

Multi-Level Model-Based Self-Diagnosis of Distributed Object-Oriented Systems

A.R. Haydarlou*, B.J. Overeinder, M.A. Oey, and F.M.T. Brazier

Vrije Universiteit Amsterdam, De Boelelaan 1081a,
1081 HV Amsterdam, The Netherlands
{rezahay,bjo,michel,frances}@cs.vu.nl

Abstract. Self-healing relies on correct diagnosis of system malfunctioning. This paper presents a use-case based approach to self-diagnosis. Both a static and a dynamic model of a managed-system are distinguished with explicit functional, implementational, and operational knowledge of specific use-cases. This knowledge is used to define sensors to detect and localise anomalies at the same three levels, providing the input needed to perform informed diagnosis. The models presented can be used to automatically instrument existing distributed legacy systems.

1 Introduction

Autonomic computing and self-management have emerged as promising approaches to the management complexity of networked, distributed systems [1]. Self-healing, one of the important aspects of self-management, mandates effective diagnosis of system malfunctioning. Self-diagnosis of anomalies in physically distributed systems for which relations between the different information sources are not always clearly defined, is a challenge.

This paper presents an approach to self-diagnosis within the context of a self-healing framework. Characteristics of this framework are that it (1) can be applied to existing distributed object-oriented applications¹, including legacy applications, (2) distinguishes different use-case based views and levels within an existing distributed system (system-level, component-level, class-level), and (3) provides a structure to support application model construction. The application models needed to achieve use-case based self-diagnosis are the specific focus of this paper.

Sections 2 and 3 present the approach to self-healing in more detail. Section 4 describes the two models: a static and dynamic use-case based model of a system. Section 5 describes the instrumentation of sensors following the same three levels distinguished in both models. Section 6 illustrates the approach for a specific use-case in the domain of financial trading. Section 7 positions the approach in the context of related research.

* This research is supported by the NLnet Foundation, <http://www.nlnet.nl>, and Fortis Bank Netherlands, <http://www.fortisbank.nl>.

¹ The terms *system* and *application* will be used interchangeably.

2 Self-Healing Design Rationale

Software engineering involves many different parties each with their own roles and responsibilities, including functional analysts, developers, and system administrators. Each of these parties is interested in the internal working (behaviour) of a system, but have their own view of the system: a *functional*, *implementational*, or *operational* view, respectively². These views are related to *system use-cases*, which describe the behaviour of a system by specifying the response of a system to a given request. These use-case specifications are acquired in the requirements acquisition phase of a software development process. The views are defined as follows:

Functional View - This view describes the high level functionality of different parts of an application and how these parts are combined to realise a use-case specification. Sequences of actions (*functional steps*) specify what needs to be done by an application to attain the expected functionality. This knowledge can be used to roughly locate the malfunctioning part of an application.

Implementational View - This view describes the low level code of an application and describes how a use-case specification has been implemented as a chain of methods (functions) at code level. Pre- and post conditions of a method are specified as are sequences of important code statements (*implementational steps*) in a method body. This knowledge can be used to localise and address the root-cause of application malfunctioning.

Operational View - This view describes the runtime processes and lightweight threads of control executing within an application and describes how they should be synchronised in order to realise a use-case specification. The structure of interacting processes and lightweight threads are specified as are sequences of process-to-process or thread-to-thread communication (*operational steps*). This knowledge can be used to discover partial system shutdowns and process-oriented application malfunctioning.

Our approach to self-diagnosis is based on these three views. Information from all three views is used to locate the malfunctioning part of a system at the corresponding level as indicated above.

3 Self-Healing Architecture

This section presents an overview of a complete self-healing architecture (Fig. 1(a)). At the highest level two modules are distinguished: a managed-system and an autonomic-manager. The *managed-system* can be any existing distributed object-oriented application that has been extended with sensors and effectors. *Sensors* (see Sect. 5) provide

² The views are somewhat similar to the logical, implementational, and process views used in Unified Modeling Language (UML) [2].

runtime information from the running application to the autonomic-manager and *effectors* provide adaptation instructions from the autonomic-manager to the running application.

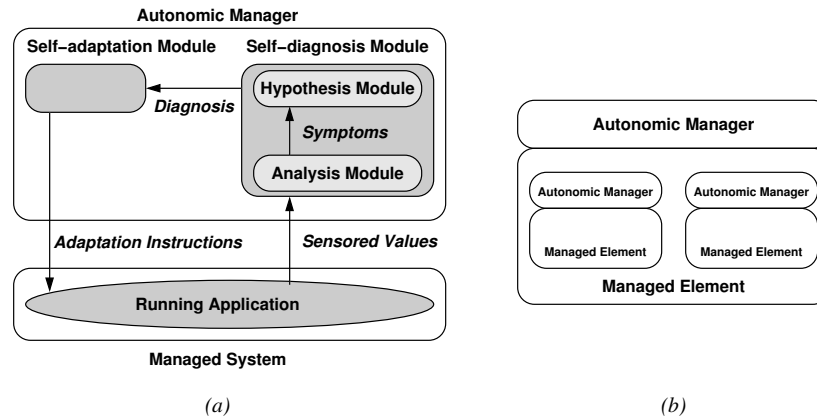


Fig. 1. Self-healing architecture. (a) Interaction between an Autonomic Manager and the Managed System. (b) Each Managed Element has its own separate Autonomic Manager.

The *autonomic-manager* itself has two modules: (1) a self-diagnosis module, and (2) a self-adaptation module. The *self-diagnosis module* continuously checks whether the running application shows any abnormal behaviour by monitoring the values it receives from the sensors placed in the application. If so, the self-diagnosis module determines a diagnosis and passes it to the *self-adaptation module*. This latter module is responsible for planning actions that must be taken to resolve the abnormal behaviour and uses the effectors to do so.

The self-diagnosis module contains the analysis module and the hypothesis module. The *analysis module* is responsible for identifying abnormal behaviour (*symptom*) of a running application based on values received by the sensors and the information available in its application model (see Sect. 4) and its analysis knowledge-base. The *hypothesis module* determines the root-cause of a given abnormal behaviour using the knowledge available in its *hypothesis knowledge-base*.

The *analysis knowledge-base* contains constraint rules (SWRL rules [3]) and meta-knowledge about the constraint rules. Constraint rules define whether observed values coming from different sensors are consistent with a certain symptom associated with a job. Depending on the type of job, symptoms can be functional, implementational, or operational. The meta-knowledge in the analysis module contains strategic rules regarding success or failure of constraint rules.

This paper focuses only on the self-diagnosis module, and in particular on the models and sensors used by the analysis module.

4 Application Model Design

The main goal of designing a model of a running application is to provide useful diagnostic information about the application to the autonomic-manager. Based on the application model and diagnosis information, the autonomic-manager can coordinate its operations for self-configuration, self-healing, and self-optimisation of the managed-application. The information provided concerns both the static structure of an application and its dynamic behaviour at runtime. The application model provides knowledge about (1) which parts of an application cooperate with each other to realise a use-case (the static application model described in Sect. 4.1), and (2) how these application parts communicate and cooperate to realise a use-case (the dynamic application model described in Sect. 4.2). The Ontology Web Language (OWL) [4] is used to represent the information in both models.

4.1 Static Application Model

Distributed object-oriented applications 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. The proposed static model considers each of these application parts as separately manageable parts: *managed-elements*. Each managed-element is associated with a separate autonomic-manager (see Fig. 1(b)).

Dividing a system into multiple, smaller managed-elements, each with an associated autonomic-manager has some advantages: (1) it simplifies the description of dynamic behaviour of structurally complex applications, (2) it simplifies and reduces the required self-healing knowledge, and (3) it facilitates the distribution, migration, and reuse of the application elements by equipping each element with its own specific self-healing knowledge. Consequently, in order to coordinate the self-healing actions, the autonomic-managers should communicate with each other as they are divided over multiple elements.

There are different types of managed-elements in a static system model. The `ManagedElement` is an abstract entity that contains the common properties of all managed-elements. Each `ManagedElement` is associated with an `AutonomicManager` and has a list of `Connectors` that bind the `ManagedElement` to other `ManagedElements`. A `Connector` has a `Protocol` that is used by `ManagedElements` to communicate with each other. Furthermore, each `ManagedElement` and `Connector` has a list of `States` and `Events`.

A `State` models a data item (of the `ManagedElement` or `Connector`) whose value may change during application lifetime and that is important enough to monitor. A `State` can be either an `AtomicState` or a `CompositeState`. A `CompositeState` consists of one or more states and corresponds to composite data items. An `Event` models unexpected happenings during execution of the application.

There are four different types of managed-elements: `ManagedSystem`, `ManagedRunnable`, `ManagedComponent`, and `ManagedClass`. The `ManagedSystem` is used to describe and manage the collective behaviour of a number of related subsystems. It is composed of a number of `ManagedRunnables`.

A `ManagedRunnable` models a part of an application that is runnable, that can be started/stopped. Examples of a `ManagedRunnable` are a subsystem or an execution

thread. When a running application is observed from the operational point of view, the application is monitored at the level of `ManagedRunnables`. In other words, the `ManagedRunnable` is the diagnosis unit. From the operational viewpoint, the list of `States` associated with a `ManagedRunnable` describes the status of a process or thread. A `State` within a `ManagedRunnable` is called a `ManagedRunnableState`. Events correspond with events such as startup and shutdown of the `ManagedRunnable`.

A `ManagedRunnable` is composed of one or more other `ManagedRunnables` or is composed of a number of `ManagedComponents`, each of which models a software component or a library that implements a specific functionality (such as a logging component, an object-relational mapping component, or an XML parser component). Usually, the `ManagedComponents` are the diagnosis units when the running application is observed from the functional point of view. From this viewpoint, the list of `States` associated with a `ManagedComponent` describes the status of a component. A `State` within a `ManagedComponent` is called a `ManagedComponentState`. Events correspond with exceptions thrown from the component, or user-defined events associated with the modeled component.

A `ManagedComponent` is either composed of one or more other `ManagedComponents` or composed of a number of `ManagedClasses`. The `ManagedClass` is an atomic managed-element that corresponds with a coding-level class and models the static properties of that class (such as, class name, class file, and file location). From the implementational point of view, the application malfunctioning can be detected at the `ManagedClass` level. From this viewpoint, the list of `States` associated with a `ManagedClass` describes the set of class and instance (object) variables declared in the modeled class. A `State` within a `ManagedClass` is called a `ManagedClassState`. Events correspond with exceptions raised at coding level.

A `Connector` binds two `ManagedElements` together. More precisely, they model the fact that two `ManagedElements` can interact with each other. For example, a method within one `ManagedClass` can call a method of another `ManagedClass` or one `ManagedRunnable` can send information to another `ManagedRunnable` over a network via some protocol. The list of `States` associated with a `Connector` describes the state of the connection itself, such as whether the connection is up or down. A `State` within a `Connector` is called a `ConnectorState`. Events correspond with exceptions such as a network timeout.

4.2 Dynamic Application Model

Conceptually, the dynamic behaviour of an application can be modeled as a sequence of the following three steps: (1) the application receives a request together with corresponding *input-data* (parameter) to perform some *job*, (2) the application internally goes into a *job execution channel*, and finally, (3) the application returns some *output-data* (result).

Within a job execution channel (Fig. 2), a job is realised by sequentially (or in parallel, but the current assumption is that tasks are sequential) executing a number of *tasks*. Tasks, in general, read some input, manipulate data, and produce some output. The data that they manipulate originates from the job input-data, the shared state of the

ManagedElement that executes the job, and/or an external data source (e.g., database, user interface, or queue).

The functional analyst, developer, and system administrator, from their own point of view, can use this simple abstraction of the application's runtime behaviour to describe the internal working of the application. The proposed dynamic system model is based on this abstraction.

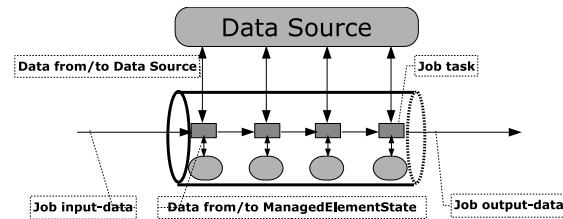


Fig. 2. Dynamic application model.

The dynamic application model of an application consists of the specification of a collection of Jobs, Tasks, States, Events, and Sensors. A Job has a name, zero or more inputs, one optional output, a number of tasks, and one or more managed-elements that cooperate to realise the job execution. Depending on which view one takes when modeling the dynamic behaviour of an application, each job corresponds to a single functional behaviour, implementational use-case realisation, or operational use-case realisation. The three types of jobs are called: FunctionalJobs which are associated with ManagedComponents, OperationalJobs which are associated with ManagedRunnables, and ImplementationalJobs which are associated with ManagedClasses.

Each Task within a job corresponds to one or more functional, implementational, or operational steps, depending on the type of job. A specification of a task consists of the task's name, its input and output states, and the name of the ManagedElement that executes this task and whose states can be modified by this task.

Tasks are the basic unit of the dynamic application model and are used to reason about the health status of an application. To detect runtime failures, the execution of a job is monitored. The correct behaviour of a job can be defined in terms of the correct job input-data, the correct job output-data, and the correct changes the job makes to the data it manipulates. Any abnormal, and therefore undesirable, behaviour of a job is described as one or more Symptoms. A Symptom has a name which is associated with a rule (*constraint*). Rules are logical combinations of comparison operators over sensed values. During execution of a job, the autonomic-manager continuously checks to see whether any of these constraints are violated.

Tasks have the common property that they are able to interrupt the normal execution process of their job and cause *exceptional* behaviours. Autonomic-managers can monitor these exceptional behaviours and react accordingly, based on the reaction policy specified in the model.

Tasks are classified as AbstractTask, StateManipulation, JobForker, PeerInvocation, JobInputChecker, and JobOutputChecker. This classification can be extended in the future, if necessary. For the sake of space, only the StateManipulation is described below.

StateManipulation Task - The most frequently occurring tasks within a job are StateManipulation tasks, which manipulate some type of State. These tasks are used to monitor the manipulation of all incoming and outgoing data items during execution of a job. StateManipulation tasks are further categorised based on the type of State they manipulate. For example, a ManagedElementStateManipulation task manipulates the state of a managed-element, and a DataSourceStateManipulation task manipulates a state coming from an external data source, such as a user interface, persistent storage, or an asynchronous queue).

5 Model Sensors and Observation Points

To perform a diagnosis, an autonomic-manager must monitor the behaviour of the application. To this end, an application is instrumented with *model sensors*, or simply *sensors*, which send runtime information to the autonomic-manager. The values provided by these sensors enable the autonomic-manager to monitor when, and determine why an application does not behave as expected. The application model described in Sect. 4 indicates where to place sensors.

The behaviour of an application is determined by its jobs, more precisely, by its tasks defined in the dynamic model. Monitoring the execution of a task consists of monitoring all data (states) that the task reads and/or writes, and also monitoring whether an event, such as an exception, has occurred during the execution of the task. Sensors are used to monitor these tasks.

The proposed framework distinguishes between two types of Sensors: StateSensors and EventSensors. A StateSensor is used to monitor specific input or output data of a task, or to monitor any data item that is read or written by the task. The data item is part of the ManagedElement associated with the task or comes from an external data source (see Sect. 4.2). Additionally, if the associated ManagedElement of the task also uses a Connector to communicate with another ManagedElement (see static model in Sect. 4.1), the ConnectorState can also be monitored by this type of sensor. The EventSensor is used to monitor any event that occurs during the execution of a task, such as an Exception.

The information supplied by these Sensors to the autonomic-manager include: (1) the value of a data item or the information regarding an event, and (2) the task that has manipulated this data item or the task that has caused the event. With this information the autonomic-manager can determine whether the constraint associated with a Symptom of the corresponding job has been violated. If so, the application has shown abnormal behaviour and the root-cause can be identified since it is known with which task the Symptom has been associated. Subsequently, actions should be taken to remedy the abnormal behaviour.

The problem of where to place a `Sensor` is now easily solved. As each task is executed by one specific `ManagedElement` (defined in the static model), the physical place for the corresponding `Sensor` in the application code is known.

`EventSensors` form a general mechanism to notify the autonomic-manager of any event that may be of interest. These events can be defined by a user of the framework in the static model (see Sect. 4.1). Examples of such events are: (1) `Exception` event: an exception has occurred, (2) `TimerExpiration` event: a timer expires, or (3) `TaskExecution` event: a specific task (one of the tasks mentioned in Sect. 4.2) starts executing.

`States`, `Tasks`, and `Sensors` are closely related: a specific state-type is manipulated by the corresponding task-type and is monitored by the corresponding sensor-type. For example, a `DataSourceState` is manipulated by a `DataSourceStateManipulation` task and is monitored by a `DataSourceStateSensor`. The framework enforces this relation during the construction of application models.

Each `Sensor` monitors the value (`observedValue`) of some item (`monitoredItem`), which could be a state or an event. Furthermore, each sensor has a corresponding `sensorTrigger`. The `SensorTrigger` represents the `Task` or `ManagedElement` that defines the context in which the data item or event is monitored.

`Sensors` are provided either by the self-healing framework or by users. The proposed self-healing framework uses Aspect Oriented Programming (AOP) [5] to instrument sensors at the specified observation points in the compiled code of an existing application. The framework also defines a *SensorInterface* which can be implemented by users to provide information from the running application.

6 Practical Scenario

The scenario described in this section is borrowed from Fortis Bank Netherlands' distributed trading application. The complete application is very complex and consists of a considerable number of subsystems and a large number of components and classes. In this paper, the proposed self-healing model is applied to a simplified version of payments in the trading application.

6.1 Trading Application Static Model

The static model of the simplified trading application consists of a `ManagedSystem` that contains five `ManagedRunnables`: a browser, a web application, a legacy backend (mainframe application), a mediator, and a database manager.

Each of these runnables has a large number of `ManagedComponents`. In this scenario, only the following components are considered: payment component, data access (Hibernate [6]), web interface (Struts [7]), and web service. Each of these components, in turn, consists of a large number of `ManagedClasses` (Java classes). The `ManagedRunnables` have four `Connectors`: (1) a `HttpProtocol` connector between the browser and the Web application, (2) a `SOAPProtocol` connector between the Web application and the mediator, (3) a `MessageOrientedProtocol` connector between the mediator and the legacy backend, and (4) a `JDBCProtocol` connector between the Web application and the database manager.

```

Fund: CompositeState {
  memberStates:
    (1) fund_name: AtomicState          (2) fund_group: AtomicState
}
Trade: CompositeState {
  memberStates:
    (1) payment_amount: AtomicState     (4) trade_fund: Fund
    (2) account_nr: AtomicState         (5) trade_id: AtomicState
    (3) trade_status: AtomicState
}

```

Specification 1. Example state specifications.

6.2 Trading Application Dynamic Model

As stated above, this scenario focuses on the payment use-case. Authorised Fortis employees inspect a trade submitted by an authorised employee of a fund company, and send a payment request to the legacy backend.

Specification 1 shows the specification of states that concern the payment job. Note that, only states that are of importance to the use-case are specified. These states are abstractions of the corresponding objects in the trading application. As the OWL representation has not been introduced in this paper, the model elements are presented in textual format.

For each view, the jobs and sensors used in our scenario are specified below. The functional and operational jobs cover the complete payment use-case. The implementational view is limited to the payment status change.

Functional Job Specifications - Specification 2 shows the specification of the functional payment job with its input, output, tasks, associated managed-elements, and abnormal functional behaviours. The input needed to determine whether a symptom occurs is provided by one or more sensors of which an example is also specified in Specification 2. For example, symptom (1) occurs if the value of `Trade.payment_amount`, retrieved from a web form after executing task (1), is negative.

Operational Job Specifications - Specification 3 shows the operational payment job related to the payment use-case from the operational point of view and clarifies which processes (or threads) cooperate during realisation of the payment use-case. *Operational symptoms* indicate infrastructural malfunctioning detected by one or more sensors of which an example is also specified in Specification 3. For example, symptom (3) occurs if, during a periodic check, the `database_manager` does not respond.

Implementational Job Specifications - Specification 4 shows the implementational job specification corresponding to the method `changeTradeStatus` defined in the Java class `TradePersistency`. *Implementational symptoms* indicate code malfunctioning detected by one or more sensors of which an example is also specified in Specification 4. For example, symptom (1) occurs if the value of `Trade.trade_status_param`, just before executing task (3), is unknown.

```

FPayment: FunctionalJob {
  input: Trade (without payment status)
  output: Trade (with payment status)
  tasks: (1) Obtain payment command from user          (4) Change trade status in the database
         (2) Send payment to backend                  (5) Show updated trade to the user
         (3) Obtain payment status from backend
  managedElements:
         (1) web interface                            (3) web service
         (2) payment component                        (4) data access
  symptoms:
         (1) Trade.payment_amount < 0                (3) not_authorized_exception occurred
         (2) Trade.account_nr == 'unknown'
}

PaymentAmountSensor: WebFormStateSensor {
  monitoredItem: Trade.payment_amount
  observedValue: StateValue,
  sensorTrigger: Task (1)
}

```

Specification 2. Functional job and sensor specifications.

```

OPayment: OperationalJob {
  input: Trade (without payment status)
  output: Trade (with payment status)
  tasks: (1) Send payment command via HTTP request    (6) Send payment status to web application
         (2) Receive HTTP request                    (7) Send trade status change to database
         (3) Send payment to mediator via SOAP       manager via JDBC
         (4) Send payment to backend via MQSeries    (8) Send transaction confirmation
         (5) Send payment status to mediator         (9) Send updated trade to browser via HTTP
  managedElements:
         (1) browser                                  (4) backend
         (2) web application                          (5) database manager
         (3) mediator
  symptoms:
         (1) mediator's max_connections reached      (3) database manager does not respond
         (2) backend does not listen to port x
}

DatabaseManagerSensor: ManagedRunnableStateSensor {
  monitoredItem: ManagedRunnableState
  observedValue: StateValue
  sensorTrigger: TimerExpiration
}

```

Specification 3. Operational job and sensor specifications.

```

IChangeTradeStatus: ImplementationalJob {
  input: trade_id_param, trade_status_param
  output: Trade (with changed status)
  tasks: (1) Start database session                  (4) Write trade to database
         (2) Read trade from database                (5) End database session
         (3) Change trade status                    (6) Return the trade
  managedElements:
         (1) TradePersistency
  symptoms:
         (1) trade_status_param == 'unknown'        (3) non_existing_trade_exception occurred
         (2) Trade.trade_status != trade_status_param
}

TradeStatusParamSensor: TaskInputSensor {
  monitoredItem: trade_status_param
  observedValue: StateValue
  sensorTrigger: Task (3)
}

```

Specification 4. Implementational job and sensor specifications.

7 Discussion

Self-diagnosis of complex systems is a challenge: especially when existing legacy systems are the target. This paper is based on the concept of a model-based framework for self-diagnosis in which three views of a complex system are defined and related: the functional view, the operational view, and the implementational view. Self-diagnosis is based on both a static and a dynamic model of a complex system in which these views are mapped onto levels of system model specification. Sensors are explicitly related to the levels: sensor types are defined for each of the levels. A system administrator is provided the structure to specify sensors which can be placed automatically.

This research can be seen to extend the Robinson's work [8] in this area. The three views of system requirements is new, as is the explicit distinction between a static and a dynamic model of a system. Symptom specification is comparable with the specification of high-level requirements expressed in a formal language. The autonomic-manager capable of reasoning about a running complex system at the three levels distinguished above, can be viewed as an extension of the *monitor program* proposed by [8].

Dowling and Cahill [9] introduce *K-Component* model as a programming model and architecture for building self-adaptive component software. In the K-Component model, which is an extension of CORBA component model, components are the units of computation. Therefore, this model does not support self-management within the internal structure of components (for example, Java classes if the components are written in Java). The *feedback states* and *feedback events* are comparable with the notion of State and Event.

Baresi and Guinea [10] propose external *monitoring rules* (specified in *WS-COL*) to monitor the execution of WS-BPEL process. A monitoring rule consists of *monitoring location*, *monitoring parameters*, and *monitoring expression*. In our framework, a Sensor contains the same information as the monitoring location and monitoring parameters, and SWRL rules are comparable with monitoring expressions.

Currently, an implementation of the proposed framework is being developed, and the research focuses on extending the framework. For example, meta-knowledge rules in the analysis module will be formulated, and support for parallel execution of tasks within a job execution channel will be designed. Additionally, the proposed framework will be applied in the field of service-oriented computing.

References

1. Kephart, J., Chess, D.: The vision of autonomic computing. *Computer* **36** (2003) 41–50
2. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. 2nd edn. Addison-Wesley Professional, Reading, MA (2004)
3. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosz, B., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML. <http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/> (2004)
4. Bechhofer, S., Harmelen, F., Hendler, J.A., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: Owl web ontology language reference. <http://www.w3.org/TR/owl-ref> (2004)

5. Elrad, T., Filman, R.E., Bader, A.: Aspect-oriented programming: Introduction. *Communications of the ACM* **44** (2001) 29–32
6. JBoss Federation: Hibernate framework. <http://www.hibernate.org> (2006)
7. Apache Software Foundation: Struts framework. <http://struts.apache.org> (2006)
8. Robinson, W.: Monitoring software requirements using instrumented code. In: HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9, Washington, DC, USA, IEEE Computer Society (2002) 276.2
9. Dowling, J., Cahill, V.: The k-component architecture meta-model for self-adaptive software. In: REFLECTION '01: Proceedings of the Third Int. Conference on Metalevel Architectures and Separation of Crosscutting Concerns, London, UK, Springer-Verlag (2001) 81–88
10. Baresi, L., Guinea, S.: Towards dynamic monitoring of ws-bpel processes. In: ICSOC: Proceedings of the 3rd Int. Conference on Service Oriented Computing. (2005) 269–282