation

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    version="2.0">

  <xsl:template match="/">
    <ValueSet>
      <xsl:apply-templates/>
    </ValueSet>
  </xsl:template>

  <xsl:template match="Value">
    <Value label="@label" type="@type"
           trigger="@trigger" goal="ancestor::Goal/@name">
    <xsl:variable name="bound-labels" select=
        "(../(Antecedent|Consequent)//Relationship/Variable
        except .//Conditions//Relationship/Variable)/@label"/>

    <xsl:for-each-group
      select=".//(Variable|Attribute)[@label = $bound-labels]"
      group-by=".">
      <BoundLabel><xsl:value-of select="@label"/></BoundLabel>
    </xsl:for-each-group>

    <xsl:variable name="attr-type">
      <xsl:choose>
        <xsl:when test="@type='float'">'float'</xsl:when>
        <xsl:when test="@type='int'">'int'</xsl:when>
        <xsl:when test="@type='string'">'string'</xsl:when>
        <xsl:when test="@type='boolean'">'boolean'</xsl:when>
      </xsl:choose>
    </xsl:variable>

    <ValueQuery>SELECT
```

```
<xsl:choose>
  <xsl:when test=".//Function[@name='avg']">
      AVG(value)
  </xsl:when>
  <xsl:when test=".//Function[@name='std']">
      STD(value)
  </xsl:when>
  <xsl:when test=".//Function[@name='max']">
      MAX(value)
  </xsl:when>
  <xsl:when test=".//Function[@name='min']">
      MIN(value)
  </xsl:when>
  <xsl:when test=".//Function[@name='sum']">
      SUM(value)
  </xsl:when>
  <xsl:otherwise>value</xsl:otherwise>
</xsl:choose>
AS val FROM attribute, entity

WHERE name='<xsl:value-of
             select=".//Attribute/@attribute"/>'
AND new!='TRUE'
AND entity.id = attribute.entity_id AND entity.type=
'<xsl:value-of select=".//Attribute/@type"/>'

<xsl:variable name="attribute-label"
              select=".//Attribute/@label"/>

<xsl:for-each select=".//Conditions//Relationship">
  AND entity_id IN
    (SELECT entity_id FROM role_entity, relationship
    WHERE relationship.type=
    '<xsl:value-of select="@name"/>'
    AND relationship.id = role_entity.relationship_id
    AND role='<xsl:value-of select=
        "./Variable[@label = $attribute-label]/@role"/>'

    <xsl:for-each
             select="./Variable[@label = $bound-labels]">
      AND relationship.id IN
      (SELECT relationship_id FROM role_entity
      WHERE role='<xsl:value-of select="@role"/>'
      AND entity_id=?)
    </xsl:for-each>
    )

</xsl:for-each>
<xsl:for-each select=".//Conditions//Equals">
  AND entity_id IN (SELECT entity_id FROM attribute WHERE
```

```
      name='<xsl:value-of select="./Attribute/@attribute"/>'
      AND value='<xsl:value-of select="./String/text()"/>')
    </xsl:for-each>
    </ValueQuery>
    </Value>
  </xsl:template>
</xsl:stylesheet>
```

# Bibliography


[Andrews 98] James H. Andrews. *Testing Using Log File Analysis: Tools, Methods, and Issues.* In the 13th IEEE International Conference on Automated Software Engineering, pages 157–166, 1998.

[Baker 02] Jason Baker & Wilson Hsieh. *Runtime Aspect Weaving Through Metaprogramming.* In the 1st International Conference on Aspect-Oriented Software Development, pages 86–95. ACM Press, 2002.

[Bates 83] Peter Bates & Jack C. Wileden. *An Approach to High-Level Debugging of Distributed Systems.* SIGSOFT Software Engineering Notes, vol. 8, no. 4, pages 107–111, 1983.

[Bennett 00] Keith Bennett & Vaclav Rajlich. *Software Maintenance and Evolution: A Roadmap.* In Anthony Finkelstein, editor, The Future of Softrware Engineering, pages 73–87. ACM Press, 2000.

[Capra 03] Licia Capra, Wolfgang Emmerich & Cecilia Mascolo. *CARISMA: Context-Aware Reflective Middleware System for Mobile Applications.* IEEE Transactions on Software Engineering, vol. 29, no. 10, pages 929–945, 2003.

[Castro 02] Jaelson Castro, Manuel Kolp & John Mylopoulos. *Towards Requirements-Driven Information Systems Engineering: The Tropos Project.* Information Systems, vol. 27, no. 6, pages 365–389, 2002.

[Chen 03] Feng Chen & Grigore Rosu. *Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation.* Electronic Notes in Theoretical Computer Science, vol. 89, no. 2, 2003.

[Chodrow 91] Sarah Chodrow, Farnam Jahanian & Marc Donner. *Run-Time Monitoring of Real-Time Systems.* In the IEEE Real-Time Systems Symposium, pages 74–83, 1991.

[Dardenne 93] Anne Dardenne, Axel van Lamsweerde & Stephen Fickas. *Goal-Directed Requirements Acquisition.* Science of Computer Programming, vol. 20, no. 1-2, pages 3–50, 1993.

[Dasgupta 86] Partha Dasgupta. *A Probe-Based Monitoring Scheme for an Object-Oriented Distributed Operating System.* In the International Conference on Object-Oriented Programming Systems, Languages and Applications, pages 57–66. ACM Press, 1986.

[Drusinsky 00] Doron Drusinsky. *The Temporal Rover and the ATG Rover.* In the 7th International SPIN Workshop on SPIN Model Checking and Software Verification, pages 323–330, 2000.

[Feather 98] Martin S. Feather, Stephen Fickas, Axel van Lamsweerde & C. Ponsard. *Reconciling System Requirements and Runtime Behavior.* In the 9th International Workshop on Software Specification and Design, pages 50–59, 1998.

[Fickas 95] Stephen Fickas & Martin S. Feather. *Requirements monitoring in dynamic environments.* In the 2nd IEEE International Symposium on Requirements Engineering, pages 140–147, 1995.

[Finkelstein 00] Anthony Finkelstein & Jeff Kramer. *Software Engineering: A Raodmap.* In Anthony Finkelstein, editor, The Future of Softrware Engineering, pages 3–22. ACM Press, 2000.

[Gates 01] Ann Q. Gates, Steve Roach, Oscar Mondragon & Nelly Delgado. *DynaMICs: Comprehensive Support for Run-Time Monitoring.* In Klaus Havelund & Grigore Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 55. Elsevier Science Publishers, 2001.

[Gotel 94] Orlena Gotel & Anthony Finkelstein. *An analysis of the requirements traceability problem.* In the 1st International Conference on Requirements Engineering, pages 94–101, 1994.

[Havelund 01] Klaus Havelund & Grigore Rosu. *Monitoring Java Programs with Java PathExplorer.* Electronic Notes in Theoretical Computer Science, vol. 55, no. 2, 2001.

[Jackson 95] Michael Jackson. *The World and the Machine.* In the International Conference on Software Engineering, pages 283–292, 1995.

[Kaelbling 90] Michael J. Kaelbling & David M. Ogle. *Minimizing monitoring costs: choosing between tracing and sampling.* In the Hawaii International Conference on System Sciences, volume 1, pages 314–320. IEEE Press, 1990.

[Kiczales 97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier & John Irwin. *Aspect-Oriented Programming.* In Mehmet Akşit & Satoshi Matsuoka, editors, the 11th European Conference on Object-Oriented Programming, volume 1241, pages 220–242, New York, NY, 1997. Springer-Verlag.

[Kiczales 01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm & William G. Griswold. *An Overview of AspectJ.* Lecture Notes in Computer Science, vol. 2072, pages 327–355, 2001.

[Kim 01] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky & Mahesh Viswanathan. *Java-MaC: a Run-time Assurance Tool for Java Programs.* In Klaus Havelund & Grigore Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 55. Elsevier Science Publishers, 2001.

[Kunz 97] T. Kunz, J. P. Black, D. J. Taylor & T. Basten. *Poet: Target-System Independent Visualizations of Complex Distributed-Application Executions.* The Computer Journal, vol. 40, no. 8, pages 499–512, 1997.

[Lamport 78] Leslie Lamport. *Time, clocks, and the ordering of events in a distributed system.* CACM, vol. 21, no. 7, pages 558–565, 1978.

[Lazovik 04] Alexander Lazovik, Marco Aiello & Mike Papazoglou. *Associating assertions with business processes and monitoring their execution.* In the International Conference on Service Oriented Computing, pages 94–104. ACM Press, 2004.

[Letier 01] Emmanuel Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering.* PhD thesis, Université catholique de Louvain, 2001.

[Letier 04] Emmanuel Letier & Axel van Lamsweerde. *Reasoning about partial goal satisfaction for requirements and design engineering.* In ACM SIGSOFT Foundations of Softawre Engineering 12, pages 53–62, 2004.

[Letier 05] Emmanuel Letier, Jeff Kramer, Jeff Magee & Sebastian Uchitel. *Fluent temporal logic for discrete-time event-based models.* In ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pages 70–79. ACM Press, 2005.

[Liao 92] Yingsha Liao & Donald Cohen. *A specificational approach to high level program monitoring and measuring.* IEEE Transactions on Software Engineering, vol. 18, no. 11, pages 969–978, 1992.

[Ludwig 04] Heiko Ludwig, Asit Dan & Robert Kearney. *Cremona: an architecture and library for creation and monitoring of WS-agreents.* In the International Conference on Service Oriented Computing, pages 65–74. ACM Press, 2004.

[Mahbub 04] Khaled Mahbub & George Spanoudakis. *A framework for requirents monitoring of service based systems.* In the International Conference on Service Oriented Computing, pages 84–93. ACM Press, 2004.

[Mansouri-Samani 97] Masoud Mansouri-Samani & Morris Sloman. *GEM: a generalized event monitoring language for distributed systems.* Distributed Systems Engineering, vol. 4, no. 2, pages 96–108, 1997.

[Mills 91] David L. Mills. *Internet time synchronization: the network time protocol.* IEEE Transactions on Communications, vol. 39, no. 10, pages 1482–1493, 1991.

[Minsky 96] Naftaly H. Minsky. *Independent on-line monitoring of evolving systems.* In the 18th International Conference on Software Engineering, pages 134–143. IEEE Press, 1996.

[Mylopoulos 92] John Mylopoulos, Lawrence Chung & Brian A. Nixon. *Representing and Using Nonfunctional Requirements: A Process-Oriented Approach.* Software Engineering, vol. 18, no. 6, pages 483–497, 1992.

[Mylopoulos 99] John Mylopoulos, Lawrence Chung & Eric Yu. *From object-oriented to goal-oriented requirements analysis.* Communications of the ACM, vol. 42, no. 1, pages 31–37, 1999.

[Pawlak 01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien & Gérard Florin. *JAC: A Flexible Solution for Aspect-Oriented Programming in Java.* In Lecture Notes in Computer Science, Metalevel Architectures and Separation of Crosscutting Concerns : Third International Conference, REFLECTION 2001, volume 2192, pages 1–24. Springer-Verlag Heidelberg, 2001.

[Peters 02] Dennis K. Peters & David Lorge Parnas. *Requirements-based monitors for real-time systems.* IEEE Transactions on Software Engineering, vol. 28, no. 2, pages 146–158, 2002.

[Popovici 03] Andrei Popovici, Gustavo Alonso & Thomas Gross. *Just-in-time aspects: efficient dynamic weaving for Java.* In the 2nd International Conference on Aspect-Oriented Software Development, pages 100–109. ACM Press, 2003.

[Qiao 99] Sanzheng Qiao & Haitong Zhang. *An Automatic Logfile Analyzer for Parallel Programs.* In the International Conference on Parallel and Distributed Processing Techniques and Applications, pages 1371–1376, 1999.

[Robinson 03] William N. Robinson. *Monitoring Web Service Requirements.* In the 11th IEEE International Requirements Engineeging Conference, pages 65–74, 2003.

[Rosenblum 95] David S. Rosenblum. *A practical approach to programming with assertions.* IEEE Transactions on Software Engineering, vol. 21, no. 1, pages 19–31, 1995.

[Sankar 93] Sriram Sankar & Manas Mandal. *Concurrent runtime monitoring of formally specified programs.* IEEE Computer, vol. 26, no. 3, pages 32–41, 1993.

[Schroeder 95] Beth A. Schroeder. *On-line monitoring: a tutorial.* IEEE Computer, vol. 28, no. 6, pages 72–78, 1995.

[Sen 04] Koushik Sen, Abhay Vardhan, Gul Agha & Grigore Rosu. *Efficient Decentralized Monitoring of Safety in Distributed Systems.* In the International Conference on Software Engineering, pages 418–427. IEEE Computer Society, 2004.

[Skene 04a] James Skene & Wolfgang Emmerich. *Generating a Contract Checker for an SLA Language.* In EDOC Workshop on Contract Architectures and Languages. IEEE, 2004.

[Skene 04b] James Skene, D.Davide Lamanna & Wolfgang Emmerich. *Precise ServiceLevel Agreements.* In the International Conference on Software Engineering, pages 179–188, 2004.

[Snodgrass 88] Richard Snodgrass. *A relational approach to monitoring complex systems.* ACM Transactions on Computer Systems, vol. 6, no. 2, pages 157–195, 1988.

[Tarr 99] Peri L. Tarr, Harold Ossher, William H. Harrison & Stanley M. Sutton Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* In the International Conference on Software Engineering, pages 107–119, 1999.

[van Lamsweerde 01] Axel van Lamsweerde. *Goal-Oriented Requirements Engineering: A Guided Tour.* In the 5th IEEE International Symposium on Requirements Engineering, pages 249–262, 2001.

[W3C 05] W3C. *XSL Transformations (XSLT) Version 2.0.* `http://www.w3.org/TR/xslt20/`, 2005.

[Yu 97] Eric Yu. *Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering.* In the 3rd IEEE International Symposium on Requirements Engineering, pages 226–235, 1997.