A Function/Artifact and Time/Incident Model of Complex Information Systems and its Application to Weakness Discovery

Michal Wosko
Brandenburg University of Technology Cottbus, Germany
wosko@tu-cottbus.de

ABSTRACT

The Author assumes as a starting point well-known assumptions on needs and weaknesses of complex business information systems. Treated as services – like in modern SOA design patterns – they are increasingly being equipped with various dependability-assuring mechanisms, but still – in the Author's view – many concrete architectures and implementations of these solutions have historically been much more an afterthought than a design-phase choice, with all disadvantageous consequences of this fact. Notably very common modular architectures did not wholly take advantage of undergoing research on dependability. An alternative to this patchwork solution should be a consistent methodology-based design procedure, whose objectives are first given in this work, to be then shown in a formal function-artifact and time-incident (FA/TI) model. The usefulness of this approach is then tested against a well-known problem (dependability and partial survivability of a realistically complex system - a modular, web-accessed, 4-tier business information system) and optimization strategies for such a system are proposed.

Key words: dependability in business information systems, 4-tier modular architecture, dependability through methodological design, function-artifact and time-incident model, generic model/metamodel

1. Introduction

Modern business information systems tend to be increasingly complex, and this complexity on one hand is a response to ever growing demand for completeness in information processing and resulting integration of so-far detached services, on the other – brings about problems in design and maintenance, that were unknown in the previous era of simple, isolated software. For a relatively long time now, design of complex systems has been brought away from the naive or romantic phase of a 100% suited to the needs of the current project, started from scratch modeling and development approach, also in response to the mentioned fact, that integration of functions available in preexisting systems was the main cause of such development (and, of course, most of the time this could be easier and cheaper achieved just bridging these preexisting systems, instead of building new ones). Thus, whatever historically the motivation was, either integration of existing heterogeneous infrastructure, or building new complex systems, modularity has been around for a while, as a very successful way of confronting complexity in design.

While this has obviously been a very successful approach to design problems, it failed to address the second issue – that of maintenance of a living system, in other words – assuring its dependability in real-life conditions. Modularity makes it easier to manage complexity, but alone it does not assure more dependability, it rather creates new potential points of failure, like in the obvious case, when a whole loosely coupled software system cannot reliably perform its function because of a failure in communications between its parts (as opposed to unitary or tightly coupled systems that do not have to rely on such "risky" communication channels). But there are many more cases, in which this holds true.

Of all the knowledge on dependable systems, notably in the most commonly used architectures of business information systems, like the 4-tier web-accessed pattern, historically very little has been apparently used at design time, in other words it was generally not a design-time concern, but rather an afterthought, provoked by failures and performance bottlenecks in already working systems, to build in some mechanisms increasing dependability, and this patchwork was done for the some of most obvious elements with little consistency and without any solid

methodological basis.
This paper tries to address these shortcomings.

2. Case study object

We will consider a well-known design of a web-accessed business information system, i.e. a standard 4-tier architecture with a web tier (layer) accepting user requests and presenting results (presentation layer), a business logic layer processing those requests with the use of data stored in an underlying database (database layer), these data being retrieved through a standardized persistence API (persistence layer), but also possibly cached in the business logic layer (when sensible, for instance from a performance-oriented point of view). See Fig. 2.1 for a general model.

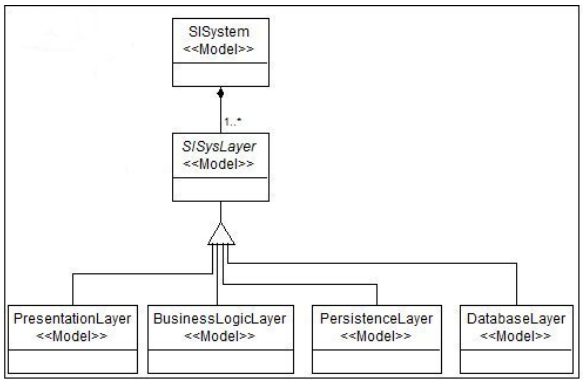


Fig. 2.1. General model of a 4-tier modular information system: such a system is an aggregation of (in this case) 4 layers, and the well-known, named layers above extend (triangle-like operator) an (abstract, thus italic) layer, which itself is a model

3. The focus in the bigger picture

Avizienis et al. [1] substantially defined dependability of a system as a threefold set (see Fig. 3.1):

1. attributes, for assessing dependability of the system
2. means of achieving or increasing dependability
3. threats that are posed to the system, that can disrupt its service.

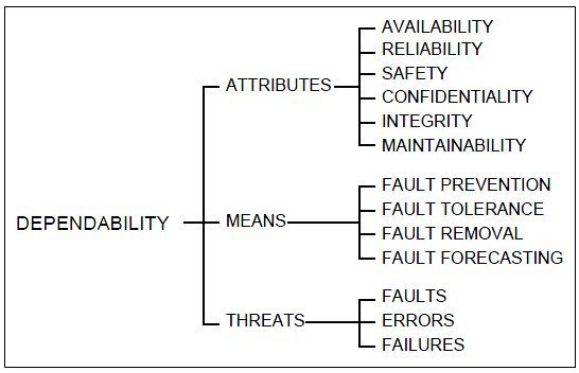


Fig. 3.1. Dependability tree as seen by Avizienis et al. [1]

In this paper only design-time and not runtime concerns are dealt with, without the ambition of ultimate completeness in respect to the given dependability taxonomy, but with the precise goal of achieving a firm base for a strictly methodological design, applied to a class of systems that the previous section described in further detail.
Given that, the objective of the proposed methodology is to achieve or increase - at design phase - integrity and through it - availability, reliability and safety (absence of catastrophic failures from the user's point of view) of this class of systems; confidentiality and maintainability are essentially left for future research (although all of them, including some aspects of maintainability, are also design and not necessarily runtime concerns).
Similarly, this approach situates itself, in the author's opinion, in the area of such means as fault prevention and tolerance; active removal and forecasting are equally left for future research. Once again, rigorous methodological design of a class of systems is given precedence over completeness.

4. The tools

Throughout this paper the generic modeling approach will be used [2], as it provides an abstraction over UML-like modeling and thus allows for unprecedented flexibility. This was the choice, because entire information systems, subsystems and functional units can hardly be represented as “classes”, and generic modeling permits to design relationships between entire models (not just classes and objects like a UML class diagram; UML itself is a specific modeling pattern applied to a domain, which can be

recreated with this metamodeling tool). First a metamodel can be created, from which different domain models can be derived. For this approach a generic modeling environment (GME, [3]) with a feature-rich GUI is provided.
Note that in metamodeling some well-known concepts, like aggregation or inheritance hierarchy, are also supported not only for (object) classes like in UML, but also for every entity type, including entire (sub-)models (and that was the reason to adopt metamodeling in the first place), but these abstractions are sometimes rendered graphically in GME in a slightly different way (see Figure 4.1). Compositions are "simplified" to aggregations with 1..1 cardinality; stereotypes are not user-definable, and for entity relationships that are not aggregations nor inheritance a construct is provided with connectors, having source, destination and an association "class" (metamodeling stereotype "connection", see Figure 4.1 on how an, in this case abstract, communication channel connects different layers in a system using layered modular architecture; this is also relevant for the further steps in the model).

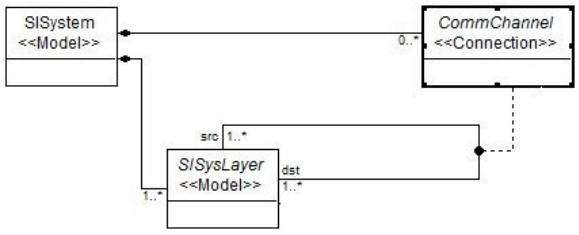


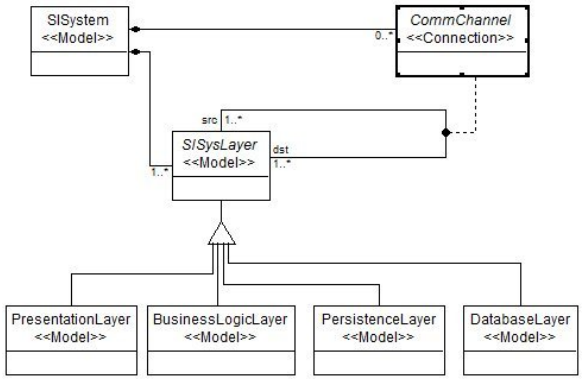
Fig. 4.1. Connections as association "classes" - layers associated by communication channels

5. Towards a formally rigorous model of a modular layered system

A modular layered information system as outlined in section 3 can be modeled as an aggregation of simpler systems (layers), that:

- do not have autonomous roles/functions
- are mutually dependent (one layer normally requires input and sends output to both neighboring layers)
- are connected with communication channels
- rely on these channels in their processing roles.

Figure 5.1. summarizes this.



5.1. A superposition of complex system, its layers and communication channels

Thus, as first conclusion, this class of systems is only then dependable (see attributes in section 2 and [1]), if *all of its layers* (functional modules) *and at least one communication channel between each pair* of neighboring layers are constantly dependable (strong dependability). A weaker property such as availability would analogously be defined as follows: a system of this class is only available in the time, when all of its layers and at least one communication channel between each pair of neighboring layers are available, and have been available and will continue to be available for the time necessary for all the processing and communication acts that are needed for a once started transaction to complete (intermittent availability that does not allow to complete a sensible transaction, for which such systems are designed, can hardly be considered "availability").
To see it in the whole picture, the introduction of new meta-entities is needed: the function-artifact and time-incident model will be constructed now.

5.1. The functional meta-entities

At the end of main section 5 it was mentioned, that even an intermittently available system has to allow to complete sensible transactions. A transaction is a model, that encompasses multiple elementary functions; these are associated by a special metamodeling construct (a metamodeling equivalent of a user-defined stereotype, here – *HasFunction*) to functional (or processing) artifacts. This can be formally put as in Figure 5.2.

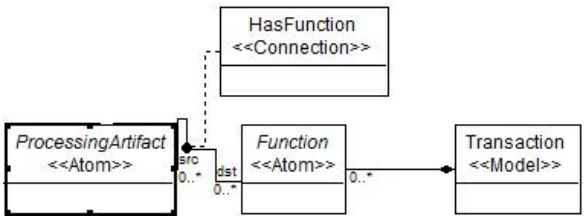


Fig. 5.2. Transactions as complex models, containing multiple functions

Formally, processing artifacts extend an abstract artifact entity, that is also the base type of the artifact carrying the communication (*CommunicationChannels*). See Figure 5.3.

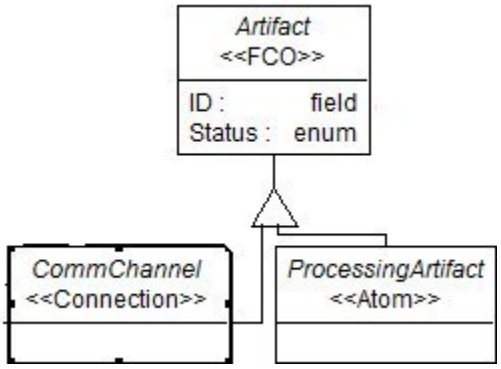


Fig. 5.3. The artifact hierarchy: the abstract entity type Artifact is an FCO – a so called First Class Object, a metamodeling construct meant to be extended by entity types of different kinds (here: Atom and Connection)

Whereas communication channel artifacts connect layers, the processing artifacts – that can be mapped directly to functional units of a generic business information system of the here discussed class (see Figure 5.4.) - are obviously also distributed in the different layers. This way a formally rigorous model of the system's functional units and their distribution is achieved as a first important step (see Figures 5.5. and 5.6.).
This model will be complete and fully functional once dependability-related meta-entities are added.

5.2. Dependability concepts and related entities in the metamodel

Following [1], the threat part of dependability tree (see Figure 3.1 for a reminder) can be modeled as well in a GME-like pattern (separately from the rest of the model for now, a way to incorporate it in the system-layer-artifact-function model has yet to be devised). Figure 5.7. shows this.

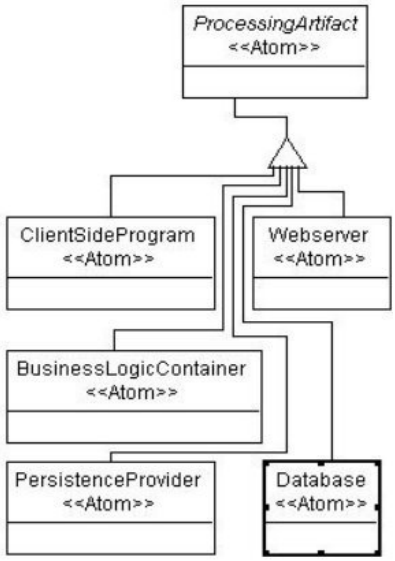


Fig. 5.4. The processing artifact, implemented by typical functional units of a generic business information system; some elements are omitted for simplicity; note that at this level every functional unit, that in reality is a subsystem, is considered elementary (atomic), because it is atomic from the viewpoint of a user transaction

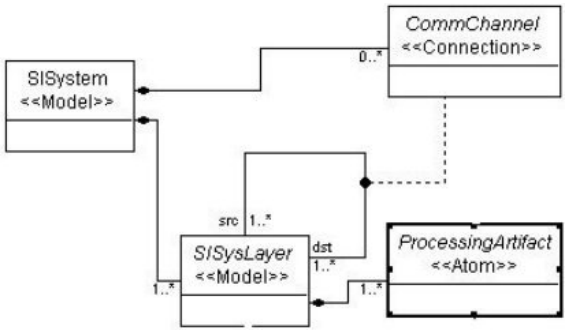


Fig. 5.5. Associations system-layers-artifacts

The next step is to link this set of associated "classes" representing threats to a fully generic system to the metamodel of our canonical layered system, constructed throughout this paper.
One very direct way of doing this (and arguably the best) is to associate failures, that are explicit, active and perceivable in nature, to artifacts (of any kind, thus to the abstract artifact "class"). Thus a failure will affect an artifact, but since there are circumstances related to this, most notably the time and duration when a failure is perceivable, a mediating entity in form of a timed incident is introduced (see Figure 5.8.).

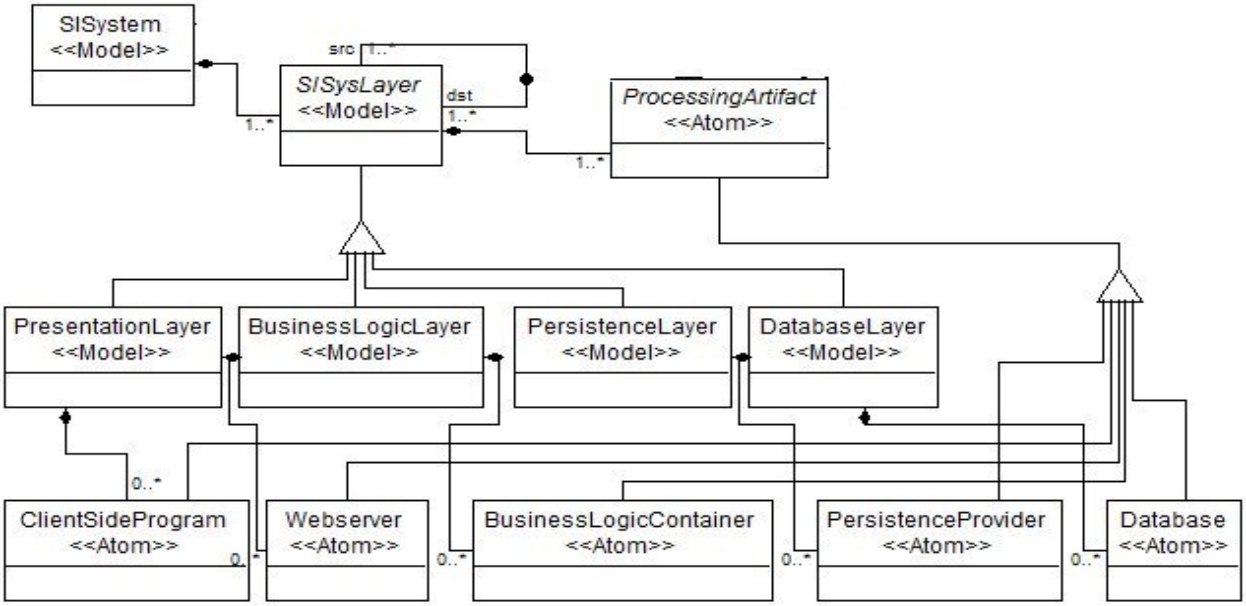


Fig. 5.6. Distribution of functional units (processing artifacts) in layers of a canonic 4-tier system (above)

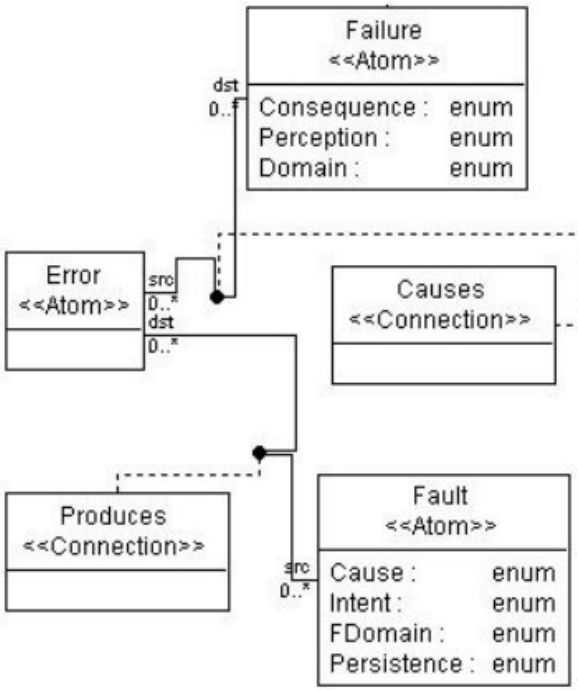


Fig. 5.7. The threat part of the dependability tree from Figure 3.1., remodeled in GME, following the relationships between concepts of fault, error and failure as assumed per definitions made in [1]; note that only a few attributes of faults are listed for simplicity's sake

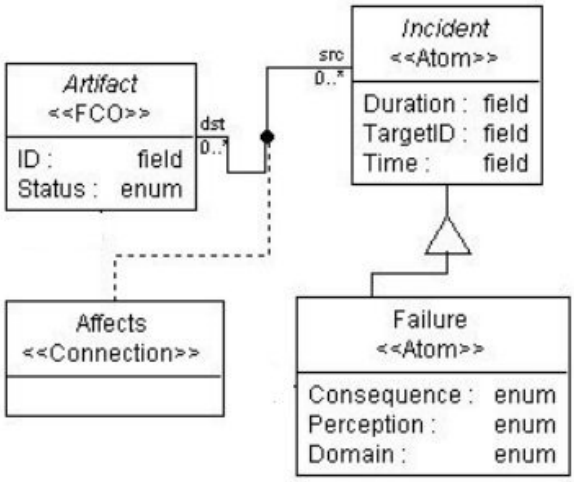


Figure 5.8. The link subsystem/artifact to failure; a failure can affect an artifact of any kind through a timed incident mediator; this entity describes time of occurrence and duration (and possibly other circumstances) through simple attributes and contains a reference to the affected artifact

The missing link is found, and the layered system artifact-function model can now be used to design new dependable systems or put existing ones to proof, as soon as mappings of functional units are performed.

5.3. Practical uses of the model

Let us say that a concrete system of the examined class has a clustered business logic container, redundant web servers and a replicated database. Thus such a system appears

not to have a single point of failure and to be strongly dependable. But this is only true, if the persistence provider middleware is integrated in the clustered container and thus clustered itself (an incident can take out any atomic processing artifact). Furthermore, if the layers are connected by single or commonly managed communication channels, each of these channels can be a single point of failure of the system (channels are artifacts and are affected by incidents). Now we see these also otherwise known facts reflected in a formal model. But some dependability aspects are only clear in the model, let us take a look at some examples.

For a layered system to be safe as per definition of safety in [1], survivability has to be "generally" assured [4]. But layers "closer" to the end user do not need to be strongly dependable, i.e. processing artifacts need not be replicated and the channels highly available, as safety is a concept applicable to persistent data (client input validation, result presentation through HTML and even business logic can fail at times). Yet such a system must at least implement transactional availability, meaning basically that the user has to be able to retrieve and manipulate data in a sensible way. Looking at the function-artifact and time-incident model, we can now formally require (and verify) that such a system has either sufficiently available functional units connected by alike communication channels, or implements replicated patterns for those artifacts, processing and communication related ones, that are likely to fail. The whole path of a data flow through the layers has to survive for the time necessary for all the processing artifacts to generate output information basing on input and for all the needed communication acts in between, or the artifacts likely to crash or produce errors have to be redundant. Communication can be repeated or take place on different channels in parallel, processing can be done in parallel or reassigned to one of the redundant artifacts. No matter the layer, parallel processing or communication can always be done but will always come at cost not only of physical resources but also of time (a mechanism is needed to agree upon results and the protocol used will be a time-based one), reassignment will always cause a delay greater in magnitude.

Once some – really few, realistically descriptive – attributes are known in a complex layered system, an application of the discussed model can be done, to simulate timed incidents and thus investigate the dependability of the system.

Let us say, we have to deal with an enterprise portal, SSO-enabled, AJAX-like web-interface, with manipulative business logic implemented through Enterprise Java Beans of various kinds, relying on container services and Java Persistence API to access a Relational Database System. Let us say furthermore that the business logic container is a clustered JBoss Application Server accessing a replicated Oracle Database.

In every JBoss instance, modified database records can be cached, until there is a programmatic or automatic workflow requiring a commit or refresh from the database, thus even intermittent failures of the Database atom will not affect the survivability of the system. This includes primarily failures in communication with the database: if this artifact becomes unavailable despite of replication or inconsistent, the system as a whole cannot be considered dependable.

Let us consider more interesting cases. The new versions of the JBoss Application Server come with clustering support out-of-the box: multiple instances started on the same LAN will automatically form a cluster, that provides (like every instance inside it) services associated with three layers: server-side presentation layer, business logic and persistence provider tiers [7]. This system appears to be strongly dependable.

But this dependability will still be a function of the dependability of the single artifacts, and which ones or which combination of them are particularly sensible is a question that the proposed FA/TI model can help to answer in a formal, rigorous and complete way. Moreover, we can study the real system's weak points, even without engaging a real enterprise portal using the aforementioned technologies. Only architectural details and parameters will suffice. That is because the model allows us to design and then instantiate a mock system, that exactly reflects the behavior of the original system (provided that the exact parameters are available). Let us see how.

All local JBoss instances extend the *ProcessingArtifact* atomic entity type, implementing all the flavors of: *Webserver*, *BusinessLogicContainer* and *Persistence-Provider* specialized entities at the same time. Thus, there are no explicit/external

communication channels between these layers inside the instances, so any incident affecting an atomic instance will make it fail completely (not just one layer). In case a central web serving proxy (like for instance, to increase throughput with static HTML-code) is used, this is not true anymore, and an incident can make this part of the presentation layer fail separately, or another can affect the communication of the proxy with the "deeper" layers. All of this, including MTBFs, recovery times and other attributes can be modeled with FA/TI and simulated with a mock instance.

Between the application server's instances there are four channels [7]: web session replication service, EJB3 stateful session beans replication service, EJB3 entity caching service and a core service named HAPartition (high availability partition). All of them rely on multiple TCP and UDP network protocols simultaneously. A timed incident taking out one or more instances, as long as one is left and reachable, will not make the cluster and thus the whole system fail. Not even client session data will be lost, thus for example once supplied credentials through single-sign-on will remain valid, nor will transactions be rolled back. But FA/TI with properly estimated parameters can still show, if and how the responsiveness of the system will be decreased, and what will happen, if a combination of incidents affecting processing artifacts (here: JBoss instances) and their communication channels occurs.

6. Conclusions and future work

At the heart of the presented work lies a formal model, linking classical modular layered business information systems architecture and the dependability concept of threats (the link is implemented through timed incidents affecting atomic functional artifacts). Through this formalism, design of new systems and screening of existing ones can proceed by means of integrating identified threats as formal design-time entities (not as something "alien" to the system itself). Also, based on this model, not only dependable systems with different attributes (different requirements for different layers) can be designed, but also a software system can be implemented to test – in a JUnit-like [8] way – response to simulated timed incidents. In section 5.2. only the threat part of the dependability tree ([1], Figure 3.1.) was remodeled using a metamodeling pattern; attributes and means were omitted. Notably, attributes like maintainability and confidentiality were left out altogether. A full model should incorporate both: possibly all attributes as a formal requirement entity type, associated with means formally represented by a strategy entity type. For instance, the model lacks the integration of watchdog (failure detection) and failure correction (task reassignment) entities.

References:

[1] A. Avizienis, J.-C. Laprie and B. Randell: "Fundamental Concepts of Dependability", Research Report No 1145, LAAS-CNRS, April 2001, http://www.cert.org/research/isw/isw2000/papers/56.pdf

[2] A. Ledeczi, M. Maroti, and P. Volgyesi, "The Generic Modeling Environment – Technical Report", Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, 37221, USA

[3] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, Ch. Thomason, G. Nordstrom, J. Sprinkle and P. Volgyesi: "The Generic Modeling Environment", Proceedings of WISP'2001, Budapest, Hungary, May, 2001

[4] Knight J.C., Strunk E.A., Sullivan K.J.: "Towards a Rigorous Definition of Information System Survivability", http://www.cs.virginia.edu/papers/discex.2003.pdf

[5] Randell B., "Software Dependability: A Personal View", in the Proc. of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25), California, USA, pp 35-41, June 1995

[6] A. Avizienis, J.-C. Laprie and B. Randell, Landwehr C.: "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, vol.1, pp. 11-33, 2004

[7] B. Stansberry, G. Zamarreno: "JBoss Application Server Clustering Guide", http://www.jboss.org/file-access/default/members/jbossas/freezone/docs/C

lustering_Guide/4/html/index.html

[8] JUnit framework: www.junit.org