Monitoring Goals With Aspects

Andrew Dingwall-Smith
Department of Computer Science
University College London
Gower Street
London, WC1E 6BT, UK
A.Dingwall-Smith@cs.ucl.ac.uk

Anthony Finkelstein
Department of Computer Science
University College London
Gower Street
London, WC1E 6BT, UK
A.Finkelstein@cs.ucl.ac.uk

Abstract

Software systems are built based on assumptions about the environment in which they will operate. Change in the environment can therefore result in failures of the system which cannot easily be anticipated.

Runtime requirements monitoring confirms, as a system operates, that it is satisfying the requirements specified for it. Requirements monitoring as part of the normal operation of a system allows unanticipated failures to be identified, and rectified through system evolution.

In this paper we describe our work in run-time monitoring of goal-oriented requirements specifications. This is achieved by instrumenting the monitored system using AspectJ. The instrumentation aspects are automatically generated using a mapping from the requirements specification to the system design. The instrumentation events are used to build an instance of the domain object model of the system, which is used to check for goal failures.

We also discuss monitoring soft goals; goals which do not have formal criteria for satisfaction. We consider what needs to be specified so that these goals can be monitored and a determination of their satisfaction can be made by the stakeholders in the system.

1. Introduction

Regardless of how effective the software engineering process followed in the creation of a system, there is always the possibility that the implemented system may at times fail to satisfy the initial requirements. This is a particular concern if the system is required to operate in dynamic environments. The ability of a system to meet its goals is dependent on both the system itself and the environment in which it operates [6]. 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.

Better design can only go so far in anticipating potential environments before the time invested in the design outweighs the benefits obtained. The solution to this problem is to plan for system evolution, updating the system after deployment as the environment changes.

The aim of this work is to support system evolution by providing facilities for monitoring, at run-time, whether the behaviour of the system satisfies its requirements. This indicates when further evolution of the system is necessary. The monitors should also provide information which helps to identify what changes need to be made to the system so that its behaviour will satisfy the requirements.

The monitoring framework is intended to be as automated as possible. The monitors should be generated from existing artifacts of the software engineering process. Automating the monitoring process as much as possible greatly increases the attractiveness of using requirements monitoring, as it is not part of the normal software development process.

The requirements of the systems we monitor are specified using goal oriented requirements engineering. We describe this type of specification in section 2. Section 3 describes the architecture of the monitoring system. Section 4 describes how the monitored system is instrumented to emit events which can be monitored. Section 5 describes how the instrumentation events are used to monitor for goal failures. Section 6 describes how soft goals can be specified and monitored.

Throughout this paper, we use examples from a case study of a peer-to-peer file sharing program called ‘Limewire’. This is a GPL[1] licensed program, written in Java.

2. Requirements Specification

The monitoring system checks requirements specified as goal oriented requirements. Such a specification expresses what the system should achieve rather than how the system should behave. As such, it is the most appropriate specification to monitor since the concern is not how the system behaves but whether it is behaving in a way that satisfies the requirements, regardless of what the behaviour is.

Our requirements specifications are based on the KAOS approach [2, 9]. In this approach, goals can be formally specified using temporal logic. This formal definition is used for the automatic generation of monitors.

KAOS goals are organised in a goal graph. Each goal is decomposed into sub-goals using a combination of AND and OR refinements. OR refinements give alternative strategies for satisfying higher level goals. Ultimately, the leaf goals are operationalised into requirements which can be satisfied by a single agent. Some goals cannot be satisfied by the system and instead become assumptions which need to be satisfied by agents in the environment.

The goal model is used to generate a domain object model. Every object used in the definition of a goal is part of the object model. There are four types of objects in the KAOS object model. Agents are objects which are capable of performing operations. Entities are objects which are independent of any agent but cannot perform any operations. Relationships are objects which link a number of other objects. Events are objects which only exist in a single state.

2.1. Example Goal Specification

A simple example of a goal graph for downloading a file is shown in figure 1. This is a client-server system in which the files are downloaded over a public network. There are three agents which are relevant here. The client and server agents are part of the system while the network is part of the environment. The top level goal is 'AND' refined into three leaf sub-goals. The goals 'RespondToRequest' and 'StoreFile' are assigned to agents in the system and are therefore requirements. The goal 'ReliableTransmission' is assigned to the 'Network' agent, which is an agent in the environment, and therefore it is an assumption. The formal definitions for these goals are given below:

Achieve[DownloadFile]
Definition When a server receives a request for a file from a client and the file is available to be uploaded, the client should eventually store the file.
Formal Definition
$\forall receivedRequest : ReceivedRequest, file : File$
$receivedRequest.occurs \wedge$
$receivedRequest.fileName = file.name \wedge$
$UploadAvailable(receivedRequest.receivedBy, file) \Rightarrow$
$\Diamond StoredFile(receivedRequest.requestedBy, file)$

Achieve[RespondToRequest]
Definition When a server receives a request for a file from a client and the file is available to be uploaded, the server should eventually send the file to the client.
Formal Definition
$\forall receivedRequest : ReceivedRequest, sentFile : SentFile$
$receivedRequest.occurs \wedge$
$receivedRequest.fileName = file.name \wedge$
$UploadAvailable(receivedRequest.receivedBy, file) \Rightarrow$
$\Diamond sentFile.occurs \wedge$
$sentFile.sentFrom = receivedRequest.requestedFrom \wedge$
$sentFile.sentTo = receivedRequest.requestedBy \wedge$
$sentFile.file = f$

Achieve[ReliableTransmission]
Definition When a file is sent by a server, the client it is sent to should eventually receive the file.
Formal Definition
$\forall sentFile : SentFile, receivedFile : ReceivedFile$
$sentFile.occurs \Rightarrow$
$\Diamond receivedFile.occurs \wedge$
$receivedFile.receivedBy = sentFile.sentTo \wedge$
$receivedFile.file = sentFile.file$

Achieve[StoreFile]
Definition When a client receives a file, it should store the file.
Formal Definition
$\forall receivedFile : ReceivedFile$
$receivedFile.occurs \Rightarrow$
$\Diamond StoredFile(receivedFile.receivedBy, receivedFile.file)$

The requirements in this system can be monitored using information from only a single agent. In fact, this will always be the case, since goals only become requirements when they are able to be assigned to a single agent.

The assumption cannot be monitored using information from a single agent. The 'Network' agent itself cannot provide any information because it is part of the environment and it is only possible to instrument the system. Monitoring this assumption requires information from both the client and the server. The 'Server' agent controls the 'SentFile' event while the 'Client' agent monitors the 'ReceivedFile' event. By combining the data from the client and the server, the assumption can be monitored.

2.2. KAOS Object Model

The object model is derived from the goal model. Every object and attribute used in the formal definition of a goal is present in the object model. The objects used in the goal specifications above are:

Event[ReceivedRequest]
Definition Occurs when a request for a file is received by a client.
Has
requestedFrom : Server

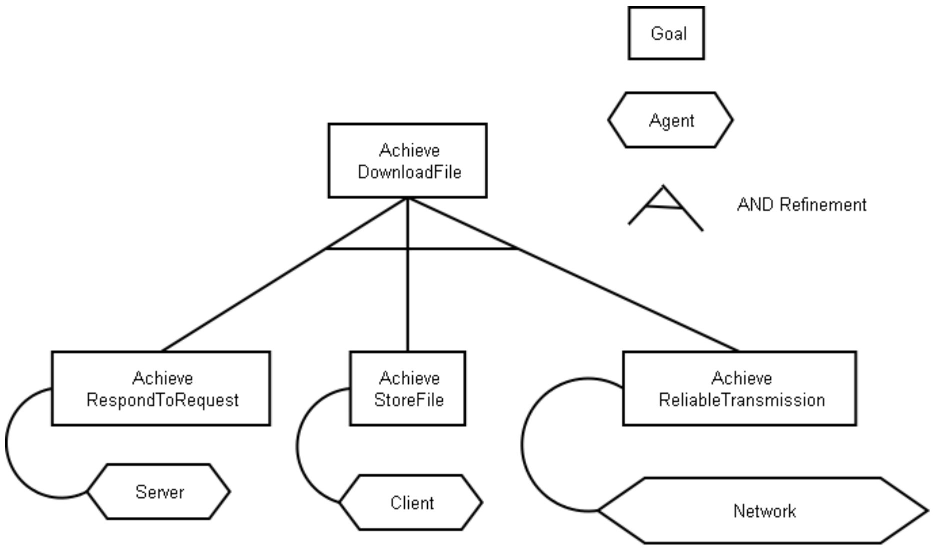

Figure 1: Goal refinement for goal Achieve [Download File]

the server which received the request
requestedBy : Client
the client which sent the request

Event[SentFile]
Definition Occurs when a server has finished sending a file to a client
Has
sentTo : Client
the client the file was sent to
sentBy : Server
the server which sent the file
file : File
the file which was sent

Event[ReceivedFile]
Definition Occurs when a client has received a file
Has
receivedBy : Client
the client which received the file
file : File
the file which was sent

Relationship[UploadAvailable]
Definition A server has a file available and the capacity to upload it
Links
Server **role** sharedBy **card** 0:N
File **role** shared **card** 0:N

Relationship[StoredFile]
Definition A client is storing a file
Links
Client **role** storedBy **card** 0:N
File **role** stores **card** 0:N

Agent[Client]
Definition A client in a client-server system

Agent[Server]
Definition A server in a client-server system
Has
availableFiles : SetOf[File]
the files made available by this server

Entity[File]
Definition A file
Has
name : String
the name of the file

3. Monitoring System Overview

An overview of the monitoring system is shown in figure 2. In the general case, the system is made up of distributed components. These components are instrumented to emits events. The events are described in terms of the domain object model of the system. These events are used to construct an instance of the domain model corresponding to the current state of the system.

The domain model instance is an abstract representation of the state of the system. It expresses the state of the system in the same terms that are used in the requirements specifications. Therefore it should be possible to check the system is satisfying its requirements by analysing the domain model instance.

Goal checkers attach listeners to the parts of the model

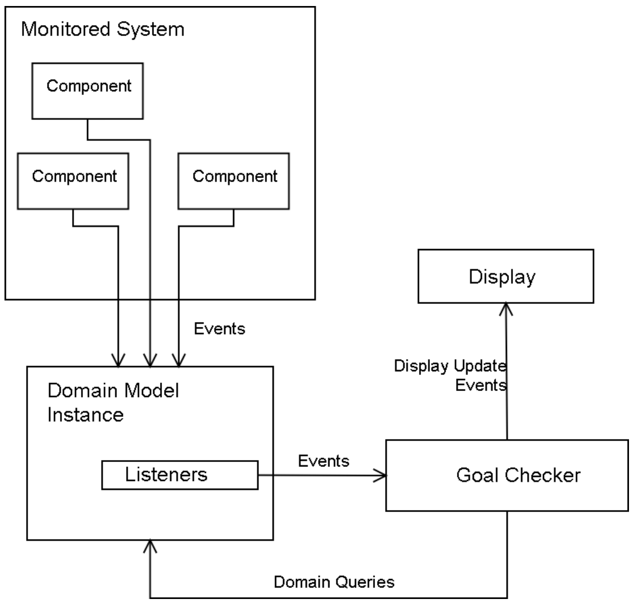

Figure 2: Monitoring System Architecture

that they are interested in. These listeners send events to the goal checkers when those parts of the model are modified by events from the monitored system. When a goal checker receives an event from the domain model instance, it checks whether the goal as a whole has been satisfied or has failed. To do this, it may need to query the domain model for additional information on other parts of the model.

An instrumentation approach is not the only possible approach to runtime monitoring. An alternative strategy is to periodically sample the state of the monitored system and to use this information to instantiate the domain model. An example of this is [4] in which the Java virtual machine debugging API is used to access the state of the system. However, the temporal logic specification of goals used does not allow state changes to be missed, which is a risk with a sampling method. Therefore, an instrumentation approach is taken.

The implementation of this system can be split into two distinct problems which need to be solved. The first problem is to instrument the system which is to be monitored. Normally, instrumentation will not be part of the system design but will be added to existing systems. The instrumentation should, as far as possible, be generated automatically from existing artifacts.

The second problem is to generate the domain model and goal checker from the domain object model and goal specifications.

4. Instrumentation

Instrumentation is code which outputs events from a program which can then be read by a monitor. It is beneficial to keep instrumentation code separate from the rest of the code so that the core code is easier to maintain and the instrumentation can be modified independently. It also helpful to be able to add instrumentation retrospectively so that monitoring can be introduced after the system is built. A Java program can be instrumented by source code modification or by byte code modification, which is the approach used in [5, 8].

The monitoring system described here uses AspectJ to insert instrumentation. AspectJ [7] is a general purpose, aspect oriented extension to Java. Its purpose is to allow cross cutting concerns to be separated. AspectJ requires the source files of the program. It can be run either as a preprocessor, generating new source code to be compiled normally, or generate byte code directly.

AspectJ adds a new entity to the Java language, called an aspect. An aspect is similar to a class, but represents a modular unit of cross cutting concern. An aspect contains pointcuts and advice. Advice is additional code which implements the cross cutting concern. Pointcuts determine where advice is executed.

A pointcut selects join points which match a given pattern. A join point surrounds a point in the execution where advice can be executed. For example, there is a join point around each method call and each modification of an attribute. Pointcuts may also have parameters, which are bound when a matching join point is reached. Examples of parameters are the object on which a method was called and the parameters of that method.

Advice can be executed either before or after the code within a join point. If it is executed after, it can be executed always or only if no exception was thrown. Each advice is related to a pointcut, and will execute at join points which match that pointcut.

Figure 3 shows the process of generating instrumentation aspects. The UML design of the system to be monitored is represented in XML using XMI. This allows individual or sets of methods, parameters and attributes to be referred to using XPaths. Then instrumentation aspects are generated from the UML design and a mapping from the domain model concepts to the UML design. The aspectJ compiler takes these aspects and the original source files for the system and outputs the instrumented system class files.

4.1. Generating Instrumentation For Events

A mapping from the domain model to the design is required to generate the instrumentation. The first step of

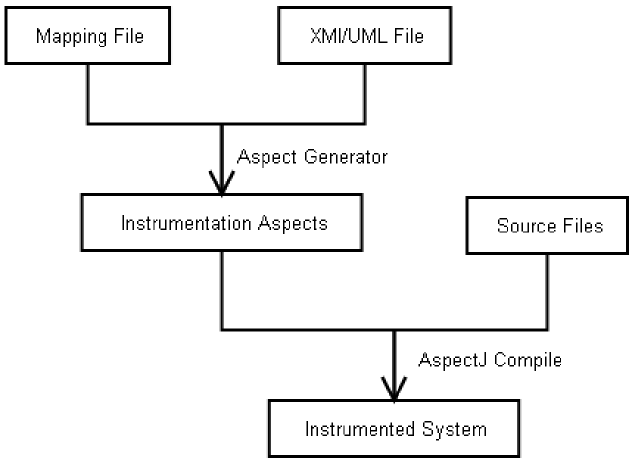

Figure 3: Instrumentation Generation Process

this mapping is to relate domain concepts such as relationships and events to corresponding types of AspectJ pointcuts. This mapping is described in XML documents. In the future it may be possible to generate these documents using some sort of mapping tool but at present these documents are written by hand.

Figure 4 shows the mapping of the events used in the example in section 2.1 to an implementation of these goals. The monitored program in this example is Limewire, a peer-to-peer file sharing program, written in Java. Limewire uses a client-server architecture to download files, although each node can take on the role of both a client and a server.

Each event in this mapping has a one or more ‘Occurs’ elements. An ‘Occurs’ element has a ‘method’ element which contains an XPath to an ‘Operation’ element in the XMI description of the system. This is the a point in the execution of the system where the event occurs. The ‘position’ attribute determines whether the event occurs before or after the method call.

Each ‘Occurs’ element contains any number of ‘Attribute’ elements. These specify the value of the event’s attributes. Each ‘Attribute’ element has an ‘instanceID-Element’ attribute which identifies an instance variable from which the value of the attribute should be extracted. By default this XPath uses the element identified in the ‘Occurs’ element as its context node. However, if the attribute has a ‘context’ attribute with the value set to ‘class’ then the ‘Class’ element which contains the ‘Operation’ element is used as the context node. The ‘Attribute’ element may also have an ‘instanceID’ attribute which is a line of Java code which used to extract a string value from the instance. If this is nor present then the object’s ‘toString()’ method is used.

It is necessary to find a unique identifier for each object which appears in an attribute. This is used by the goal checker to match different occurrences of the same object. In the example, the client and server are identified by their IP addresses while the file is identified by its name.

An example of a generated aspect for the ‘SentFile’ event is shown in figure 5. An instrumentation aspect is generated for each event. The event aspect will have a pointcut for each ‘Occurs’ element. In the example, there is only one ‘Occurs’ element, so there is only one pointcut, labelled ‘sentFileOccurs0’. All the attributes for this event are object attributes or accessed by method calls so the only parameter the pointcut requires is the instance of the ‘HTTPUploader’ class. The ‘call’ primitive pointcut matches all calls to methods with signature ‘void writeResponse()’. The ‘target’ primitive pointcut matches join points in which the target object is an instance of ‘HTTPUploader’. The pointcut as a whole matches only calls to the ‘writeResponse’ method of ‘HTTPUploader’.

There is also advice generated for the pointcut. In this example, the advice executes after the method returns without throwing an exception. This was specified by the ‘position’ attribute in the ‘Occurs’ element. The advice actually sends the instrumentation message, using the Java logging API. The attribute values are all use the ‘targetClass’ parameter of the pointcut to access member variables and methods. The ‘sentBy’ attribute also uses the information from the ‘instanceID’ attribute to access specific information about a socket.

4.2. Generating Instrumentation For Relationships

The mapping for the relationship ‘UploadAvailable’ is shown in figure 6. Relationship mappings are quite similar to event mappings. In place of the ‘Occurs’ element, relationship mappings have ‘Transition’ elements which indicate the positions in the execution where a new instance of a relationship exists or ceases to exist. The type of the transition is indicated by the value attribute of the ‘Transition’ element.

Matching a particular method is not sufficient to specify the transitions for this method. The ‘Condition’ element gives a condition which must also be met for the transition to occur. In the case of the first ‘Transition’ element in the example, the condition is that the ‘_stateNum’ should be equal to the value ‘Uploader.CONNECTING’. Conditions may also be used in events if necessary.

The roles in the relationship work match like the attributes in the event. The ‘instanceIDelement’ attribute specifies which element to use to get the value. The ‘instanceID’ attribute extracts the value from the instance.

```
<Event domainName="ReceivedRequest">
    <Occurs position="before"
            method="//UML:Class[@name='HTTPUploader']//UML:Operation[@name='HTTPUploader']">
        <Attribute domainName="requestedFrom"
                   instanceIDElement="//UML:Parameter[@name='socket']"
                   instanceID="socket.getLocalAddress().getHostAddress()"/>

        <Attribute domainName="requestedBy"
                   instanceIDElement= "//UML:Parameter[@name='socket']"
                   instanceID="socket.getInetAddress().getHostAddress()"/>

        <Attribute domainName="requestedFileName"
                   instanceIDElement="//UML:Parameter[@name='fileName']"/>
    </Occurs>
</Event>

<Event domainName="SentFile">
    <Occurs position="afterReturning"
            method="//UML:Class[@name='HTTPUploader']
                    //UML:Operation[@name='writeResponse']">
        <Attribute domainName="sentTo"
                   context="class"
                   instanceIDElement="//UML:Operation[@name='getHost']"/>
        <Attribute domainName="sentBy"
                   context="class" instanceIDElement="//UML:Attribute[@name='_socket']"
                    instanceID="_socket.getLocalAddress().getHostAddress()"/>
        <Attribute domainName="file"
                   context="class"
                   instanceIDElement="//UML:Operation[@name='getFileName']"/>
    </Occurs>
</Event>

<Event domainName="ReceivedFile">
    <Occurs position="afterReturning"
            method="//UML:Class[@name='HTTPDownloader']//UML:Operation[@name='doDownload']">
        <Attribute domainName="receivedBy"
                   context="class"
                   instanceIDElement="//UML:Attribute[@name='_socket']"
                   instanceID="_socket.getLocalAddress().getHostAddress()"/>
        <Attribute domainName="file"
                   instanceIDElement="//UML:Class[@name='HTTPDownloader']
                                      //UML:Operation[@name='getFileName']"/>
    </Occurs>
</Event>
```

Figure 4: Mapping from events to Limewire implementation

5. Goal Checking

The goal checking part of the monitoring system receives events from the program instrumentation. These events are used to construct an instance of the domain model of the system. To do this, the instrumentation has to produce three types of instrumentation events. Update to relationships indicate when specific instances of relationships exist or when they cease to exist. Update to attributes are sent whenever the value of an attribute changes. Updates to events indicate when an event occurs and what the attributes of that event are.

Goal checkers are generated by breaking the temporal logic formula into predicates and creating a tree structure of the predicates. The structure of the checker generated for the goal Achieve[ReliableTransmission] is shown in figure 7.

The top level of this tree is a checker for the goal type. In

```
package com.instrumentation;

import java.util.logging.*;

privileged aspect SentFile {
    protected Logger logger;

    public SentFile() {
        logger = logger.getLogger("EventLogger.SentFile");
    }

    pointcut sentFileOccurs0(com.limegroup.gnutella.uploader.HTTPUploader targetClass) :
        call(void writeResponse()) && target(targetClass);

    after(com.limegroup.gnutella.uploader.HTTPUploader targetClass) returning :
        receivedRequestOccurs0(targetClass) {
            Object[] recordParameters = new Object[2];
            String[] attributesNames = new Object[3];
            String[] attributeValues = new Object[3];

            attributeNames[0] = "sentTo";
            attributeValues[0] = targetClass.getHost();
            attributeNames[1] = "sentBy";
            attributeValues[1] = targetClass._socket.getLocalAddress().getHostAddress();
            attributeNames[2] = "file";
            attributeValues[2] = targetClass.getFileName();

            recordParameters[0] = attributeNames;
            recordParameters[1] = attributeValues;
            logger.log(Level.FINER, "SentFile", recordParameters);
    }
}
```

Figure 5: Aspect generated for ‘SentFile’ event

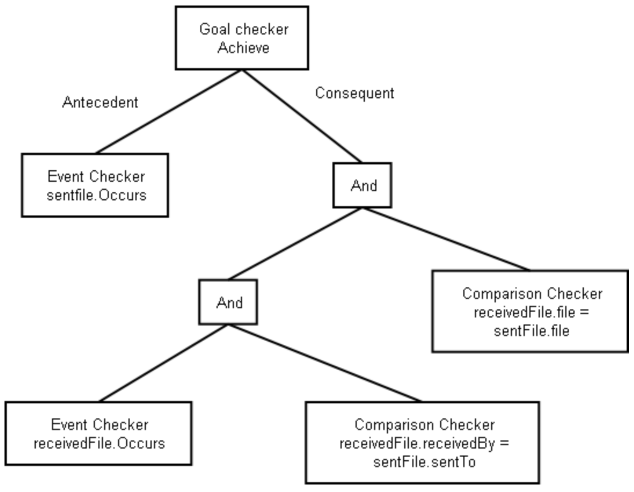

Figure 7: Goal Checker Structure For Achieve[ReliableTransmission]

this case, an achieve goal. This has an antecedent predicate and a consequent predicate. In this case, the consequent is a compound predicate.

The goal checker determines goal satisfaction using the domain model. Whenever the domain model is modified, the predicate checkers which depend on that part of the domain model are informed. Each relationship checker depends on the relationship in the domain model. Each comparison checker will depend on the attributes in the comparison. Each event occurrence checker will depend on the event checked.

Once a predicate checker has received an update, the goal checker starts to determine whether the parent predicate is true. The predicate binds instances of the objects in the update to the variable labels in the predicate. These bindings are then sent to the parent checker. Compound checker, such as AND and OR checkers, must examine their other sub-predicates to determine which bindings exist which will make the whole predicate true. This set of bindings is then passed up to the parent monitor. This pro-

```
<Relationship domainName="UploadAvailable">

    <Transition value="true" position="after"
        method="//UML:Class[@name='HTTPUploader']//UML:Operation[@name='setState']">
        <Condition>
            <Equals>
                <Value context="class"
                       instanceElement="//UML:Attribute[@name='_stateNum']"/>
                <Value instance="Uploader.CONNECTING"/>
            </Equals>
        </Condition>

        <Role domainName="sharedBy"
              context="class"
              instanceIDElement="//UML:Attribute[@name='_socket']"
              instanceID="socket.getLocalAddress().getHostAddress()"/>
        <Role domainName="shared"
              context="class"
              instanceIDElement="//UML:Attribute[@name='_fileName']"/>
    </Transition>

    <Transition value="false" position="after"
        method="UML:Class[@name='HTTPUploader']//UML:Operation[@name='setState']">
        <Condition>
            <Equals>
                <Value context="class"
                       instanceElement="//UML:Attribute[@name='_stateNum']"/>
                <Value instance="Uploader.CONNECTING"/>
            </Equals>
        </Condition>

        <Role domainName="sharedBy" instanceIDElement="//UML:Parameter[@name='socket']"
            instanceID="socket.getLocalAddress().getHostAddress()"/>
        <Role domainName="shared" instanceIDElement="//UML:Parameter[@name='fileName']"/>
    </Transition>

</Relationship>
```

Figure 6: Mapping from relationship to Limewire implementation

cess continues until a goal monitor is met, which is responsible for determining the satisfaction of temporal properties.

In the example goal checker, when the event 'SentFile' occurs, the event checker for the event will be be informed by the domain model listener. The event will then be bound to the label 'sentFile'. This binding will be passed up to its parent which is the achieve checker. The achieve checker will store this event and its attributes as an instance of the goal which is still to be satisfied.

When the event 'ReceivedFile' occurs, the checker for this event will be informed. The event will be bound to the label 'receivedFile' and then passed to its parent. In this case the parent is an AND checker. The AND checker then checks its other predicate to see if it is satisfied. If there is an instance of the 'Client' entity with the same identifier as the 'receivedBy' attribute of the 'ReceivedFile' event then that identifier is bound to the 'sentTo' attribute of the 'SentFile' event. The event is passed up to the parent of the AND checker which is another AND checker. This checker binds an instance of the 'File' entity to the file attribute if one exists. The event is then passed to the goal checker which records previous 'SentFile' events. This then compares the bindings for the unsatisfied instances with the new bindings. If any bindings match then an instance of the goal is satisfied.

5.1. Goal Checker Design

When a goal monitor is reached, its satisfaction is determined using the pattern base approach used in KAOS. Each goal specification belongs to a pattern and the monitoring system has a class for each goal pattern which is responsible for determining the satisfaction or failure of the goal.

The determination of the satisfaction or failure of goal patterns is done using the formal goal operationalisation patterns in [9]. For example, the operationalisation pattern for a bounded achieve goal is shown below:

$C \Rightarrow \Diamond_{\leq d} T$
Domain Pre-condition $\neg T$
Dom Post-condition T
Required Trigger Condition $\neg T \; S_{\leq d-1} C$

This operation specifies that there must be a transition from $\neg T$ to T when the relationship T has not been true for $d-1$ time units since C became true at the latest. The operationalisation pattern for the after invariant goal pattern is:

$C \Rightarrow \Box Q$
Domain Pre-condition $\neg C$
Domain Post-condition C
Required Post-condition Q

Domain Pre-condition Q
Domain Post-condition $\neg Q$
Required Pre-condition $\blacksquare \neg C$

In this case there are two operations. The first says that when the transition from $\neg C$ to C occurs, then Q must be true. The second says that when the transition from Q to $\neg Q$ occurs, C must have been false in all previous states.

6. Monitoring Soft Goals

Soft goals[10, 11] are goals which do not have formal criteria for their satisfaction. This is a different division than the division into functional and non-functional goals which can also be made, but is not important to the implementation of goal monitoring. The determination of whether a soft goal has been satisfied must be made by the stake holders in the system. The designer of the system must try to implement soft goals to a sufficient extent, taking into account conflicts with other soft goals.

Although it is not possible to formally specify criteria for the satisfaction of a soft goal, it should be possible to formally specify what needs to be monitored so that the determination of satisfaction can be made. We identify some specification patterns for soft goals and categorise them by what type of soft goals they may be suitable for expressing.

An unreliable goal is specified using a hard goal which cannot always be satisfied. The soft goal requires that as many instances as possible of this goal should be satisfied. This soft goal monitor needs to report the number of times the hard goal succeeds or fails. This type of goal specification allows reliability soft goals to be monitored.

By specifying an unbounded achieve goal but requiring that each instance of the goal should be satisfied in a reasonably short time. The monitor for this soft goal needs to report the time taken to satisfy each instance of the hard goal. This type of goal specification allows responsiveness soft goals to be monitored.

A maintain goal in which the consequent predicate cannot be true at all times. The monitor for this goal should report when the consequent predicate is not true. This type of goal specification allows availability soft goals to be monitored.

Soft goals are monitored by adding soft goal checkers to the monitoring framework described in figure 2. If the soft goal specification is based on a hard goal then the hard goal is checked by the existing hard goal checker. Instead of displaying failures of this goal, the output is sent to a soft goal checker, which collates the output. This collation may take the form of calculating averages, rolling averages, standard deviations or maximum and minimum values.

7. Conclusions

This paper describes two contributions. The first is a mapping from concepts in the KAOS meta-model to the design of a system. The second is the automatic generation of instrumentation aspects from the mapping.

We have a prototype implementation, written in Java, of some of the ideas in this paper. This implementation can generate the instrumentation files automatically. The domain model instance is constructed using Java data structures. The goal checker can check a subset of the goals expressible in KAOS. We have not yet implemented soft goal checkers or visual output from the monitors.

We have used this implementation to generate monitors for our Limewire case study. This shows that the approach can produce monitors for simple hard goals.

8. Related Work

Our work is similar to [3] which also looked at monitoring KAOS goal-oriented requirements specifications. This paper describes a system to monitor goals at run-time. The system is instrumented to emit events to a monitor which then informs a reconciler. The reconciler automatically resolves the failure by adapting individual goals or by switching to alternate goal refinements. The system relies on manual addition of instrumentation code to the monitored agents to produce events.

In [8], the concept of using a high level specification to generate the run-time checker and a low level specification to generate the instrumentation was introduced. The low level specification maps the concepts in the high level specification to the implementation.

In [5], a monitoring system is described which uses temporal logic as a specification language. This papers discusses some of the complexities involved in monitoring temporal logic formulae at run-time. However, we avoid some of that complexity by restricting ourselves to only monitoring the temporal logic formulae used in KAOS goal patterns.

9. Future Work

At present, the monitoring system only runs as a local monitor, running on the same machine as the system, which can only run on that system. The next step is to allow the monitor to run on a different machine from the monitored system and to accept instrumentation events from several machines. This introduces a timing problem which has to be solved, as instrumentation events may not arrive at the goal checker in the same order they are generated.

It still need to be determined what feedback goals monitors should generate. The general aim is that the feedback should support system evolution. The intention is that the goal checkers should generate generic information which can then be interpreted by various gauges in different ways.

10. Acknowledgements

This research has been supported by BTexact and EPSRC as part of the collaborative programme in ‘Generative Software Development’. We are grateful for their generous support.

References

[1] Gnu general public license. `http://www.gnu.org/copyleft/gpl.html`.

[2] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[3] M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *Proceedings of the 9th International Workshop on Software Specification and Design*, pages 50–59, 1998.

[4] R. J. Hall. Cpprofj: Aspect-capable call path profiling of multi-threaded java applications. In *17th International Conference On Automated Software Engineering*, pages 107–116, 2002.

[5] K. Havelund and G. Rosu. Monitoring java programs with java pathexplorer. In K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[6] M. Jackson. The world and the machine. In *International Conference on Software Engineering*, pages 283–292, 1995.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[8] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-mac: a run-time assurance tool for java programs. In K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[9] E. Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD thesis, Université catholique de Louvain, 2001.

[10] J. Mylopoulos, L. Chung, and B. A. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *Software Engineering*, 18(6):483–497, 1992.

[11] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering*, pages 226–235, 1997.