A Model for Dynamic and Adaptable Services Management

Patrick Martin, Wendy Powley, Imad Abdallah, Jun Li, Andrew Brown
Queen's University, Kingston ON
{martin, wendy, imad, junli}@cs.queensu.ca

Kirk Wilson, Chris Craddock
CA Labs, CA Inc
Kirk.Wilson@ca.com

Abstract

The dynamic nature of Service-Oriented Architectures challenges traditional systems management practices which tend to be static in nature. We propose a goal-oriented, agent-based approach to management using autonomic computing. In this paper we define a services management model that consists of a number of constructs including managed resources, agents, events, event streams and management goal graphs. Agents accept new events on one or more event streams, and the arrival of an event triggers local processing that includes the generation of a new event and/or changes to the state of the managed system. Simple agents are combined into management goal graphs to carry out complex management tasks. We provide details of an implementation of our services management model and show how this model could be used in a sample management scenario.

1. Introduction

Systems management ensures the correct, efficient and secure operation of managed systems and applications. The growing popularity of Service-Oriented Architecture (SOA), which allows applications to be constructed of existing services in a dynamic manner, encourages a reexamination of the suitability of traditional management system architectures.

The managed systems in a SOA environment have several unique characteristics, namely the systems are composed of independent, distributed services; the composition of the services is determined dynamically; the services may be from different organizational domains; the management goals for a composition may be determined when the composition is created, and there is a need to reconcile the management of a new composition of services with the existing managed services and their goals. The management of a SOA environment must therefore be both dynamic and adaptable. Traditional management systems are intended for client-server architectures that are more staticin nature than SOAs and so do not naturally satisfy these requirements.

We propose a goal-oriented agent-based approach to defining a management system for services. We first identify the specific management goals, such as enforcing an application's Service Level Objective (SLO), ensuring an overall level of performance for a managed resource (or set of managed resources), detecting and correcting a problem or ensuring the recency of activities such as backups of a managed resource. We then create a management system by combining cooperating agents where an agent's behaviour is determined by a user-defined policy that is deployed to the agent. The agent approach is preferable to the traditional centralized approach to management for several reasons [9]:

- The agent approach is more adaptable to the dynamic nature of services.
- The agent approach is better able to handle overloads (both communication and computation).
- There is no single point of failure in the agent approach.

The main management tasks include monitoring, analysis, action planning, action execution and report generation. We observe that the first four of these tasks correspond with the stages of the MAPE loop proposed in the autonomic computing paradigm [10]. We believe that autonomic computing is a natural fit to the management of services, in general and Web services, in particular (for example see our previous work [12]) and use the paradigm in the development of our approach. We can identify different types of agents for each of the above management tasks.

A framework to support our approach to managing services must involve the following three key components:

- A services management model to describe management tasks and goals.
- A method to generate management system components from specifications using the model.

- An infrastructure to allow the integration of management tasks and user interaction with the management system.

In this paper we focus on the services management model developed for our framework. The aim of the model is twofold. First, the model provides a method for describing the autonomic management of a service and its components. In our model these are defined in terms of agents, event streams and policies. Second, the specifications produced with the model can serve as input to the automatic (or semi-automatic) generation of an agent-based implementation to carry out the management of the service.

The remainder of the paper is structured as follows. Section 2 discusses relevant work from the literature. Section 3 introduces an example scenario that is used throughout the paper to illustrate our model. Section 4 details the services management model. In Section 5 we discuss our implementation of the management scenario described in Section 3 and show how the model is used to build a management system. Section 6 provides the conclusions and suggestions for future work.

2. Related Work

SLA-based service management has been proposed by several researchers where different aspects of service management have been addressed. Dan *et al.* [5] propose a comprehensive framework for SLA-based automated management for Web services with a resource provisioning scheme to provide different levels of service to different customers in terms of responsiveness, availability, and throughput. Levy *et al.* [11] propose an architecture and prototype implementation of a performance management system to provide resource provisioning and load balancing with server overload protection for cluster-based Web services. Sahai *et al.* [15] propose a Management Service Provider (MSP) model for remote or outsourced monitoring and control of E-services on the Internet. Sahai *et al.* [14] also propose an automated and distributed SLA monitoring engine for Web services using the Web Service Management Network (WSMN) Agent. Their approach uses proxy components attached to SOAP toolkits at each Web service site of a composite process to enable message tracking.

Cohen et al. [4] use the EventScript language to solve complex event processing problems. They showed that the simple set of primitives is enough to solve many event processing tasks. EventScript is a reactive process programming language for embedded actions. It uses regular expressions to specify patterns of occurrence of the event flows in the event. A stream of incoming events is matched to the regular expression. Actions embedded within the regular expression are executed in response to the pattern matching the input events.

Adi and Etzion [1] introduce the concept of situation and situation manager as part of the Active Middleware Technology (AMIT) project. AMIT is used as the core technology behind the E-business Management Services of IBM Global Services [8]. It extends the semantics of Snoop [3] by adding additional operators, introducing the lifespan element to the situation manager's definition language, and enabling the definition of time intervals.

Our management model uses policies to define the behaviour of the system. One of the major research issues is the decomposition of high-level policies into low level management policies. Rubio-Loyola et al. [13] use goal-oriented requirements engineering and model checking techniques, as a basis for their study of policy decomposition. They aim to translate business objectives to configurations on the managed resources.

We previously developed the *Autonomic Web Services Environment (AWSE)*, which is capable of self-management to ensure SLA compliance [12] AWSE is comprised of many *sites*, each site consisting of a collection of components and resources necessary for hosting Web services provided by an organization. Individual components, HTTP servers, application servers, database servers, and Web service applications, each have one or more associated autonomic managers. AWSE was implemented using the Web Services Distributed Management (WSDM) standard [17].

3. Management Example

To illustrate the use of our management model, we consider a management scenario for a Web server. In this scenario, the management system watches for, and reacts to, system overloads. The metrics that are supplied by the Web server include the workload intensity (the number of calls per unit time that are issued to the Web server) and the rejection rate (that is, how many calls are rejected due to lack of resources to service the calls). An increase in the workload intensity combined with an increase in the number of rejected calls indicates that the Web server is overloaded, possibly requiring an adjustment to the configuration of the Web server. One solution to alleviate overload symptoms such as these is to increase the number of threads to service client calls, providing that the load that can be serviced by the

additional threads will not overload other components such as the database system.

In order to recognize the symptoms of system overload, the management system must be able to determine that that there has been an increase in workload intensity, that there has been an increase in the number of rejected calls and that these two symptoms are occurring simultaneously. When an overload has been detected, appropriate action must be taken.

This management example is used to illustrate the various constructs and relationships defined in the Services Management Model.

4. Services Management Model

We view a management system for services as a collection of agents that interact through the processing and generation of event streams. Each agent performs a relatively simple function and all agents follow the same general behavior pattern. Agents accept new events on one or more event streams and the arrival of an event triggers local processing. The results of the processing can be the generation of a new event on an outgoing event stream and/or changes to the state of the managed system or the management system itself. These simple agents are combined into what we call management goal graphs in order to carry out complex management tasks. The management goal graph for the management scenario is shown in Figure 1.

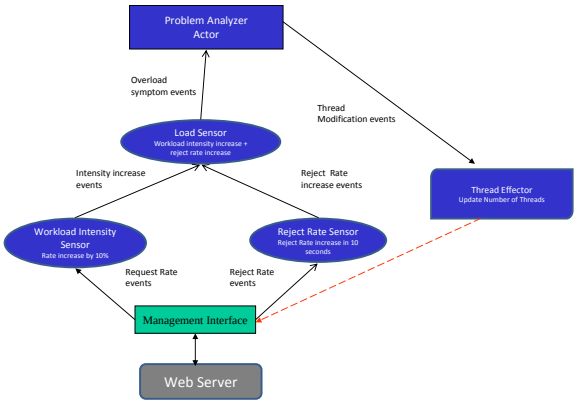

Figure 1: Management Scenario

Our services management model, which is described in detail below, consists of a number of constructs including managed resources, agents, events, event streams and management goal graphs. These constructs are illustrated using the management example outlined in Section 3.

4.1. Managed Resources

Managed resources are the services and components being managed. A managed resource provides a set of metrics to describe its state and performance and a set of configuration parameters that can be adjusted to affect its state and performance. We assume that performance metrics are pushed from the managed resource to the management system in the form of events.

4.2. Events

An *event* is the occurrence of a situation, or incident, within a service or the management system. Events may signify a problem such as a threshold violation or a deviation from typical behaviour or they may simply indicate normal or expected occurrences within the system such as the completion of a task.

We consider two kinds of events in our model; concrete and inferred events. *Concrete events* happen in the managed system, thus, outside of the management system. These events are relayed into the management system via sensors. Examples of concrete events include the arrival of a client, the failure of a transaction, or the report of a performance metric such as throughput. *Inferred events*, on the other hand, occur in the management system and are logically concluded by viewing the current state of the managed system and the relevant history of event occurrences. We call this relevant history of event occurrences the *context* of the current event. An event is represented in the management system by an *event instance* that contains relevant information about the event.

4.2.1. Event type. An *event type* describes the common properties of a similar set of event instances. In other words, it defines a schema of attributes that are instantiated at run time when an event of that type occurs. An event type **E** is a tuple **<id, attr$_1$, …, attr$_n$>** where

id is a unique type name;

attr$_i$ ($1 \le i \le n$) is a pair **<attr-id$_i$, attr-type$_i$>** where **attr-id$_i$** is a unique attribute name within the type **E** (that is **id.attr-id$_i$** is globally unique) and **attr-type$_i$** is one of {integer, string, real, timestamp, URI}.

For example in Figure 1, the event type for the events reporting the average request rate from the Web Server over a time interval is E = (WSReqRate,

(startTime, timestamp), (endTime, timestamp), (reqRate, real)).

4.2.2. Event instance. An *event instance*, as stated above, represents the occurrence of an event. An event instance **e** is a tuple **<id, type, stream, timestamp, attr-value$_1$, …, attr-value$_n$>** where

id is a unique identifier;

type is the event type of the instance e;

stream is the id of the event stream containing the instance e;

timestamp is the time at which the instance was created;

attr-value$_i$ ($1 \leq i \leq n$) is a value for the attribute **attr$_i$** associated with the type of the instance.

For example, an event instance of the WSReqRate event type for the time period 1:00 – 1:05 AM (is e = (e1, WSReqRate, s1, 1:05, 1:00, 1:05, 62). This indicates that at time 1:05 an event instance was created on the event stream s1 indicating that the average request rate during the time period 1:00 and 1:05 was 62 requests per second.

4.2.3. Event context. It is assumed that a history of event instances is kept by the management system. The *event context* of an event instance e_i is a description of the relevant history for e_i. In addition to the arrival of e_i at an agent, the context must also be true in order for that agent to act. The combination of e_i and its context is analogous to a pattern in complex event processing and matching that pattern causes the agent to act.

An event context can be specified as a predicate of the form

$$\exists c_1, c_2, ..., c_n\,(context(e_i))$$

where $\boldsymbol{c_j}$ *(1 ≤ j ≤ n)* is an event instance and ***context(e_i)*** is a conjunction of conditions of the following forms:

Sequence (c_1, c_2, …, c_n): the events c_j *(1 ≤ j ≤ n)* exist in the event history and appear in the specific order with respect to time.

Type(c_j) = E: the event c_j where *(1 ≤ j ≤ n)* is an event instance of event type *E*.

Attribute condition: $c_j.attr_k$ <*op*> <*operand*> where $attr_k$ is an attribute of the event instance c_j, <*op*> is one of {=, >, ≥, <, ≤} and <*operand*> is a value of the type of $attr_k$ or an attribute of another event instance, say $e.attr_m$, with the same type.

Window condition: a window in which the context events must occur using a time-based condition, i.e. *WindowFromTime (t, n)*, where *t* is a point in time and n is the minimal number of events that must occur within the window, or an event-based condition, i.e. *WindowFromLast (E, n)*, where *E* is an event type. and n is a count of events of type E.

For example, on the arrival of an event instance *e* of event type WSReqRate the Workload Intensity Sensor in Figure 1 looks for an increase of 10% in the request rate of the Web Server over the last reported rate. The context in this case can be specified as

$$\exists c((WindowFromLast(WS\,\mathrm{Re}\,qRate,1)) \cap (Type(c) = WS\,\mathrm{Re}\,qRate) \cap (e.throughput > 1.1 * c.throughput))$$

4.2.4. Event Stream. An event stream represents the flow of events from a source (external or agent) to one or more destination agents. All agents subscribed to a stream see all the event instances published on the stream. An event stream is a tuple M = <**id, E, source, destinations**> where

id is a unique id for the event stream;

E is the event type of event instances carried on the stream **M**;

source is the publisher of events on the stream **M** (it can be an external metric in the case of concrete events or an agent in the case of inferred events);

destinations is a set of agents subscribed to the stream **M**.

4.3. Agents

There are three main types of agents: Sensors, Actors and Effectors. *Sensors* monitor event streams and produce new event streams based on what they observe. *Actors* carry out a management function when triggered by the input of an event or by a user. *Effectors* impose changes for the management system on the managed resources. An agent is a tuple **A = <id, p>** where **id** uniquely identifies an agent, and **p** is a *policy*.

The behaviour of an agent is defined by **p,** its active policy, which specifies the input streams, the output stream, the event context(s) and, in the case of actors and effectors, the management action. A policy is a tuple, **p = <I, O, pattern, f >** where

I = *{I_1, I_2, …, I_p}* is the set of *p* input streams.

O is the output stream. When one of the patterns in the set **pattern** is matched, an inferred event is created and placed on **O**. We assume that effectors do not have an output stream.

pattern = *{(E_1,$context_1$), (E_2, $context_2$), …, (E_p,$context_p$)}* is a set of *p* patterns, one for each input

stream. For an input stream I_j, when an event instance of event type E_j arrives, the context predicate $context_j$ is checked**f** is a management function that is triggered when the pattern in ***pattern*** is matched. In the case of a sensor node, **f** is a null function. **f** will usually produce side-effects, in terms of modifications to the managed system or to the state of the management system.

4.4. Management Goal Graphs

Agents are combined into a *management goal graph* to achieve a specific management goal. A management goal graph is a tuple **G** = <**A**, **S, R**> where

A is set of agents;

S is a set of event streams;

R is a set of managed resources.

Management goal graphs are typically hierarchical in nature, with the managed resources at the bottom, sensors forming the lower level of the hierarchy, actors in the middle tier, and effectors at the top level. The sensors monitor and aggregate information from event streams that are emitted either by the managed resources or by other sensors. New event streams are generated by sensors and are passed either to other sensors, or to one or more actors. The actors use the information received from the sensors to make management decisions and emit event streams to invoke the effectors which imposes the desired change to the managed resources.

An example of a management goal graph for our management example is shown in Figure 1. The Web server is the managed resource and it has two outgoing concrete event streams. In this example, these streams are emitted by a management interface which is assumed to be monitoring the performance of the Web server. One stream carries concrete events about the rate of requests and the other stream carries concrete events about the rate of rejection of requests.

Several agents, including Sensors, Actors and Effectors, are defined in this scenario. The input stream for the Workload Intensity Sensor carries the request rate events from the Web server. The pattern defined for the Workload Intensity Sensor is (RequestRateEvent, "two consecutive events each show an increase of 10% in the request rate"). If the pattern is matched, that is, a request rate event arrives and it is found that two consecutive events each show an increase of 10% in the request rate, then the sensor places an intensity increase event (WorkloadIntensityEvent) on its outgoing stream.

The Reject Rate Sensor's input stream consists of reject rate events. The pattern set for this sensor is (RejectRateEvent, "the reject rate has increased within the last 10 seconds"). If the pattern is matched, then the sensor places a reject rate increase event (RejectRateIncreaseEvent) on its outgoing stream.

The input for the Load Sensor consists of two input streams. The pattern set for this sensor contains two elements as follows: (RejectRateIncreaseEvent, "a Workload Intensity Increase Event was emitted less than one minute ago"), (WorkloadIntensityEvent, "a Reject Rate Increase Event was emitted less than one minute ago"). If one of the patterns in the set is matched, then it indicates that events from both the Reject Rate Sensor and the Workload Intensity Sensor have been received. In this case, the Load sensor places an overload symptom event (OverloadEvent) on its outgoing stream.

The Problem Analyzer Actor determines what action should be taken as a result of overload symptom events that arrive on its input stream. The pattern for the Problem Analyzer Actor is simply the existence of the arrival of an OverloadEvent event. This triggers the management function, CheckCapacity(), which runs code that analyzes system capacity and determines whether or not adding more threads to the server to service the increase number of requests will be beneficial. If increasing the number of threads will be beneficial, the Problem Analyzer Actor places a Thread Modification Event on its outgoing stream for the Thread Effector. The Thread Effector then adjusts configuration parameters on the Web Server to increase the size of its thread pool.

5. Implementation

A prototype implementation of a management system using the services management model has shown the viability of this approach for a set of scenarios involving the management of Web services and components used to support these services. We describe our implementation with reference to the scenario described in Section 3.

An implementation of the services management model must provide support for the building and deployment of agents, expression for the policies that govern their behaviour, storage and query capabilities for event histories, as well as event processing and message passing.

Our implementation is based on the OASIS standard for Web Services Distributed Management (WSDM) [17]. WSDM specifies how the manageability of resources is made available to management clients by means of Web services. WSDM is based on the fact that Web services technology provides an open, standard platform on

which to base management infrastructure. WSDM provides standard communication protocols and message passing for the agents used in our model. WSDM employs Web Services Notification (WSN) [18] to support the publish/subscribe pattern of message exchange. Using this mechanism, event notifications are published by a WSDM entity to a "topic" and those entities wishing to receive events subscribe to relevant topics. A topic is simply a way to organize and categorize items of interest for subscription.

We assume that each managed resource employs a WSDM management endpoint through which the management system can obtain performance metrics and adjust configuration parameters. This management endpoint corresponds to the "Management Interface" shown in Figure 1. The metrics provided by each managed resource are obtained through the WSDM management endpoint via notifications that are published periodically by the endpoint to appropriate topics. A sensor is defined for each metric produced by a managed resource and this sensor subscribes to the topic defined by the WSDM endpoint for the metric of interest.

In our sample scenario, the concrete events from the managed resource are provided by the Management Interface, which publishes on two topics, namely the RejectRateTopic and the WLIntensityTopic. The reject rate metrics are published to the RejectRateTopic and are consumed by the Reject Rate Sensor. The throughput metrics are published to the WLIntensityTopic and are consumed by the Workload Intensity Sensor.

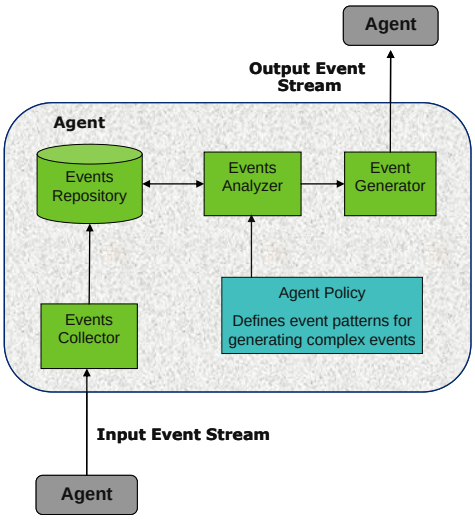

Figure 2: Agent Architecture

All agents are constructed in an identical fashion following the architecture shown in Figure 2. Each agent is implemented as a WSDM entity which enables the agents to communicate using standard protocols and, in particular, provides publish/subscribe messaging capabilities. The defining feature of an agent is the policy that specifies its behaviour. The policy contains the set of input streams, the output stream, a set of patterns, and for an actor, a management function that is triggered when the specified pattern is matched. The input streams are implemented by the topics to which the agent subscribes. The output stream is implemented by a topic to which the agent publishes event notifications. Thus, for our example, the input stream is the topic to which the Workload Intensity Sensor subscribes, namely WLIntensityTopic, and the output stream for the Workload Intensity Sensor is the topic to which it publishes, namely WLIncreaseTopic. Events published to WLIncreaseTopic are inferred events in our model.

Incoming event instances are gathered by the agent's *Events Collector* and saved in the original XML format in a repository, which is implemented by a DB2 database that is unique to each sensor. The events repository represents the events history and is used for the event context of an event instance.

Insertion of a new event in the repository triggers the *Events Analyzer* which evaluates the new event against the set of patterns specified in the policy. The pattern consists of the event type and a context. The context in our implementation is specified using XQuery, a standard language for querying XML data. The XQuery returns a set of "matches" if the pattern has been matched. A null set is returned if no matches are found, thus indicating that the pattern has not been matched. If the pattern is matched, an event notification is generated by the *Events Generator* and published to the appropriate topic, notifying subscribers of the event.

In our scenario, for example, the pattern for the Reject Rate Sensor looks for increases in the reject rate over the past 10 seconds whenever a RREvent is received. The pattern specified for this sensor would be as follows:

(RREvent,
xquery declare namespace muws1="http://docs.oasis-open.org/wsdm/muws1-2.xsd";declare namespace muws2="http://docs.oasis-open.org/wsdm/muws2-2.xsd" ; for $a in db2-fn:xmlcolumn('RR.EVENT')/muws1:ManagementEvent, $b in db2-fn:xmlcolumn('RR.EVENT')/muws1:ManagementEvent where xs:dateTime(fn:string($b/@ReportTime)) > xs:dateTime(fn:string($a/@ReportTime)) and xs:dateTime(fn:string($b/@ReportTime)) < xs:dateTime(fn:string($a/@ReportTime)) + xdt:dayTimeDuration("PT10S") and

fn:number($b/muws2:Situation/muws2:Message) > fn:number($a/muws2:Situation/muws2:Message) return <x>{$a/muws1:EventId/text()}, {$b/muws1:EventId/text()}</x>)

Actors contain a management function as part of their policy. The management functions are implemented as stored procedures and are called using the XQuery specified in the pattern portion of the policy. A stored procedure is an external subroutine (usually written in Java or C) that is available to database applications, in our case, through the XQuery. The stored procedure is used to consolidate, compartmentalize, and externalize the logic for the actors. Stored procedures may involve SQL statements, but they are not limited to database functionality. For instance, in our sample scenario, the management function for the Problem Analyzer Actor evaluates the potential benefit of adding additional threads to service Web server requests. Adding additional threads means that more clients are contending for the same hardware resources, thus, not necessarily improving performance. The function that is implemented for the Problem Analyzer Actor predicts overall system performance using different system configurations and determines whether or not additional threads will be beneficial. If so, a message is placed on the actor's outgoing stream.

Effectors impose change for the management system. Effectors are implemented using the same architecture as sensors and actors and are implemented as WSDM endpoints. Although they have an input stream and a pattern, the context is often null (that is, it is enough that an event instance is received to trigger the effector's action). The action of an effector is the communication with the management endpoint for the managed resource. The effector usually makes a call to adjust one or more configuration parameters or to take some management action on the managed resource via the management capabilities provided by the resource. One effector is created for each manageability capability of the managed resource, that is, each management function provided by a managed resource. For example, to change the number of threads of the Web server in our example, the effector would call the "SetNumThreads" operation defined by the WSDM endpoint for the Web server.

A graphical tool to define, view and modify management goal graphs is shown in Figure 3. This tool allows users to define agents, patterns and management functions, as well as defining the communications paths of the agents. Figure 3 shows the management goal graph for our sample management scenario. The graph is depicted in the left pane. The user may click on an agent's icon to view details about the agent (shown in the rightmost panes) or to modify the agent's details, including its policy, or to delete the agent.

New agents can be defined using the "Add Sensor" wizard shown in Figure 4. The first tab of the labeled "Sensor" provides fields to define the name of the agent and to specify the location of the agent. In our implementation agents are defined as WSDM entities, thus, the location is expressed as a URI.

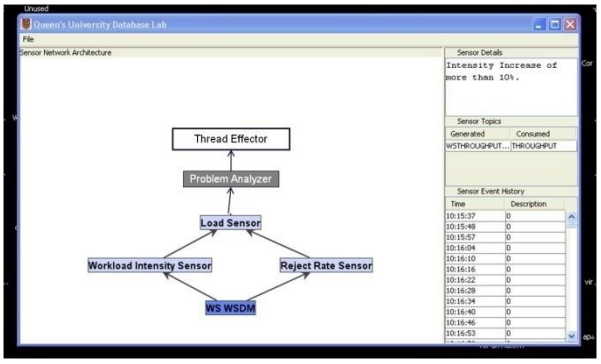

Figure 3: Management Goal Graph Tool

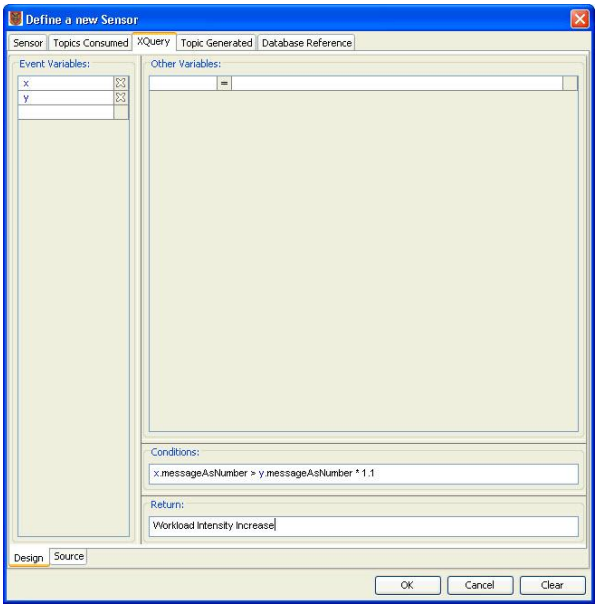

Figure 4: XQuery Definition in Design Mode

Using the tabs labeled "Topics Consumed" and "Topics Produced" the user specifies the input and output streams of the agent. Input streams are chosen from the list of topics generated by existing agents, thus forming connections between agents.

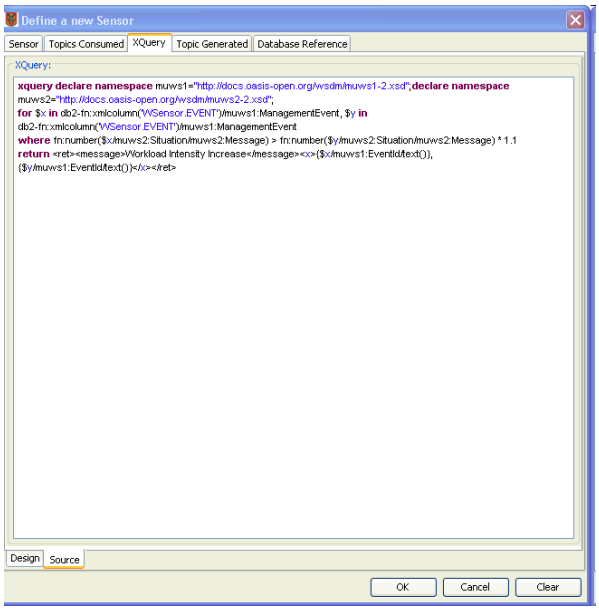

Figure 5: XQuery Source

The tool also reduces the complexity of writing XQuery requests by providing a template for the creation of several frequently used query types. Figure 4 shows the XQuery tab in "Design" mode. The pattern defined in this case identifies increases in the reported value of 10%. Figure 5 shows the automatically generated XQuery.

Our implementation uses a number of tools including Eclipse TPTP [7], Apache Muse [2] and IBM DB2 [6].

6. Conclusions and Future Work

The growing popularity of Service-Oriented Architecture (SOA), which allows applications to be constructed of existing services in a dynamic manner, introduces new challenges to systems management and requires a reexamination of traditional management architectures. We are developing a new framework for services management that will consist of three key components, namely a services management model, a method to generate management system components from model specifications and an infrastructure to allow the integration of management tasks and user interaction with the management system.

In this paper we describe a new services management model. The model provides a method for describing the autonomic management of a service and its components in terms of agents, event streams and policies. Individual agents are combined into a network of cooperating agents in the form of a management goal graph. We show the viability of the model with an approach to implementing the model specifications based on Web Service Distributed Management. We find that the constructs in the model map well to the capabilities of WSDM. We plan to investigate to what degree this approach can be automated in the future.

The construction and integration of management goal graphs will be the primary activities in this approach to managing services. The sources of a goal graph will always be sensors accepting concrete events from a managed resource. The sinks will be effectors applying changes to configuration parameters. A key question is, given a management goal, how do we construct the graph? One possible approach would be to borrow from event trees and graphs [1][3] in the work on event algebras. The context description for the final goal condition is expressed, and from this, a graph is formed by making every operator in the statement a node in the graph. When we create a new management goal graph we must determine how it integrates with existing graphs. Two graphs may share nodes so these become the integration points. Nodes from both graphs can subscribe to the event stream published by the common node. The graph produced by the integration of the management goal graphs is the *management system graph*.

We will consider a number of other issues in the future. We want to provide a method for the formal definition and verification of the management function introduced in Section 4.3. We plan to map our model to policy-based management [16] components. This requires using an efficient representation language for the policies defined in the agents, and specifying the types of policies that our model can handle. We also will consider the problem of policy conflicts when we attempt to integrate management goal graphs.

7. References

[1] Adi, A. and O. Etzion, "Amit - The Situation Manager". The VLDB Journal, 2004. **13**(2): p. 177-203.

[2] *Apache Muse*. 2007, Apache Software Foundation,; April 25 2008; http://ws.apache.org/muse.

[3] S. Chakravarthy and D. Mishra. "Snoop: An Expressive Event Specification Language for Active Databases", *Data and Knowledge Engineering 14(1)*, 1994, pp. 1 – 26.

[4] Cohen, N.H. and K.T. Kalleberg, "EventScript: An Event-Processing Language Based on Regular Expressions with Actions", *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 2008, ACM: Tucson, AZ, USA.

[5] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef: "Web Services on Demand: WSLA-driven Automated Management". *IBM Systems Journal*, Vol.43(1), pp. 136–158 (2004).

[6] *DB2 9*. 2007, IBM; July 2 2008, http://www-306.ibm.com/software/data/db2/9/.

[7] *Eclipse TPTP*. 2008, The Eclipse Foundation; July 2 2008, http://www.eclipse.org/tptp/.

[8] IBM. *E-business Management Services*. Available from: http://www-935.ibm.com/services/us/igs/.

[9] H. Kelash, H. Faheem and M. Amoon. "A Multiagent System for Distributed Systems Management", *Proceedings of World Academy of Science, Engineering and Technology Volume 11*, February 2006, 91 – 96.

[10] J. Kephart and D. Chess. "The Vision of Autonomic Computing", *IEEE Computer*, 36(1), 2003, 41 – 52.

[11] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef: "Performance Management for Cluster-Based Web Services", In *Proc. of IFIP/IEEE Int. Symposium on Integrated Network Management (IM'03)*, Colorado Springs, USA, pp. 247-261 (2003).

[12] P. Martin, W. Powley, W. Tian, K. Wilson, J. Zebedee and Z. Xu. "The WSDM of Autonomic Computing: Experiences in Implementing Autonomic Web Services". *Proc of ICSE Workshops SEAMS '07 (International Workshop on Software Engineering for Adaptive and Self-Managing Systems),* Minneapolis MN, May 2007, pp. 9 -16.

[13] Rubio-Loyola, J. Serrat, J. Charalambides, M. Flegkas, P. Pavlou, G. and Lafuente, A.L., "Using Linear Temporal Model Checking for Goal-Oriented Policy Refinement Frameworks," *Policies for Distributed Systems and Networks, 2005. Sixth IEEE International Workshop* , pp. 181-190, 6-8 June 2005.

[14] A. Sahai, V. Machiraju, M. Sayal, A. Van Moorsel, and F. Casati: "Automated SLA Monitoring for Web Services", In *Proc. of the IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management (DSOM'02),* Montreal, Canada. LNCS, Springer, Vol. 2506, pp. 28-41 (2002).

[15] A. Sahai, V. Machiraju, and K. Wurster: "Monitoring and Controlling Internet Based Services", In *Proc. of IEEE Workshop on Internet Applications (WIAPP)*, San Jose, CA (2001).

[16] D. Verma, Simplifying network administration using policy-based management. *IEEE Network*, **16**(2): p. 20-26, 2002.

[17] *WSDM v1.1*. 2008, Organization for the Advancement of Structured Information Standards; May 18 2008, http://www.oasis-open.org/specs/index.php#wsdmv1.1.

[18] *WS Notification*. 2004, Organization for the Advancement of Structured Information Standards; July 2 2008; http://www.oasis-open.org/specs/index.php#wsnv1.3.