Use Case Driven Approach to Self-Monitoring in Autonomic Systems

A.R. Haydarlou, M.A. Oey, B.J. Overeinder, and F.M.T. Brazier
Vrije Universiteit Amsterdam, De Boelelaan 1081a,
1081 HV Amsterdam, The Netherlands
E-mail: {rezahay,michel,bjo,frances}@cs.vu.nl

Abstract

Self-monitoring of autonomic distributed systems requires knowledge of the states and events of many different parts of a system. One of the main challenges is to determine which information is most crucial for analysis of a system's behaviour, and when. This paper proposes a model-based approach to self-monitoring for which structural and behavioural models of a system are described at different levels: application, subsystem, component and class level. In this approach, a system's behaviour is monitored in the context of a hierarchy of use-cases related to these levels. The structural and behavioural models are used to automatically instrument an existing distributed system. The proposed architecture of a self-monitoring engine is described as is the implementation. The models have been specified in the Ontology Web Language (OWL) and the self-monitoring (as a part of our self-management framework) has been implemented in Java. The scenario used to illustrate the approach is that of authentication for a simplified version of a distributed portal application.

1. Introduction

Self-management is central to autonomic computing, encompassing various self-* aspects, including self-configuring, self-healing, and self-optimising [13]. Self-management requires *self-diagnosis* to analyse a problem situation and to determine a diagnosis, and *self-adaptation* to repair the faults discovered. The ability of a system to perform adequate self-diagnosis depends largely on the quality and quantity of its knowledge of its current state.

This paper focuses on *self-monitoring*: the process of *obtaining knowledge* through a collection of *sensors* instrumented within a self-managed (i.e., autonomic) system. Note that self-monitoring is not responsible for diagnostic reasoning or adaptation tasks. One of the main challenges of self-monitoring is to determine which information is most crucial for analysis of a system's behaviour, and when.

The self-monitoring approach presented in this paper proposes to base the choice of when and where to place sensors on use-cases. *Use-cases* describe the behaviour of a system by specifying the response of the system to a given request and the tasks involved. These tasks are used to structure monitoring data acquisition: to determine where to place sensors to monitor system behaviour as described by a use-case. Monitoring a use-case involves monitoring *state changes* and *event occurrences* during the realisation of the use-case. This paper introduces three levels of use-cases: operational, functional, and implementational. These levels correspond with the views of software domain experts (e.g., system administrators, functional analysts, and software developers).

Two models of the self-managed system are distinguished: a *structural* model and a *behavioural* model. The former model describes the internal structural of a system; the latter model describes its dynamic behaviour (i.e., the use-cases). These models facilitate: (1) recognition and classification of the different sensor types, (2) determination of important observation points, and (3) automated instrumentation of sensors. The proposed approach structures the self-monitoring process such that it only provides monitoring data that can be directly related to execution of the tasks described in the use cases.

This paper is organised as follows. Section 2 describes an authentication scenario regarding a simplified portal application used to illustrate the concepts employed within this paper. Section 3 presents the notions of structural and behavioural models. Section 4 introduces sensors and observation points. Section 5 presents an architecture for self-monitoring and describes an implementation based on this architecture. The final section, Section 6, discusses related work.

2. An Authentication Scenario

To illustrate the types of problems for which the use-case based approach to self-monitoring has been designed, this

section presents an authentication scenario for a simplified version of a complex portal application.[1] Possible causes of failure at the different levels distinguished in our model are described, as are the types of observations needed to identify such causes.

Most enterprises, including Fortis Bank Netherlands, use the scenario, described below, to authenticate business users requesting access to the company's secure portal applications. Note that portal applications typically integrate a number of legacy applications, presenting them on the web as a single application. Consequently, the authentication logic for the application as a whole is usually spread out and embedded in different subsystems of the portal application including legacy subsystems.

Figure 1 visualises this application. To access a portal application, business users provide their certificates to the *AccessManager* subsystem using a *HTTPS* negotiation process, established between the user's browser and *AccessManager*. Upon receiving the user's certificate, the *AccessManager* verifies the certificate, and passes it to the *BusinessIntegrator* subsystem using a *JRMI* connection. The *BusinessIntegrator* communicates with the *DatabaseManager* subsystem via a *JDBC* connection, extracts the user's identity (userid), gets the password for the given userid, constructs login information (userid/password), and sends it to the *BusinessManager* subsystem (legacy back-end) using a *SOAP* connection. The *BusinessManager* authenticates the user and returns the result of the authentication to the *BusinessIntegrator*. Finally, the *BusinessIntegrator* passes the result of the authentication through the *AccessManager* back to the browser.

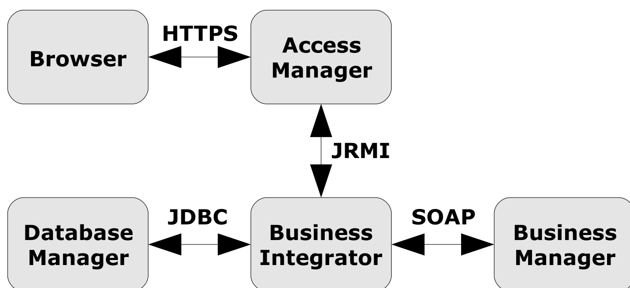
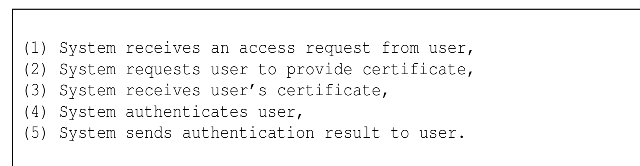


Figure 1. Authentication example.

This description of the application assumes correct behaviour. Suppose, however, that an error occurs: a user is denied access even though he/she has a valid certificate and identity. The root-cause of this malfunctioning needs to be discovered. It might be somewhere in the system code, or it may not. It is obvious that pinpointing such a root-cause in a system with thousands of lines of code is not an easy task. The approach proposed in this paper is to distinguish a hierarchy of levels in the structure (see Section 3.1) and the behaviour (see Section 3.2) of a system on the basis of use-case descriptions.

On the highest level, the portal application is composed of a number of communicating subsystems, in this paper referred to as runnables. On the next lower level, each subsystem is composed of a number of components, and finally, each component is composed of a number of classes and methods.

The highest level in the hierarchy of the application behaviour is described by the *application level use-case*. Figure 2 shows the authentication process from the viewpoint of a user, described as an application level use-case. From the viewpoint of a user the application is, in fact, a black box. The application level use-case specifies the interaction the user has with the application. The user–application interactions provide the starting point for self-monitoring.



```
(1) System receives an access request from user,
(2) System requests user to provide certificate,
(3) System receives user's certificate,
(4) System authenticates user,
(5) System sends authentication result to user.
```

Figure 2. Application level use-case.

The three remaining levels of use-case descriptions are described in the following sections together with illustrations of level-related errors and their root-causes. The sensors needed to detect such errors (i.e., to monitor specific states and events) are defined. More specific information about positioning and instrumentation of sensors, localisation of observation points, and information transferred are explained in Sections 3 and 4.

2.1. Runnable Level Monitoring

Figure 3 shows the authentication process from the viewpoint of a system administrator, represented in a *runnable level use-case*. From this viewpoint, only runnables and their connections are of importance. The use-case thus describes the authentication process as interactions between runnables. Note that each runnable may implement one or more use-case tasks. Examples of errors that can occur at the level of runnables include: a broken connection, an incorrect start-up sequence of runnables, excessive heap usage. For the sake of brevity, in this example, only tasks (3) and (4) in Figure 3 are assumed to be monitored. An error in the execution of task (3) may be caused by a broken connection between *AccessManager* and *BusinessIntegrator*. This connection is modelled as a *connector* that contains information on the type of protocol used (in this case *JRMI*), and

[1] In this article, the terms *system* and *application* are used interchangeably.

a *connector state* indicating the status of the connection.

```
(1) AccessManager receives user's certificate,
(2) AccessManager verifies certificate,
(3) AccessManager passes certificate to BusinessIntegrator,
(4) BusinessIntegrator prepares login info,
(5) BusinessIntegrator delegates login info to BusinessManager,
(6) BusinessManager authenticates the user,
(7) BusinessManager passes result to BusinessIntegrator,
(8) BusinessIntegrator passes result to AccessManager,
(9) AccessManager passes authentication result to Browser.
```

Figure 3. Runnable level use-case

Monitoring task (3) is implemented as follows. Based on the information in the connector, the self-monitoring engine generates a code fragment that uses the *JRMI* protocol to establish a connection with the *BusinessIntegrator*. This code can be used to check whether a connection to the *BusinessIntegrator* is functioning correctly. In addition, a sensor is created that includes a call to this code fragment. This sensor is automatically instrumented in the code of the *AccessManager*, just before the location of the code that performs task (3). During runtime, the instrumented sensor code is executed and the value of the connector state is determined.

Task (4) in Figure 3 is an invocation of the component level use-case specified in Figure 4. The output state of this invocation indicates whether the invoked use-case (authentication preparation) was successful (or not). To monitor the output state of the invocation, the self-monitoring engine generates a sensor that watches the result of the invocation. This sensor is automatically instrumented in the code of the *BusinessIntegrator* just after the occurrence of the invocation code.

2.2. Component Level Monitoring

Figure 4 shows the authentication process from the viewpoint of a functional analyst, represented in a *component level use-case*. From this viewpoint, only components and their interactions are of importance. Each component performs one or more use-case tasks. Assume that the runnable *BusinessIntegrator* contains two components: the *CertificateParserComp*, which parses a certificate and extracts a user identity, and the *PrepareAuthComp*, which constructs login information. For this use-case, tasks (1) and (5) are monitored.

One of the aspects that needs to be monitored is whether the components are compatible: whether the version numbers of the components correspond. It is assumed that the interface file of *CertificateParserComp* has a state that is modelled as a *component state*. This state represents the current version of *CertificateParserComp*. The self-monitoring engine generates a specific sensor to monitor the component version, and it is automatically instrumented in the code of *CertificateParserComp* (in the interface file), at the beginning of the method realising task (1) in Figure 4.

```
(1) CertificateParserComp receives certificate,
(2) CertificateParserComp extracts userid,
(3) CertificateParserComp passes userid to PrepareAuthComp,
(4) PrepareAuthComp receives userid,
(5) PrepareAuthComp prepares login info,
(6) PrepareAuthComp returns login info.
```

Figure 4. Component level use-case

Task (5) in Figure 4 is an invocation of the class level use-case specified in Figure 5. Monitoring of this task is similar to monitoring task (4) in Figure 3.

2.3. Class Level Monitoring

Figure 5 shows the authentication process from the viewpoint of a system developer, represented in a *class level use-case*. From this viewpoint, only classes, methods, and their interactions are of importance. Note that each class or method may perform one or more use-case tasks. Assume that the *PrepareAuthComp* component contains a class called *AuthInfoClass*, which establishes a connection with a database and retrieves authentication information from it. This class contains a method called *RetrieveAuthInfo* which retrieves authentication information from the database. One of the errors that can occur is that the *DatabaseManager* provides incorrect information about a user's identity. This error could, for example, occur when the scheme for specifying userids changes without the related information in the database being changed. Moreover, the same problem could also cause a *NullPointerException* during the construction of the login info. Task (1) is monitored to see whether the value of userid satisfies a specific scheme. In this case, the self-monitoring engine generates a sensor that watches the parameter state (userid) of the constructor method of *AuthInfoClass*. This sensor is automatically instrumented in the code of the constructor method of *AuthInfoClass*, at the beginning of the constructor code.

```
(1) AuthInfoClass receives userid,
(2) AuthInfoClass establishes a database connection,
(3) RetrieveAuthInfo requests password from database,
(4) RetrieveAuthInfo constructs login info,
(5) RetrieveAuthInfo returns login info.
```

Figure 5. Class level use-case

To monitor an event occurrence (the *NullPointerException*), the self-monitoring engine generates a sensor that

gathers information (such as timestamp, stack trace, file name, line number, etc.) about the exception. In addition, the self-monitoring engine automatically instruments a *try-catch* block around the location of the code realising task (4) in Figure 5, which belongs to the *RetrieveAuthInfo*. The *catch* part contains a call to the generated sensor. Consequently, during execution of task (4), the *NullPointerException* is identified and the information about the exception is reported to the self-diagnosis unit.

The scenario described above shows that applications can be observed at different levels: application, runnable, component, and class level. Monitoring an application at these levels, allows information to be related to use-case descriptions at these levels. The monitoring information in context provides both information in relation to the application's software architecture and the application's behaviour.

3. Application Model

The self-monitoring approach proposed in this paper is based on a model of distributed object-oriented applications. The model provides structure not only for self-monitoring but also self-diagnosis. As an important requirement for model-based approaches is a component[2]-oriented system model [17], the application model is analysed to its (i) structural components and their relationships, and (ii) behavioural components and their relationships. The model's behavioural components are use-cases (called *jobs*) and use-case steps (called *tasks*). The model's structural components are runnables, (software) components, classes, and methods that act as containers where the behavioural components are executed and where sensors are instrumented.

Figure 6 shows how the structural components, i.e., runnables, components, and classes, relate to each other, and how the behavioural components, i.e., jobs and tasks, flow through the structural components. The input of a runnable level use-case is used by different tasks executing within runnables in a specific order. One of these tasks (*RunTask 1*) is a job invocation task that invokes the component level use-case. The two components shown at this level in the figure are structural parts of *Runnable 1*. Similarly, multiple tasks are executed within the components in a specific order, one of which (*CompTask 1*), invokes the class level use-case. Moreover, an invocation task at one level can invoke another use-case at the same level or at a higher level (not depicted for the sake of simplicity). For example, starting an executable program (runnable) from inside a class can be modelled as an invocation to a higher level use-case.

Acquiring domain knowledge necessary for model-based approaches is known to be difficult [17, 8]. In the application model presented in this paper, the domain knowl-

[2]Here the term component is used in the style of model-based diagnosis and it refers to the smallest diagnosable unit.

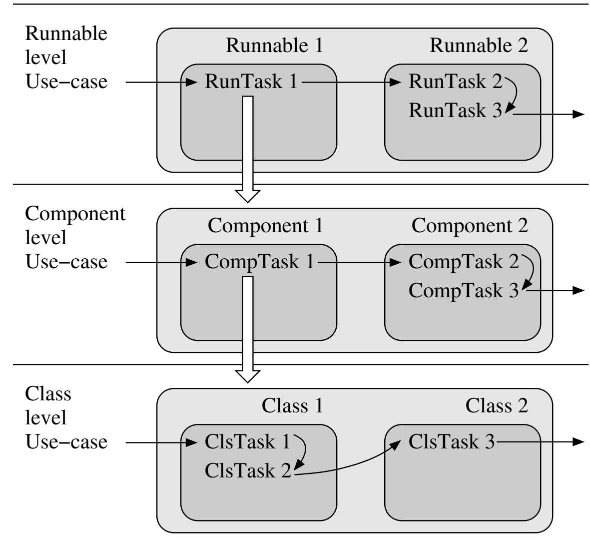


Figure 6. Use-case levels in an application.

edge is obtained from three domain experts, each with his/her own specific view to the system: *system administrators*, *functional analysts*, and *software developers* [11, 14]. To facilitate knowledge acquisition, the views of these domain experts are incorporated in both structural and behavioural components of the application model. The following sections describe both models in more detail.

3.1. Structural Model

The structural model of an application describes the software architecture of the distributed object-oriented system. Distributed object-oriented systems are usually composed of one or more subsystems, each of which is composed of a number of components. Components either contain other (sub)components or a number of classes. Classes may contain other (inner)classes and a number of methods.

The common properties of these different structural elements have been modelled as a *managed element*. There are six different types of managed elements: system, runnable, component, class, method, and connector. A *managed system* is used to monitor the collective behaviour of a number of related subsystems. It is composed of a number of runnables. A *managed runnable* models a part of a system that can be started/stopped (such as a subsystem or an execution thread).

Runnables can be connected with each other by means of a *managed connector* that models the connection between two subsystems communicating with each other using a specific protocol. A runnable is composed of one or more runnables or components.

A *managed component* models a software component or library (such as a logging component). It contains one

or more dedicated methods (*interface methods*) as its entry points, and is composed of one or more components or classes.

A *managed class* in the model contains a dedicated method (*constructor method*) as its entry point. Similar to the recursive definitions of runnables and components, a managed class can be composed of one or more classes or methods.

Finally, a *managed method* is an atomic managed element that corresponds with a coding-level method and models the method-parameters, local variables, and method-body.

The above structural decomposition of distributed object-oriented systems with the hierarchical relationship between different managed elements provides viable granularity levels concerning monitoring information for a hierarchical model-based diagnosis approach [16].

3.2. Behavioural Model

To understand the complex behaviour of a large distributed system, the system's behaviour is described by a collection of use-cases. Each use-case describes the response of the system to a given request. Note that an individual use-case may include calls to other use-cases. The system provides its response by executing a number of actions (tasks) that are realised within managed elements. During the execution of an action, a number of state changes, and internal or external events may occur.

The behavioural model consists of the following concepts: jobs, tasks, states, and events. A job represents a use-case. Corresponding to the three types of domain-expert views mentioned above, there are three types of jobs: an operational job, a functional job, and an implementational job. These job types correspond with the runnable level, component level, and class level use-cases, respectively.

An operational job describes a runnable level use-case realisation from the viewpoint of a system administrator. According to this view, a system is composed of a number of processes and threads (runnables in the structural model) that cooperate with each other to realise a use-case. A functional job describes a component level use-case realisation from the viewpoint of a functional analyst. According to this view, a system is composed of a number of functional components (components in the structural model) each of which is responsible for realising a specific part of some business functionality. An implementational job describes a class level use-case realisation from the viewpoint of a system developer. According to this view, a system is composed of the low level code bundled in classes (classes in the structural model).

Each job consists of one or more tasks that represent system actions and are basic units for monitoring. The model only supports execution of each individual task by one managed element (modelled in the structural model) to help the self-diagnosis unit to pinpoint a specific managed element as the location of a malfunctioning. To facilitate the specification of knowledge in the behavioural model, tasks have been categorised into tasks such as: state manipulation tasks, invocation tasks. For more information see [11].

State manipulation tasks are closely related to the notion of *state* which represents the assignment of a value to a variable. The monitoring engine needs to provide information about the origin of the states it is monitoring. Based on the origin of states, the following state types are identified in the behavioural model: the job input state describing the parameters of a job; the state belonging to the result of a job; the state originating from data sources such as a database, file, queue, topic, user interface, etc.; the state of a managed element, e.g., the state of a runnable, component, connector, and class; and the state of a method (parameters, local variables).

Invocation tasks represent the invocation of a job in the behavioural model. Switching from one job type to the other one (e.g., an invocation from an operational job to a functional job) are explicitly modelled in the model.

An *event* models expected and unexpected behaviour during the execution of system actions (tasks). Events have been categorised as follows: runnable startup event, runnable shutdown event, job startup event, job invocation event, user request event, system request event, timer expiration event, and exception event that causes the normal execution of a system action to terminate. To allow the self-diagnosis unit to perform hierarchical diagnosis, each job has been associated with two events that indicate the start and the end of the job, respectively.

4. Automated Instrumentation of Sensors

Recall that the main responsibility of a self-monitoring engine is to provide knowledge about the current state of the monitored system to the self-diagnosis unit. This knowledge is captured during the execution of the monitored system by observing all state changes, event occurrences, interactions with the environment, and communications between subsystems. These observations are passed on to the self-diagnosis unit, together with the following meta-information about each observation: (i) the type of the observation, (ii) when the observation was captured, and (iii) where in the system the observation was captured.

The proposed self-monitoring approach uses use-cases to determine where to place sensors. The structural and behavioural models (see Section 3) help to place sensors only on the places where they are needed to aid proper diagnosis. Moreover, the models also provide the meta-information for each observation, which is explained below.

Classifying Observations Two types of sensors are distinguished: *state sensors* and *event sensors*. Each sensor provides information on the value of an observation, its type, and a number of additional attributes. Both sensor types are further subdivided into a hierarchy of sensor types that corresponds to the hierarchy of states and events in the behavioural model.

Determining the Moment of an Observation Both qualitative and quantitative temporal information are provided. The qualitative temporal information refers to the specification of job and task in relation to other jobs and tasks: the chronological order of a job execution in a hierarchy of jobs, and the chronological order of a task execution in the context of a job. The quantitative temporal information refers to the actual timestamp of the observation.

Localising Observations As sensors are associated with tasks, and tasks execute within exactly one managed element, spatial information about an observation is part of the monitoring information derived. This information also facilitates automatic instrumentation of the sensors in the appropriate managed element.

4.1. State Sensor Instrumentation

A task in the behavioural model is specified in terms of its input and output states. For automated state sensor instrumentation, a distinction is made between *concrete* state types specified in the model that correspond with specific states in the code of the system, and *abstract* state types specified in the model that has no counterpart in the application code.

If a task input or output in a use-case is of a concrete state type, then the self-monitoring engine locates the managed element associated with the task, inspects the task's code to discover the corresponding state, and finally instruments the proper state sensor around the occurrence of the state. These state sensors are called *system provided* state sensors.

If a task input or output in a use-case is of an abstract state type, a domain expert is responsible for providing a code fragment to retrieve the runtime value of the abstract state. With the instrumentation of the state sensor in the code of the managed element associated with the task, a call to the user provided code fragment is included. These state sensors are called *user provided* state sensors. Note that both system provided and user provided state sensors are instrumented automatically.

4.2. Event Sensor Instrumentation

Event sensors are used for three main purposes: (i) to indicate which tasks are executing in the context of which use-cases, (ii) to perform periodically or incidentally inspection of some important parts of the system, and (iii) to determine which task has aberrantly terminated the normal execution thread of a use-case. In the following, the event sensors related to the invocation, timer expiration, and exception events (see Section 3.2) are explained.

An invocation event is attached to an invocation task, which indicates a call to another use-case. The self-monitoring engine locates the managed element associated with the invocation task, inspects the code to discover the corresponding call statement, and finally instruments the proper *invocation sensor* before (to indicate the start of the use-case) and after (to indicate the end of the use-case) occurrence of the call statement.

A timer expiration event is attached to the task that inspects the status of a connector between two runnables. The self-monitoring engine has code fragments for a timer and for checking the status of a connector with a specific protocol. The domain expert specifies the properties of the timer (such as time interval), the connector (such as protocol), and the runnables (such as host and port) in the model. Based on this information, a *timer expiration sensor* is instrumented in the bootstrap code of the self-monitoring engine.

An exception event is attached to tasks that can cause an exception during their execution. In this case, the self-monitoring engine looks for the managed element associated with the task, inspects the code to discover the corresponding task, and finally instruments exception handling code (a *try-catch* block containing a call to an *exception sensor*) around the task. Note that all of these sensors are instrumented automatically.

5. Self-Monitoring Architecture

Figure 7 shows the architecture of the proposed self-monitoring approach. At the top of the figure are descriptions of the structure of the monitored system and the use-cases that describe the behaviour of that system. The Semantic Web languages OWL [5] and SWRL [12] are used to model the system structure and the use-cases [10]. The resulting OWL-specification contains both the structural and the behavioural model of the monitored system.

An OWL parser parses this OWL-specification and extracts the information on all sensors defined in the specification needed to monitor the use-cases *(Sensor Specifications*). Furthermore, the parser also generates a library containing all the code needed to receive and process monitoring sensor information. This library includes all information on jobs, tasks, the structure of managed elements, etc., as specified in the OWL-specification. Sensors from the OWL-specification can be automatically instrumented in the code of the system, as described above. The resulting instrumented system uses the monitoring library to

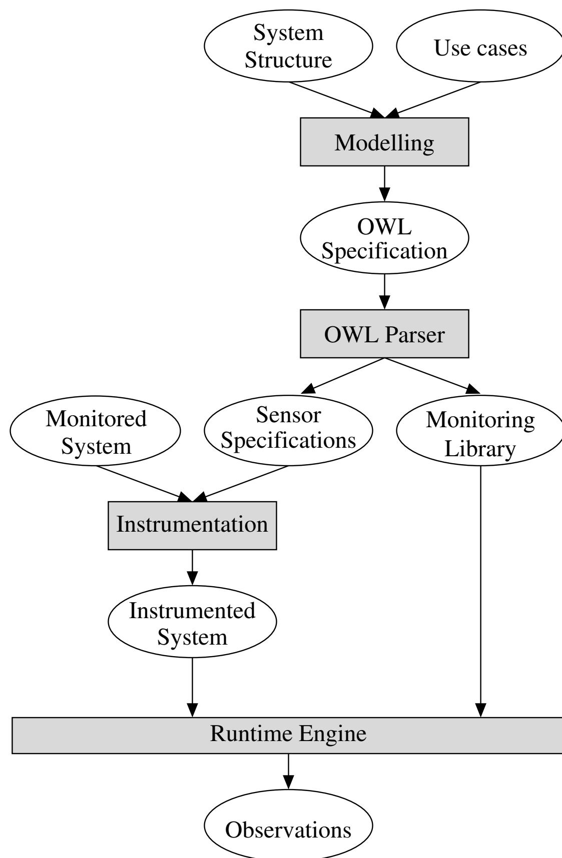


Figure 7. Self-monitoring architecture

send runtime observations to the self-diagnosis unit (not depicted) to diagnose any malfunction.

A Java version of the self-monitoring architecture shown in Figure 7 has been implemented and evaluated.[3] The OWL parser uses the *Protege OWL Java API* [18] to parse the OWL specifications that describe the structure and behaviour of the monitored system. From the OWL specification, the parser generates a monitoring library in Java. The Sensor Specifications, which are also generated from the OWL-documents, contain information for each sensor, including the name of the monitored item (state or event), the type of the task manipulating that monitored item, the name of the Java class containing that monitored item, and the location of the monitored item in the Java class. With this information, the sensors can be automatically instrumented in the code of the monitored system.

Automatic sensor instrumentation is done using *Javassist* [7], which uses the *Aspect Oriented Programming* paradigm to add functionality to an existing system by directly altering its byte-code. To instrument a sensor, first the line of (byte-)code where the monitored item occurs is located. Next, new code is inserted just before and/or after that line depending on the task type. This code, the sensor, will send the before-value and/or after-value of the monitored item to the diagnosis unit during execution. The sensor communicates with the diagnosis unit using *Apache ActiveMQ* [1], an open source Message Broker.

[3] The source code can be downloaded from `http://www.iids.org/research/self-management/self-healing-framework.zip`.

6. Discussion

The self-monitoring approach, discussed in this paper, is part of our model-based framework for self-management of distributed object-oriented systems in which the three important modules, i.e., self-monitoring, self-diagnosis, and self-adaptation share the same model. This model contains both structural and behavioural aspects of a system, and is specified formally in OWL. By associating sensors with behavioural components in the model, sensors become an integral part of the model, supporting automated sensor instrumentation. Because of the different use-case levels distinguished in the model, all state changes and event occurrences are directly related to execution of the tasks described in the relevant use-case level. Below, the related work regarding system model, sensor types, and sensor instrumentation is discussed.

Garlan et al. [9, 6] present a self-adaptation architecture based on a system's *architectural model*. This model consists of computational elements and connectors which are annotated with various properties using an *Architectural Description Language (ADL)*. The computational elements, connectors, and probes are comparable to runnables, connectors, and sensors; the ADL specifications comparable to runnable level operational specifications in OWL. Garlan et al., however, do not monitor lower level constituents such as components and/or classes, nor do they provide information on the context within which an error has occurred. The result is less fine grained monitoring information.

Ardissono et al. [2, 3] propose a model-based approach for self-diagnosing complex Web Services. These complex services are composed of remotely cooperating Web Services orchestrated by a WS-BPEL engine. Little is said about acquisition of monitoring information, other than that monitoring occurs locally at the level of runnables.

Baresi et al. [4] propose an external *monitoring definition file* for monitoring the execution of a WS-BPEL process. The file contains general information regarding the WS-BPEL process, and *monitoring rules*. A monitoring rule consists of *monitoring location*, *monitoring parameters*, and *monitoring expressions*, and specifies the point of sensor instrumentation in the process definition. This monitoring approach again focuses solely on Web Services. Although this approach is not model-based, it shows many similarities to our approach with respect to sensor types,

sensor-activity relationship, and sensor instrumentation at the level of runnables (Web Services).

The self-monitoring approach described in this paper, and implemented for evaluation purposes supports not only monitoring of Web Services but also legacy systems, on the basis of multi-level use-cases and automated instrumentation of these applications. Related to the structural model, a complex Web Service composed of two or more Web Services is modelled as a *composite managed runnable* containing two or more *managed runnables*, and each Web Service operation is modelled as a *managed method*. The dynamic behaviour of a Web Service can be directly described in OWL as jobs and tasks. The Semantic Web community has proposed a Web Service ontology (*OWL-S* [15]) which is written in OWL and describes the Web Service profile and its dynamic behaviour. We believe it would not be difficult to automatically convert the knowledge in the OWL-S description of a Web Service to the knowledge needed by the proposed approach in this paper. Further research shall focus on the use of monitoring information in diagnosis.

Acknowledgements

This research is supported by the NLnet Foundation, `http://www.nlnet.nl`, and Fortis Bank Netherlands, `http://www.fortisbank.nl`, for which the authors are grateful.

References

[1] Apache Software Foundation. Apache activemq. `http://activemq.apache.org/home.html`, 2005.

[2] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. T. Dupré. Enhancing web services with diagnostic capabilities. In *ECOWS*, pages 182–191, 2005.

[3] L. Ardissono, R. Furnari, A. Goy, G. Petrone, and M. Segnan. Fault tolerant web service orchestration by means of diagnosis. In *EWSA*, pages 2–16, 2006.

[4] L. Baresi and S. Guinea. Towards dynamic monitoring of ws-bpel processes. In *ICSOC: Proceedings of the 3rd Int. Conference on Service Oriented Computing*, pages 269–282, 2005.

[5] S. Bechhofer, F. Harmelen, J. A. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. `http://www.w3.org/TR/owl-ref`, 2004.

[6] S.-W. Cheng, A.-C. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *ICAC*, pages 276–277, 2004.

[7] S. Chiba. `http://www.csg.is.titech.ac.jp/~chiba/javassist/`, 2006.

[8] T. Cofino, Y. Doganoata, Y. Drissi, T. Fin, L. Kozakov, and M. Laker. Towards knowledge management in autonomic systems. In *Proceedings of the Eight IEEE International Symposium on Computers and Communications (ISCC'03)*, pages 789–794, Kemer, Turkey, June 2003.

[9] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002.

[10] A. Haydarlou, M. Oey, B. Overeinder, and F. Brazier. Using semantic web technology for self-management of distributed object-oriented systems. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI-06)*, Hong Kong, China, Dec. 2006.

[11] A. Haydarlou, B. Overeinder, M. Oey, and F. Brazier. Multi-level model-based self-diagnosis of distributed object-oriented systems. In *Proceedings of the 3rd IFIP International Conference on Autonomic and Trusted Computing (ATC-06)*, Wuhan, China, Sept. 2006.

[12] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML. `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`, 2004.

[13] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[14] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):45–50, 1995.

[15] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic markup for web services. `http://www.w3.org/Submission/OWL-S/`, 2004.

[16] I. Mozetic. Hierarchical model-based diagnosis. *International Journal of Man-Machine Studies*, 35(3):329–362, 1991.

[17] B. Peischl and F. Wotawa. Model-based diagnosis or reasoning from first principles. *IEEE Intelligent Systems*, 18(3):32–37, 2003.

[18] Stanford Medical Informatics. The protégé owl editor. `http://protege.stanford.edu/overview/protege-owl.html`.