Modelling Self-Management Requirements in Service-Oriented Systems using SelfMML

Carlos Rodríguez-Fernández[1], Jorge J. Gómez-Sanz[1], and Juan Pavón[1]

Facultad de Informática,
C/ Prof. José García Santesmases, s/n,
28040 Madrid, Spain
{carlosro,jjgomez,jpavon}@fdi.ucm.es

Abstract. This paper introduces a language called Self-Management Modelling Language (SelfMML) which supports the modelling of self-management capability requirements. The paper presents a case study related to the automatic re-binding features in services which illustrates and analyses the language usage in service-oriented systems.

1 Introduction

Software Systems intended to provide services usually should be operative at peak performance 24/7 to meet the end-user needs and the business requirements. Therefore, it is desired that such systems have the capability of managing themselves without human intervention, since a system could done such operations in a faster and more accurate way. This capability is called self-management[1].

The self-management definition is detailed by its four aspects: self-optimisation, self-configuration, self-healing and self-protection[1]. *Self-optimisation* is the automatic looking and finding of opportunities to tune the system to improve the performance and the efficiency; *Self-configuration* is the automatic configuring of the system following high level policies; *Self-healing* is the automatic recovering from unhealthy state; and *Self-protection* is the protecting itself from attacks and cascading errors, with anticipatory or reactive actions [1].

Obtaining such capabilities in a service-oriented system is not a trivial work and require a costly engineering effort. This work could be facilitated if the developer had specialised tools which support the definition of those capabilities. To support this claim, this paper studies the impact of applying the Self-Management Modelling Language (SelfMML) for the modelling of self-management capability requirements in a service-oriented system.

The Self-Management Modelling Language (SelfMML) is a language which intends to assist in the engineering of self-management capability requirements, providing visual representations related to the specification of them. A visual editor tool for this language is provided to create, view, edit and store SelfMML specifications. This tool can be downloaded at `http://selfmml.sf.net`.

The chosen case study for applying SelfMML is based on a well known scenario in service-oriented computing: the re-binding. Specifically, the re-binding

capability requirement in an on-line blog system, which is considered as a self-management capability requirement that will be modelled using the proposed language.

It is important to remark that we focus on the modelling aspects, not in the requirement engineering process. Hence, we intend to provide a modelling language that permits a developer to effectively capture and specify a self-management requirement, but we do not provide assistance for the requirement engineering process that uses this language. The SelfMML scope is limited to the self-management requirement specification. A self-management capability represents an expected behaviour from the system, which should be implemented in some way during the chosen development process. During the realisation of these, the engineer can identify elements and design the architecture of the final system according to the specified expected behaviour, analogous to the realisation of use cases. The developer could consider some framework or reference architecture for self-management or just develop an ad-hoc solution. A method to realise and verify self-management requirements modelled by SelfMML is not study in this work.

This work is structured as follows. Section 2 describes the language SelfMML. Section 3 presents a case study used as illustrative example of the language usage. Section 4 shows some related work that has been taken into account for this language. Section 5 introduces the conclusions.

2 The Self-Management Modelling Language (SelfMML)

The Self-Management Modelling Language (SelfMML) is a language to be used in the modelling of self-management capabilities that a system should have, that is, self-management capability requirements. This language is made from UML 2.2 Superstructure, concretely, copies all elements from the *Use Cases* and *Activities* packages and extends them with new elements. Also imports elements from the *Kernel* package for the definition of the elements.

The language has a meta-model that defines its abstract syntax. Further information about the meta-model can be found in `http://selfmml.sf.net`. The language is described below (see figures 1, 2 and 3).

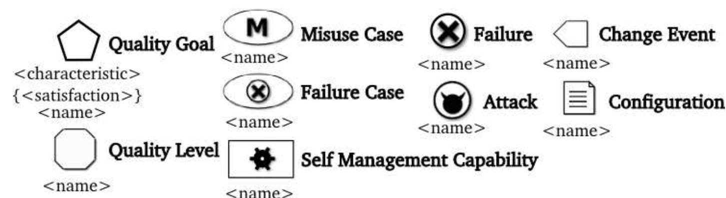

Fig. 1. SelfMML (1)

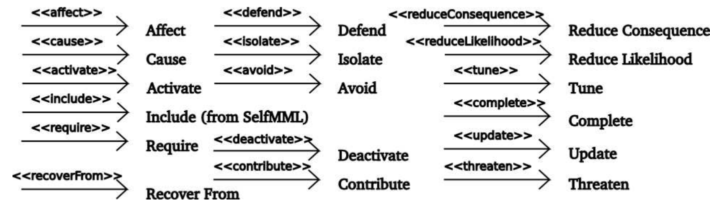

Fig. 2. SelfMML (2)

- **Self-Management Capability**. Self-Management Capabilities are the abilities of systems to do management operation by themselves on themselves. This element is provided for the representation of self-management capability requirements in a system.

Self-Management Capabilities are usually related to quality requirements as maintainability, portability, reliability, usability, availability, among others [2,3], in a way that contribute to the satisfaction of them. The language has the Quality Goal and Quality Level elements for the representation of quality requirements (see figure 1).

- **Quality Goal**. This element represents a quality requirement described as a goal that the system should maintain. It has a *characteristic* property that describes the characteristic or factor related to the quality requirements, and also has a *satisfaction* property that describes how the quality goal is satisfied by the using of some expression in natural language or another language such as Object Constraint Language (OCL). Such expressions could describe what are acceptable values for the quality metrics related to the quality requirement, following the IEEE 1062-1998 Standard recommendations.
- **Quality Level**. This element represents a quality level which groups quality goals that the system should maintain.

The language lets developers model how self-management capabilities are related to Quality Requirements and Use Cases, by the using of the relationships *contribute* for quality goals and *require* for use cases. Also the *include* relationship is provided to describe what quality goals are included in a specific quality level (see figure 2). Quality Goals can be connected to describe contribution with the *contribute* relationship too.

Problems are the target of self-protection and self-healing capabilities. The language provides elements to model possible problems in the system following the philosophy from the Failure Modes and Effect Analysis methods (FMEA)[4]. These elements are:

- **Failure**. This element describes a failure that could happen in the system.
- **Failure Case**. This element describes what is the wrong behaviour that a system shows when some failure has happened (failure modes in FMEA).
- **Misuse Case**. This element describes a misuse of a system that an user does which can lead to a failure[5,6,7].

– **Attack**. This element describes an attack against the system.

A failure, a misuse case or an attack can be related to a quality goal with the *threaten* relationship indicating that the first ones threaten the satisfaction of the second ones. Failures can be connected to failure cases with the *activate* relationship indicating that the first ones activate the wrong behaviour described in the second ones. Also failures can be connected to use cases with the *affect* relationship indicating that the first ones affect the normal operation of the functionality described in the second ones. Failures, misuse cases and attacks can be connected to failures with the *cause* relationship indicating that the first ones cause the second ones.

A self-management capability, as a self-protection capability, can be connected to problems (failures, misuse case and attacks) to indicate that the capability takes anticipatory actions to avoid, reduce the consequence or reduce the likelihood of such problems using the *avoid*, *reduceConsequence* and *reduceLikelihood* relationships respectively. Also, as a self-protection capability, the elements can be connected to attacks with the *defend* relationship to indicate that the system reacts to such attacks; and can be related to failures with the *isolate* relationship to indicate that the system isolates such failures to avoid the cascading failures that they could cause.

A self-management capability, as a self-healing capability, can be connected to a failure with the *recoverFrom* relationship indicating that the capability recovers the system from the failure. Also, it can be connected to a failure case with the *deactivate* relationship indicating that the capability deactivates the wrong behaviour of the system.

According to the self-configuration and self-optimising aspect of the self-management, a self-management capability could have the intention to tune a certain configuration parameters to improve the performance of the system, and could also have the intention to complete or update a certain configuration parameters following high level policies, in reaction to change events in the system or in the environment. SelfMML provides two elements to model such configurations and change events:

– **Change Event**. This element describes a change event in a system.
– **Configuration**. This element describe a configuration description.

A self-management capability could be related to a configuration with the following relationships: *tune*, *complete* and *update*, indicating that the capability try to tune, compete or update the configuration. A capability could be related to a change event with *treat* relationship indicating that the capability treats the event.

SelfMML provides a set of activity nodes to specify the abstract process of a self-management capability using activities diagrams (see figure 3). A self-management process usually has a structure according to four phases[1]: monitoring, analysis, planning, and plan execution. In monitoring phase the interesting information is gathered; in the analysis phase, the information is processed in order to infer some needed knowledge; in the planning phase the obtained

knowledge is used to decide what plan executes or to build a new one; finally in the plan execution phase the selected or built plan is executed.

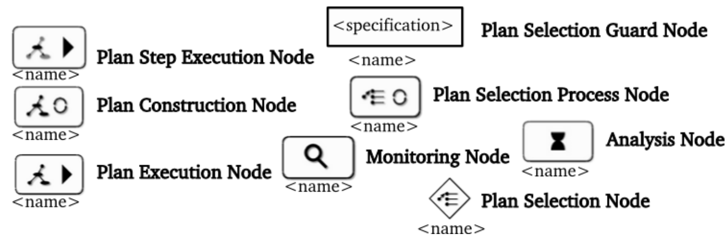

Fig. 3. SelfMML (3)

The elements for monitoring and analysis are the following (see figure 3):

- **Monitoring Node**. This element describes the monitoring activities and it continually generates tokens after each monitoring cycle. The method to obtain information by monitoring could be pulling, pushing, interception or any other kind or variant; the node does not imply the using of any particular method.
- **Analysis Node**. This element describes the analysis activity to infer knowledge from the monitoring reports.

The provided elements to model the selection or construction of plans are the following (see figure 3):

- **Plan Selection Node**. This element is for the selection of plans using OCL expression, it receives tokens from the incoming edge and copy one for each outgoing edges. The continuity of the token depends on the *Plan Selection Guard Node*.
- **Plan Selection Guard Node**. This element is to constrain the execution of plans, it receives tokens from the incoming edge and presents them to the outgoing edges only if the OCL expression described in the *specification* property is evaluated to true.
- **Plan Selection Process Node**. This element describes a selection which is done by a more sophisticated process that cannot be expressed by OCL. It copies all incoming tokens to all outgoing edges, but the continuity of such tokens depends on the evaluation of the guards on the outgoing edges. Then, a simple terms in guards can be used to describe what plan is selected.
- **Plan Construction Node**. This element describes a plan construction activity.

In order to specify the plan execution activities the language provides the following elements (see figure 3):

- **Plan Step Execution Node**. This element describes a step of a certain plan to be executed.

– **Plan Execution Node**. This element describes an undefined plan to be executed (useful when the plan is constructed).

3 Case Study: The Service Re-binding

The studied system is a blog system such as *blogger.com*. Publisher can submit posts using a publishing service. This publishing service lets users write a post and attach to the post any kind of files. It will use an external storage service to store the attached files.

This service is constrained by two quality requirements related to the availability and the reliability. Both requirements are included in an acceptable quality level for standard users. The first requirement constrains the availability of the publishing service in a value that should be equal or greater than 98%. The second requirement constrains the reliability in a fault response likelihood value equal or less than 0.05 (see figure 4).

Several problems can affect the requirements fulfilment, this paper identified only four of them. There are two failures that can affect the normal operation of the service: the used storage service becomes unavailable; and the used storage service gives too many fault responses, becoming unreliable. Both failures can cause other two: the unavailability and the unreliability of the publishing service respectively, and can threaten the quality requirements satisfaction described before (see figure 4).

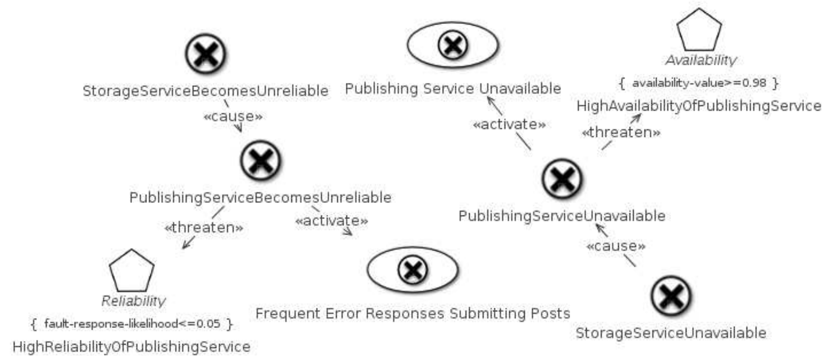

Fig. 4. Failures and Quality Goals diagram

Both failures in the publishing service activate two failure cases (see figure 4). The *Frequent Error Responses Submitting Posts* describes that "randomly" the system answers with an unexpected error when the publisher is submitting a new post. The *Publishing Service Unavailable* describes that when the publisher accesses to the service the system shows the message "The Publishing Service is Unavailable, please try later".

In order to treat these problems, there is a requirement: "The system should have the capability of detecting such problems and automatically re-bind to an alternative Storage Service, following a given selection criteria based on the quality levels offered by them. It assumes that exist a Registry that have a full description of Storage Services that can be used by the Publishing Service. The system should try to select the service that offers the highest quality level related to the availability and the reliability. But, when the selection is not clear (because exist an alternative with the highest availability but without the highest reliability, or otherwise) the system should follow the policy described by an administrator". In order to avoid the re-selection of a problematic service the system will manage a black list of them and will use the list to subtract problematic services from the list given by the Registry. This capability should be present in the provider software agents of the Publishing Service.

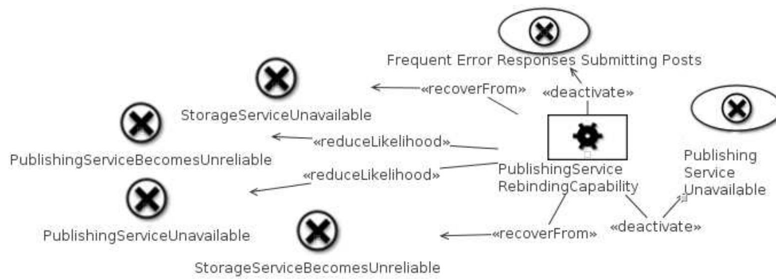

Fig. 5. The Publishing Service Re-binding Capability treating problems diagram

The capability will try to *recover* the system from the *Storage Service Unavailable* and *Storage Service Becomes Unreliable* failures. It also *deactivates* the failure cases caused indirectly by the failures. These "intentions" are related to the self-healing aspect of the capability (see figure 5).

Since the capability selects the service which offers the highest quality level in availability and reliability attributes, it reduces the likelihood of the *Publishing Service Unavailable* and *Publishing Service Becomes Unreliable* failures (see figure 5). Therefore, the capability contributes to the satisfaction of both quality requirements: the high availability and the high reliability of the service (see figure 6).

The capability requires the development of some use cases in order to operate properly (see figure 6). The alternative storage services are found in a Registry, thus the capability needs the location of this registry as input. The *Configure Registry Location* use case describes the functionality of configuring the location of the registry. The capability will manage a black list of services, but problematic services could be healed and the administrator could need remove it from the black list at run-time. But even, the administrator could need add problematic services to the black list because it was detected by another system. The *Manage*

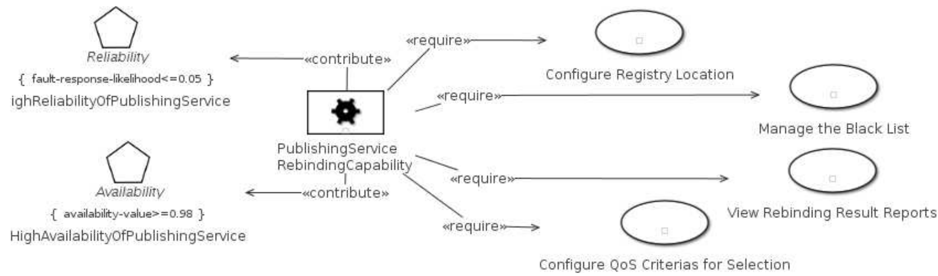

Fig. 6. The Publishing Service Re-binding Capability connected to functional and quality requirements diagram

Black List use case describes this functionality. Also the capability requires the *Configure QoS Criteria for Selection* use case to let the administrator configure what QoS criteria the capability should take into account to select alternatives; and also requires the *View Re-binding Result Report* use case which describes the functionality of viewing, by the administrator, the re-binding logs that are produced at run-time.

This capability is detailed by a self-management process model (figure 7, 8 and 9). There are two monitoring activities: *Storage Service Faults Monitoring* and *Storage Service Availability Monitoring*. The former monitors the number of faults in a period of time in order to update the current value of the quality metric "fault-response-likelihood" of the Storage Service; and the latter monitors the current operational status (available, unavailable) of the Storage Service too. These monitoring activities generate reports which are consumed by the *Analysis Of the Needs to Rebind*. The *Storage Service Faults Monitoring* activity will monitor the using of the Storage Service in order to catch faults, and the *Storage Service Availability Monitoring* could monitor the using, but also could do the monitoring either directly asking to the Storage Service, or subscribing itself to some heartbeat signal.

The *Analysis Of the Needs to Rebind* activity decides if the rebinding is needed or not. The decision is made using the monitoring reports and the quality requirements of the storage service. If the "fault-response-likelihood" metric of the Storage Service is equal or greater than 0.05 the reliability of the Publishing Service will decrease, and if the Storage Service becomes unavailable the Publishing Service will get into failure. Therefore, in these cases the rebinding will be needed. However, the Storage Service could have a notification system that inform to its clients of temporary unavailability for administration purpose. Then the analysis activity could take into account such informations to make a better decision calculating how the estimated period of time in unavailability can affect the quality level of the Publishing Service. Another case where the rebinding is not needed is when there is already a rebinding in execution.

If the rebinding is needed, then the process get into a plan selection phase (see figure 7). The plan selection is made checking if there are available services

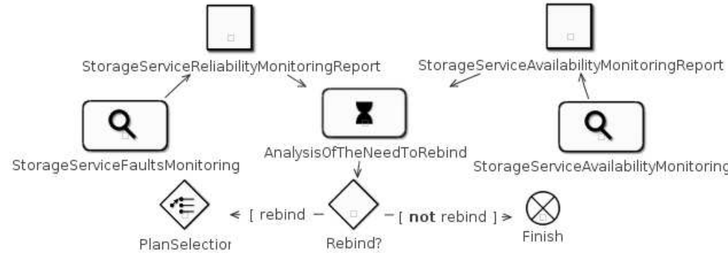

Fig. 7. The Publishing Service Re-binding Process (1) diagram

candidates or not. There are two plans: update the black list, block the using of the publishing service, and do nothing, because there are no candidates; or update the black list and rebind, because there is at less one candidate.

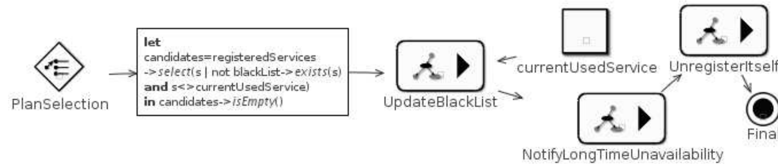

Fig. 8. The Publishing Service Re-binding Process (2) diagram

The figure 8 shows the diagram of the part corresponding to the selection and execution of the plan when there are no candidates. The first elements in the flow is the plan selection guard node that contains the OCL specification which constrains the copy of the token to the outgoing edge. In this case, the constraint is: "if the candidates list is empty then lets the token continue". The rest of elements in the flow are the updating of the black list with the current storage service, the unavailability notification and the unregistering to isolate itself from the rest of the system. The Final Node indicates that the self-management process will stop completely, even the monitoring activities.

The figure 9 shows the diagram of the part corresponding to the selection and execution of the plan when there is at less one candidate. As before, the first elements in the flow is the plan selection guard that constrains the copy of the token to the outgoing edge. When there is at less one candidate in the list the token is copy and the flow continue through the rest of plan step activities. The plan first updates the black list and notifies a temporary unavailability for administration purpose. After that, selects one service according to the given selection criteria and rebinds the publishing service to it. Finally, notifies the availability. The Flow Final Node indicates that the flow stop, but not the process.

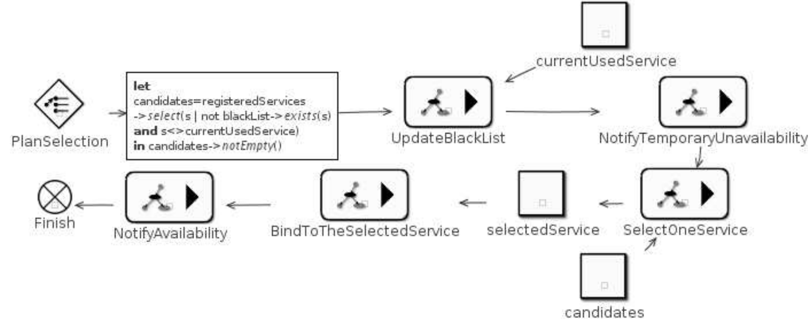

Fig. 9. The Publishing Service Re-binding Process (3) diagram

4 Related Work

The Related Work focus on the works which try to develop a language that can be used to model self-management aspects. There are some works that are more related to the topic of this paper.

One of them is the "UML Profile for Modelling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification" [8]. This one has three parts: an UML Profile and a Catalog for Quality of Service; an UML Profile for Risk Assessment; an UML Profile for Fault Tolerance Mitigation.

The QoS part is for modelling QoS requirements. This profile is limited to quality of service modelling. SelfMML lets developers model not only quality of service requirements but rather any kind of software quality requirements that could be related to self-management.

The Risk Assessment part is to be used for risk assessment modelling, but also included treatment of risks. This part is based on the CORAS method for security analysis[9]. The interesting part for this work is the meta-model and the modelling language of the treatment of risk by use cases and actors. The threats are personified and modelled as actors, the threat scenarios as use cases, and the treatments as use cases too. The connection with QoS is the modelling of risks that could affect the QoS Level of the system. The treatment of risk is specified by use cases. SelfMML makes a difference between a capability of the system to do some management operation by itself and the use cases that such a capability could require. This difference allows the modelling of self-management capabilities that could not require any use case. But also there are cases when a treatment of problems is not well described by an use case but rather by an dedicated entity, e.g. the case study presented in this work, the rebinding could not be well described if only use cases are used.

Finally, the Fault Tolerance Mitigation meta-model and Profile part is for modelling fault tolerance mechanisms for the system. It is mainly for modelling structures that will enable a system to support faults. It includes the modelling of redundancy configuration, monitoring collaborations, fault detection policies, among others. Our work will include fault tolerance, since the fault tolerance

could be considered as part of autonomic computing [3], specially when it is related with self-protection and self-healing. The structure of systems for the maintenance of health is not addressed by the language presented here, but will be considered in future works.

Another work comes from Ian Alexander [7]. This work does not propose a meta-model, what makes that work more informal, but defines a graphical language that suggest a concepts related to some aspect of self-management, e.g. “misuse case”, “mitigates”, “threat”, or “threatens”. It was used to inspire part of the work presented here.

Finally, there are several works which present graphical languages for supporting goal-oriented requirement engineering. Some of them are $i*$[10], TROPOS [11] and GRL[12]. In these works requirements are identified as goals which can represent functional requirements and non-functional requirements (named as “soft-goal”). Also the languages provide another concepts, which can help in requirement engineering works, such as: actor, task (or plan) and several relationships . Using these languages, self-management requirements can be modelled as goals. However, these languages lack certain elements which can help in the capturing, tracing and specification of these kind of requirements. Some of them are provided by SelfMML, specifically, elements and relationships for modelling failures, their causes and their effects, the *require* relationship, and the elements related to the specification of self-management processes.

5 Conclusion

This work has presented the Self-Management Modelling Language as a language that can be used in the modelling of self-management capabilities in service-oriented systems. Also a case study has been presented to study the application of the language to model self-management requirements in service-oriented systems.

This language has enabled the modelling of a self-management capability in the case study as a requirement in the system, facilitating the capturing and specification of it. However, there are other issues, that could be identified in the case study, which would be interesting to model in service-oriented systems, but the language at this moment did not provide any specific way to do it. The requirement required the specification of some policies in order to make a selection. A model of what policy options the administrator has and how the policies are used to make the selection could be interesting to have. Also, the location of the capability could be inferred by the name of the capability, but it would be interesting to associate the capability to a specific service in a service architecture model. The integration of SelfMML with a language that can be used to model service architectures like Soa Modelling Language (SoaML), could fill this gap. We would like to study both issues as future works in order to develop a more completed language.

A method for realising and verifying self-management requirement specified with SelfMML is an open issue for future work. We would also like to explore model to model transformation from self-management elements to design ele-

ments to support more completely the engineering of self-management capabilities of a target system. Specifically, since agents are suitable to realise self-management capabilities [1,13], self-management elements seem suitable to be mapped to design elements related to agent approaches.

Acknowledgements

This work has been developed with support of the program "Grupos UCM-Comunidad de Madrid" with grant CCG07-UCM/TIC-2765, and the project TIN2005-08501-C03-01, funded by the Spanish Council for Science and Technology.

References

1. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer **36**(1) (2003) 41–50
2. Nami, M.R., Sharifi, M.: Autonomic Computing: A New Approach. In: AMS '07. (March 2007) 352–357
3. Sterritt, R., Bustard, D.: Autonomic computing – a means of achieving dependability? IEEE ECBS **0** (2003) 247
4. : Failure modes and effects analysis. Technical Report MIL-P-1629, U.S. Army (1949)
5. Sindre, G., Opdahl, A.: Eliciting security requirements by misuse cases. In: TOOLS-Pacific 2000. (2000) 120–131
6. Andreas, G.S., Opdahl, A.L.: Templates for misuse case description. In: REFSQ'01. (2001) 4–5
7. Alexander, I.: Misuse cases: Use cases with hostile intent. IEEE Software **20** (2003) 58–66
8. : UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification. OMG. v1.1 edn. (April 2008)
9. Braber, F., Hogganvik, I., Lund, M.S., Stølen, K., Vraalsen, F.: Model-based security analysis in seven steps — a guided tour to the coras method. BT Technology Journal **25**(1) (2007) 101–117
10. Yu, E.S.K.: Modelling strategic relationships for process reengineering. PhD thesis, Toronto, Ont., Canada, Canada (1996)
11. Giunchiglia, F., Mylopoulos, J., Perini, A.: The tropos software development methodology: Processes, models and diagrams. In Giunchiglia, F., Odell, J., Weiß, G., eds.: AOSE'02. Volume 2585 of Lecture Notes in Computer Science., Springer (2002) 162–173
12. Amyot, D., Mussbacher, G.: URN: Towards a new standard for the visual description of requirements. In: Telecommunications and beyond: The BroaderApplicability of SDL and MSC: Third International Workshop, SAM 2002, Aberystwyth, UK, June 24-26, 2002. Revised Papers. Volume 2599 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2003) 21–37
13. Kota, R., Gibbins, N., Jennings, N.R.: Decentralised structural adaptation in agent organisations. In: Organized Adaption in Multi-Agent Systems. Volume 5368 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2009) 54–71