From Requirements to Monitors by way of Aspects

Andrew Dingwall-Smith, Anthony Finkelstein
Department of Computer Science
University College London
Gower Street, London, WC1E 6BT UK
{a.dingwall-smith,a.finkelstein}@cs.ucl.ac.uk

Abstract

Using goal driven requirements engineering, requirements are derived from a goal model that captures multiple strategies for satisfying the goals and takes into account environmental constraints on the system. The model is therefore more stable than a conventional requirements document.

We present early work in building a system for runtime monitoring of system goals, as part of normal system operation, so that failure to achieve goals caused by changes in the system environment can be detected and acted on. We make use of Hyper/J to separate instrumentation for monitoring from the core code, and to add instrumentation directly to class files, without the need to modify the core class files. We are currently using a peer to peer networking client as a testbed and we present examples based on this program.

1. Introduction

This position paper presents early work on using requirements specifications to support monitoring of software systems. In the face of a changing environment, a system may no longer be able to meet the goals required of it, due to early assumptions about the environment on which the design has been predicated no longer holding. For this reason, monitoring a system as part of its normal operation is important so that user can be informed of failures or the system configuration changed to take account of environmental changes. This has similarities to [2] which also deals with monitoring a system based on the system goals.

Monitoring systems typically work by instrumenting programs to emit events to the monitoring system, either on the same machine or another machine. Instrumentation of Java programs can be done by inserting instrumentation into the source code or into class files [3]. Instrumentation of class files is done by creating custom tools which manipulate the byte code to insert the instrumentation, according to high level specifications. This enables the monitoring system to be separated from the program to be monitored.

Our monitoring system makes use of Hyper/J [4],[6] to instrument class files. By using a general purpose tool to support separation of concerns, we eliminate the need to write tools for manipulating bytecode directly.

We are using the Limewire, Gnutella peer to peer networking program as a testbed for our approach. This program is freely available and is written in Java. In Limewire we have a relatively simple system to deal with. However, it is a system which should still be subject to a changing environment as it is affected by the many other Gnutella servants it connects to and issues such as bandwidth and network traffic. This paper uses examples based on our work using Limewire.

2. Goal driven requirements engineering and KAOS

Goal driven requirements engineering is an approach to requirements engineering that aims to capture the rational for requirements and assist in the elicitation of requirements. Goals are elicited from the stakeholders in the system. These goals are used to build a goal model in which the goals are decomposed into sub-goals which describe in greater detail how the goals should be satisfied. Ultimately, requirements can be derived from the goal model. New higher level goals can also be added to capture the purpose of goals.

In general, a set of goals can be decomposed in many ways, providing many possible implementations. The goal model captures multiple goal decompositions and assists in selecting the appropriate decomposition by capturing conflicting goals. The goal model should be more stable than a requirements specification.

The KAOS approach[1], is an example of a goal driven requirements engineering method. This approach specifies goals in terms of an objects model. This provides two views of the system, with one view crosscutting the other.

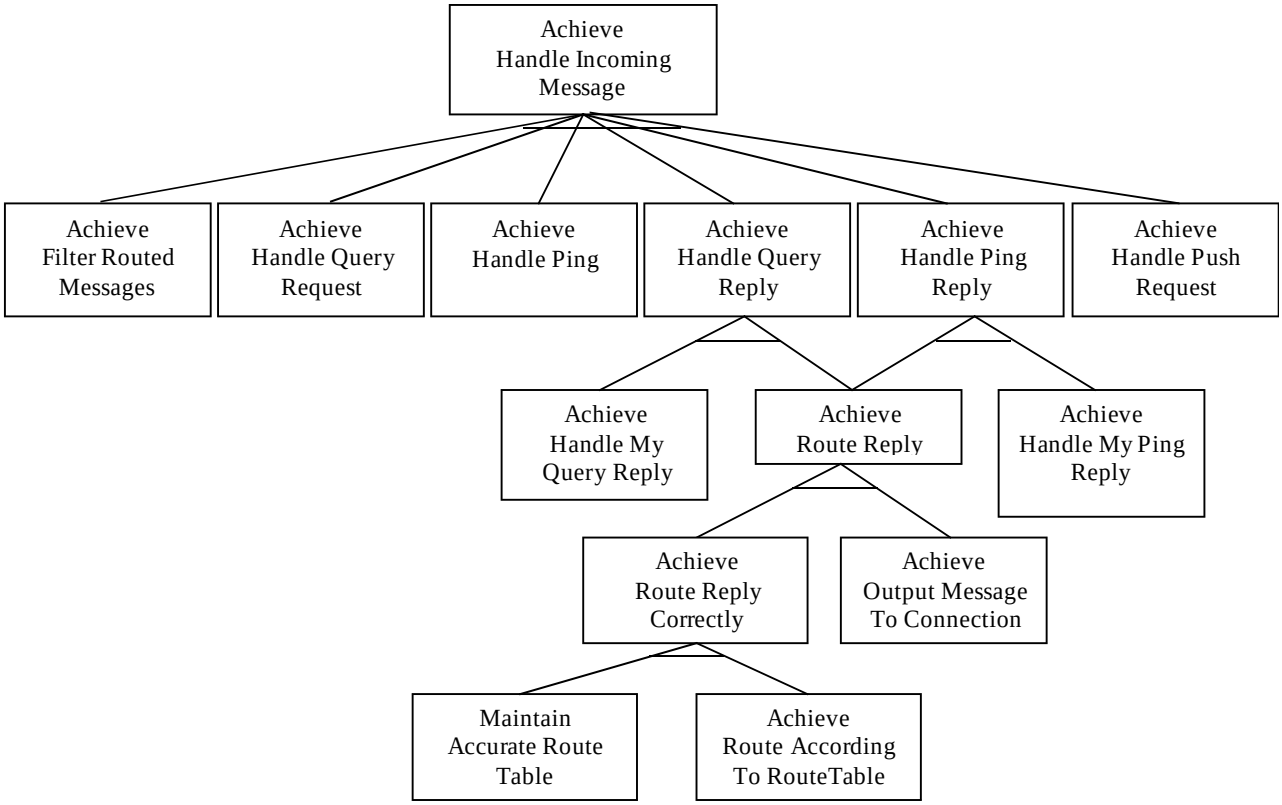

Figure 1. Partial goal decomposition for goal HandleIncomingMessage

KAOS allows goals to be defined using temporal logic if formal specification is desired. The temporal logic formulae refer to objects in the object model. KAOS defines several goal patterns such as Achieve, Maintain and Avoid, which correspond to a certain types of temporal logic formulae. Some common definitions are:

Achieve: $P \Rightarrow \Diamond Q$
Maintain: $P \Rightarrow \Box Q$
Avoid: $P \Rightarrow \Box \neg Q$

These logic operators may also have time constraints on them.

Each instance of an achieve goal must eventually be satisfied by the system. The formal specification of the goal may constrain the time in which the goal instance must be achieved.

3. Goal decomposition for Limewire

We have identified goals that stakeholders in the system are likely to require and constructed monitors for these goals. A partial goal graph for the high level goal 'Handle Incoming Message' is shown in Fig. 1. This goal requires that incoming messages should be handled, in accordance with the Gnutella protocol [5]. This involves sending requests to other connected servants, sending replies back to their originators, responding to requests and so on. Sub-goals are to filter out any unwanted messages and to handle each type of message defined in the Gnutella protocol.

Both the goals 'Handle Query Reply' and 'Handle Ping Reply' have the sub-goal 'Route Reply', as these messages are routed in the same manner. This goal is defined as:

Achieve[RouteReply] Use the route table to route this reply to the connection from which this connection originated.

The goals 'Handle My Ping Reply' and 'Handle My Query Reply' deal with the cases where the reply is a response to a request sent out by our own servant program. The 'Route To Source' goal is further decomposed into the goals 'Routing Reply Correctly' and

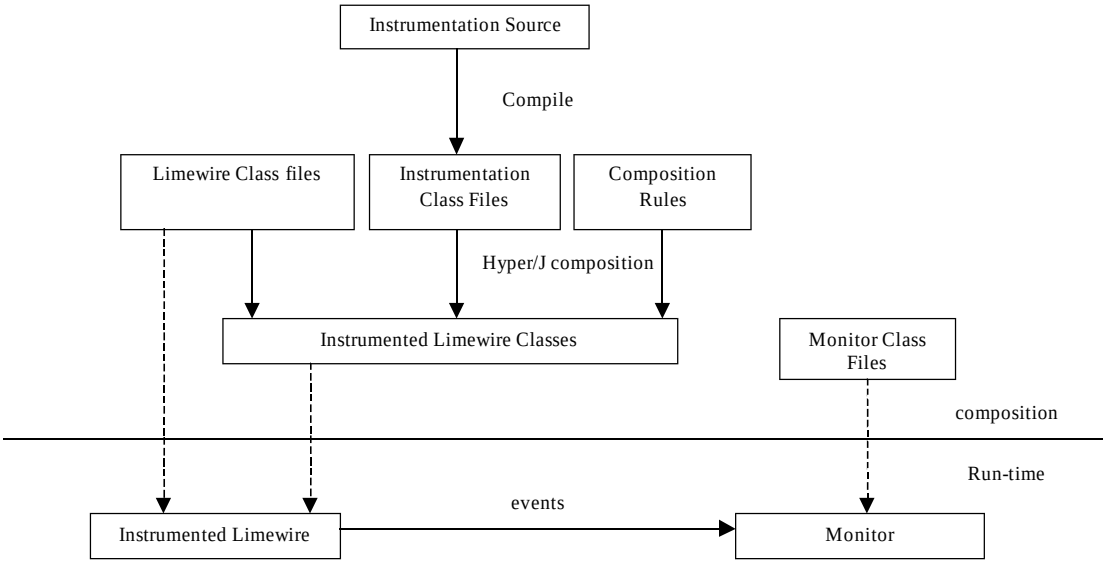

Fig. 2. Monitoring system architecture

'Output Message To Connection'. The first of the goals requires that the messages sent to a connection are sent to the connection held in the route table. The second goal requires that all the messages received are sent to a connection, not dropped. These goals are defined as:

Achieve[RouteReplyCorrectly] Messages should be sent to the correct connection, according to the route table.

Achieve[OutputMessageToConnection] A managed connection which is given a message should output that message to the connection.

The 'Route Reply Correctly' goal can be achieved by sending the reply to the connection in the route table, which is the goal 'Route According To Route Table', and the goal 'Accurate Route Table' which says that the entries in the route table should be correct.

4. Monitoring architecture

The monitoring system architecture is shown in Fig. 2. The inputs to the system are the Limewire class files, instrumentation code, written in Java, the Hyper/J composition rules and the monitor systems class files. The instrumentation source must be compiled by a normal Java compiler as Hyper/J operates on class files. Hyper/J must then be run to integrate the instrumentation classes and the Limewire classes, according to the composition rules. To run the instrumented program, the Java runtime environment needs access to the original Limewire class files, the instrumented Limewire classes and the monitoring system classes.

The original classes are required as the composition rules tell Hyper/J to only output the classes which need to be instrumented. The instrumented Limewire classes will have the same names as the original Limewire classes so it is important that the class path is set so that the instrumented classes are searched before the original classes.

The monitoring system runs on a single machine using a multi-threaded architecture. Instrumentation is inserted into the program, using Hyper/J, which places events in a queue. At a set interval, a monitoring thread reads the events from the queue and uses the events to determine whether the monitored goal is being satisfied.

5. Hyper/J composition rules

The Hyper/J composition rules, Fig. 3, introduce two dimensions of concern, in addition to the existing Object dimension, which is created automatically. In the ‘Goal’ dimension, each monitor belongs to the concern corresponding to the goal it is trying to monitor. In the ‘Aspect’ dimension, the original Limewire classes are in the ‘Core’ concern. Only those classes which need to be instrumented are imported into the hyperspace, using ‘as in package’ at lines 14 and 15 of the specification file.

In Java, a class which has no constructor defined for it

```
1. -concerns
 2.   package monitor.outputmessage : Goal.OutputMessage
 3.   operation monitor.outputmessage.ManagedConnection.<init> : Goal.None
 4.   operation monitor.outputmessage.PingReply.<init> : Goal.None
 5.   operation monitor.outputmessage.QueryReply.<init> : Goal.None
 6.   operation monitor.outputmessage.RouterService.<init> : Goal.None
 7.
 8.   package monitor.routecorrectly : Goal.RouteCorrectly
 9.   operation monitor.routecorrectly.ManagedConnection.<init> : Goal.None
10.   operation monitor.routecorrectly.MessageRouter.<init> : Goal.None
11.   operation monitor.routecorrectly.RouterService.<init> : Goal.None
12.   operation monitor.routecorrectly.RouteTable.<init> : Goal.None
13.
14.   package com.limegroup.gnutella as in package monitor.outputmessage : Aspect.Core
15.   package com.limegroup.gnutella as in package monitor.routecorrectly : Aspect.Core
16.
17. -hypermodules
18.   hypermodule MonitoredLimewire
19.     hyperslices:
20.       Aspect.Core,
21.       Goal.OutputMessage,
22.       Goal.RouteCorrectly;
23.     relationships:
24.       mergeByName;
25.
26.       order action Goal.RouteCorrectly.ManagedConnection.handlePingReply
27.       before action Aspect.Core.ManagedConnection.handlePingReply;
28.
29.       order action Goal.RouteCorrectly.ManagedConnection.handleQueryReply
30.       before action Aspect.Core.ManagedConnection.handleQueryReply;
31.
32.       order action Goal.OutputMessage.ManagedConnection.handlePingReply
33.       before action Aspect.Core.ManagedConnection.handlePingReply;
34.
35.       order action Goal.OutputMessage.ManagedConnection.handleQueryReply
36.       before action Aspect.Core.ManagedConnection.handleQueryReply;
37.
38.       order action Goal.OutputMessage.PingReply.writePayload
39.       after action Aspect.Core.PingReply.writePayload;
40.
41.       order action Goal.OutputMessage.QueryReply.writePayload
42.       after action Aspect.Core.QueryReply.writePayload;
43.
44.       order action Goal.RouteCorrectly.MessageRouter.handlePingReply
45.       before action Aspect.Core.MessageRouter.handlePingReply;
46.
47.       order action Goal.RouteCorrectly.MessageRouter.handleQueryReply
48.       before action Aspect.Core.MessageRouter.handleQueryReply;
49.
50.   end hypermodule;
```

Fig. 3. Hyper/J composition rules

has a default constructor generated by the compiler. This can cause problems with Hyper/J, as classes without constructors are often not intended as stand alone classes. The default constructors generated for the goal monitor classes need to be explicitly assigned to the Goal.None concern so that they are excluded from the composition.

The relationships section specifies the merge by name composition strategy, meaning that classes with the same name are merged together by merging fields and methods with the same name. The order relationships specify the relative order of method bodies in the merged methods, in the cases where the order is important.

The goals which are being monitored can be changed by simply adding or removing those concerns to the composition and then running Hyper/J again.

The classes into which the instrumentation has to be composed are obtained from the goal specifications. The goals are defined in terms of a domain-level object model. The domain-level objects then have to be related to the implementation object model. At present, we only do this informally, using the natural language definition of a goal and the objects that are referred to in this definition. Using

the temporal logic specification used in KAOS, and a formal object model, may allow us to formally derive this relationship. As an example, the goal 'Output Message To Connection' refers to the objects "Managed Connection' and 'Message'. In the Hyper/J composition rules, instrumentation for this goal is composed into the Limewire classes 'Managed Connection', 'Ping Reply' and 'Query Reply'. 'Ping Reply' and 'Query Reply' are sub-classes of the Limewire 'Message' class. This shows the correspondence between the Limewire classes which are instrumented and and the goal definition.

6. Monitor design

We have implemented monitors for the goals 'Output Reply To Connection' and 'Route Message According To Route Table'. Since both of these goals fit into the achieve pattern, the monitors are quite similar. In the case of the 'Output Message To Connection' goal, whenever the 'Managed Connection' class receives a reply message, the monitor stores the record of that event. The record includes the time of the event and a reference to the Message object that was received. At set intervals, the monitor checks every stored record to see if the time since it occurred is greater than the time constraint on the goal. If it is then the record is removed from the store and the monitor reports that the program has failed to achieve the goal.

Whenever a message is sent to the connection by the 'Query Reply' or 'Ping Reply' classes, the monitor searches the record of events for one with a matching Message reference. If one is found then it indicates that the goal has been successfully achieved. Otherwise, the time constraint on the message must already been violated and the failure reported, so no further action is necessary.

7. Conclusions and further work

Our system currently only monitors goals which match the KAOS achieve pattern. Implementing monitors for other patterns such as maintain and avoid should be fairly trivial.

We also intend to look at monitoring non-functional goals and soft goals. Non-functional goals seem particularly suited to our approach. Soft-goals, that is goals that do not have defined conditions for achievement, are more difficult. For soft goals, it is not obvious what sort of information is useful to the user or the program itself.

Our goal is to generate monitors from high level specifications, that is, generating monitors from the temporal logic used in KAOS. To date we have used the specification as a guide but hard coded the monitors to give us a sense of whether the approach might work.

8. Acknowledgements

This research has been supported by BTexact and EPSRC in their collaborative programme 'Generative Software Development'. We are grateful for their generous support.

9. References

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

[2] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. "Reconciling System Requirements and Runtime Behaviour", Proc. IWSSD'98 - 9th International Workshop on Software Specification and Design, IEEE CS Press, April 1998.

[3] M. Kim, S. Kannan, I. Lee, O. Sokolsky and Mahesh Viswanathan. "Java-MaC: a Run-time Assurance Tool for Java Programs", Proc. RV'01- 1st Workshop on Runtime Verification, Electronic Notes in Computer Science, 55(2), 2001.

[4] P Tarr, H Ossher, W Harrison and SM Sutton, Jr. "N degrees of Separation: Multidimensional separation of concerns", Proc. ICSE 99, IEEE, May 1999, ACM press

[5] "The Gnutella Protocol Specification v0.4", http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf

[6] "Hyper/J" http://www.alphaworks.ibm.com/tech/hyperj