periodica polytechnica

Electrical Engineering
51/1-2 (2007) 21–31
doi: 10.3311/pp.ee.2007-1-2.03
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

Aspect-oriented modelling and analysis of information systems

Péter Domokos / *István* Majzik

Received 2006-05-19

Abstract

In this paper we introduce an approach of aspect-oriented modelling and analysis of information systems. First we give an overview of the concepts of Aspect Oriented Programming and provide an outlook to model aspect-oriented programs. On the basis of this introduction, we describe a method of using aspects at the modelling level and weaving them into a single integrated model. Finally, we extend this framework with the automatic construction of analysis models based on separate aspect models. In our example, fault tolerance structures are modelled by aspects and the analysis model is a dependability model that is used to determine the non-functional properties of the system like reliability and availability. In this way the separate design of the functionality and the dependability is supported and the design decisions concerning fault tolerance can be analysed on the basis of the dependability model.

Keywords

Aspect Oriented Programming · model-based design · dependability analyzis

Péter Domokos
Department of Measurement and Information Systems, BME, H-1117 Budapest
Magyar Tudósok krt. 2., Hungary
e-mail: pdomokos@mit.bme.hu

István Majzik
Department of Measurement and Information Systems, BME, H-1117 Budapest
Magyar Tudósok krt. 2., Hungary
e-mail: majzik@mit.bme.hu

1 Introduction

As software systems are growing in size and complexity, modularization becomes more and more important. As an early step, procedural programming languages were introduced. An important milestone in the history of modularization is the Object Oriented Programming paradigm (OOP), which provides a way to encapsulate real-world objects or similar concepts into a single unit.

Although OOP is a well-known and widely used paradigm, there may be several concerns in a program that crosscut the boundaries of OOP, thus, they can not be captured this way. These concerns are called *crosscutting concerns*. Typical examples for crosscutting concerns are logging debug information (e.g. class and method call sequences, with their parameters), performance analysis (how often classes and methods are called, how long they run etc.), authentication, persistence etc. Even if these concerns can be modularized into a single object, the calls to these objects are scattered through the code causing *code scattering*. The mission of Aspect Oriented Programming (AOP) is the *modularization of crosscutting concerns* into a single unit [1].

Modelling is an important phase of the object oriented design process. The use of standard models, like the Unified Modelling Language (UML), improves the communication between the designers and provides a basis of both analysis and automated code generation. The need for modelling the crosscutting concerns, i.e., modelling the aspects, and this way integrating the aspect-oriented paradigm into a model-based design environment is a current topic of research. Besides the above mentioned general advantages, the model-based aspect-oriented approach gives us the opportunity to support the design process by *model-based analysis*. Namely, the analysis of non-functional properties like dependability and persistence can be performed more easily if those crosscutting concerns that determine these properties are *modularized at the level of modelling*. The UML-based models of the core system and the aspects can be processed systematically to derive the analysis model that can be solved by well-established tools.

This approach can be illustrated by the analysis of depend-

ability. The basis of dependability modelling in an early phase of the design process is the static structure model of the system under design extended with local dependability attributes of the system components. We aim at *modularizing fault tolerance concepts* (e.g. the management of redundant components) in aspects and using these enhanced models for model-based analysis. In this case the core structure model of the system and the models of the aspects are processed together, and the analysis model is generated in the form of a Stochastic Petri net. The modularization of the dependability-related concerns allows the *automatic construction of the analysis model* and the *re-use* of both the fault tolerance concepts (aspect models) and the corresponding analysis models (subnets in Petri nets). The complete analysis model is solved to derive system-level dependability attributes like reliability and availability.

In this paper, we overview AOP to give the reader an idea about the aspect oriented approach, and summarize the approaches about the modelling of aspect-oriented programs. Then we introduce our proposal for integrating *aspect-oriented modelling and analysis*.

Accordingly, the paper is structured as follows. The next section introduces the idea of the separation of crosscutting concerns, and different approaches to the realization of this idea. This is followed by the introduction of *aspect-oriented programming* through a concrete AOP implementation, AspectJ. The following sections introduce several proposals for *modelling* aspect-oriented systems both from the point of view of structural and behavioural modelling. The next section introduces our proposal for creating *aspect oriented models*, that is, applying the concept of aspect orientation at the modelling level. Section 6 will introduce our approach to the analysis of aspect-oriented models and presents its advantages. The final section concludes the paper.

The new results in our paper can be summarized in the following. We elaborate an approach and the corresponding notation for aspect-oriented modelling in UML, and present a method to derive analysis model on the basis of these aspect-oriented UML models. The method is applied in the case of dependability analysis in the early phase of system design.

2 Separation of Crosscutting Concerns

Separation of concerns is an important principle in software engineering. OOP is intended to map the concerns into classes according to real-world objects or concepts. However, there are concerns that do not fit these modules; instead, they crosscut the boundaries of classes. These are called *crosscutting concerns*. Aspect Oriented Programming (AOP) aims at the encapsulation of the crosscutting concerns.

There are several approaches to the separation of the crosscutting concerns. Such approaches are the ones used by AOP implementations like Hyper/J [9] or AspectJ [2], and reflective programming also realizes this concept. In the following, we give a short introduction to Hyper/J and reflective programming, and a more detailed introduction to AspectJ.

Hyper/J is based on the concept of the multidimensional separation of concerns (MDSOC). Hyper/J modularizes the crosscutting concerns into *hyperslices*.

Hyper/J allows developers to identify and noninvasively encapsulate new concerns, including concerns that affect and are scattered across, and tangled within existing software. This capability is called on-demand remodularization: the ability to add new modularizations and needed to reflect new concerns, without disturbing any of the existing modularizations and maintain existing relationships between concerns.

Crosscutting concerns are modularized into hyperslices, the supported granularity is at the method level. With Hyper/J, a concern mapping is defined for each software configuration, that is, the same software can be modularized from different viewpoints at the same time.

We opted to follow the concepts and the terminology of AspectJ in contrary to Hyper/J, because we chose AspectJ as the implementation language for a feasibility study about the implementation of fault tolerance using AOP. The reason for our choice was that in our case, there was no need to handle multiple decompositions and the composability of aspects, only a modularized specification of scattered behaviour was needed.

Reflection is the ability of a program to observe and possibly modify its structure and behaviour. Reflection typically refers to runtime or dynamic reflection. In reflective software, there are two levels connected by the metaobject protocol: (1) the base level (containing objects that implement the business logic) and (2) the meta level (containing meta objects that influence the behaviour of the business logic objects). Each base level object is connected to a meta object.

"Mechanisms" are redirected at the meta level, that is, instead of a method call, an appropriate method of the meta object is called which can forward the request to the base level object. Fault tolerance can be implemented in the meta objects: the meta objects retrieve the requests sent to the base level objects, they implement the appropriate fault tolerance logic, and return the results.

However, it seems that AspectJ is nearer to our concepts. In case of reflection, the intervention points are designated in the base level object, while in case of AspectJ, the intervention points (*join points*, in AspectJ terminology) are specified with the crosscutting concern (*advices* in AspectJ terminology).

3 An Introduction to Aspect Oriented Programming

In this section, a short introduction is given to AOP concepts and programming structures as they appear in AspectJ, an AOP implementation for Java [2]. Note that this section is not intended to teach AspectJ programming, but to give an idea about the possibilities of AOP. Following this introduction, the modelling approaches of AspectJ programs will be discussed.

3.1 Aspect Oriented Programming Using AspectJ

AOP, and more specifically, AspectJ introduce several concepts. The *core concern* is the basic functionality of the system, i.e. the business logic. The *crosscutting concerns* are the functions that crosscut the boundaries of traditional programming constructs, e.g. logging or performance monitoring of selected code pieces. *Aspects* are the units that modularize the crosscutting concerns, similar to classes in case of OOP. An *advice* is contained by an aspect, and it is a piece of code that is inserted at one or more specific points of the core concern (e.g. at the start of specific methods). The advice can be executed either *before* the original piece of code is executed or *after* the original code was executed, or it can be executed *around* the original piece of code (manipulating the input parameters, changing the return value or making a decision, whether the original code should be executed at all). A *join point* is a point in the execution, where an advice is inserted (e.g. at the start of specific methods). A *pointcut* is a language construct that designates a join point.

In the following, a few types of pointcuts are introduced through examples.

Join points can be designated at method calls, including the call to the constructor. The pointcut can be restricted by the visibility, the return type, the class and the signature of the method, but some or all of these parameters can be omitted. The advice is executed after the arguments of the method are evaluated, but before the method itself is called. Examples: call (public void MyClass.myMethod(String)) designates a join point at the call of myMethod in MyClass that takes a String argument, the return type is void and has public access. As another example, call (MyClass.new(..)) points to the call of MyClass's constructor, with any signature. The call (* *.myMethod(..)) pointcut designates any myMethod call in any class in the default package, while call (MyClass+.myM*(String, ..)) defines a pointcut to any method starting with myM in MyClass or in any of its subclasses, taking a String as a first argument.

Field access pointcuts capture read and write access to a field (attribute) of a class. For example, get (PrintStream System.out) designates read access to the field out of type PrintStream in class System, while set (int MyClass.x) designates write access to the field x of type int in MyClass. It depends on the advice type (*before*, *after* or *around*), whether the corresponding advice is executed before, after or around the variable access designated by the pointcut.

Exception handler pointcuts capture the execution of exception handlers. For example, handler (IOException+) designates the execution of catch blocks handling IOException or its subclasses.

This is a far not complete introduction to AOP or AspectJ, it is only intended to give you an idea how AOP is used. For more informations, refer to [2] and [3].

3.2 Modelling Aspect Oriented Programs

There is a demanding need for model-based software development. The Unified Modelling Language (UML) is a well known, industry standard modelling language for object oriented systems. UML does not support the concepts introduced by AOP, but it is extensible. The extension mechanism of UML introduces *stereotypes* that allow the creation of subtypes of model elements at the meta-level and *tagged values* that extend the model elements with additional attributes. Using stereotypes and tagged values, a UML *profile* can be defined to model a specific domain. Accordingly, UML can be extended with new constructs to be able to express the concepts introduced by AOP.

There are at least two interpretations of the expression "Aspect Oriented Modelling" depending on which "meta-level" aspect-orientation is interpreted: (1) we can talk about *the modelling of an aspect-oriented program*, in which case, the crosscutting structures and language-specific artifacts must be denoted in a diagram, or (2) *the model itself can be aspect-oriented*, that is, the model can consist of core and crosscutting concerns that can be woven into an integrated model (e.g. for the sake of code generation). In the following, these interpretations will be discussed. In this section, we discuss the first interpretation, while our concept of aspect-oriented modelling is introduced in Section 5.

The modelling of aspect-oriented programs aims at creating models that depict the core concern, the crosscutting concerns and the join points. There are two basic approaches in this case: (1) the definition of a UML profile which allows the use of existing UML tools, or (2) the extension of the UML metamodel and the definition of an appropriate visual notation.

The first approach, i.e. the definition of an UML profile, is followed in [4]. The aspect can be considered as a subtype of Class in the original UML metamodel. Accordingly, the aspect appears in the class diagram as a class stereotyped ⟨⟨aspect⟩⟩, and contains the advices as operations. The advice type is denoted with a stereotype associated to the operation. The valid values are the following:

- ⟨⟨**before**⟩⟩**:** the advice is executed before a given point (designated by the pointcut) of the extended program,
- ⟨⟨**after**⟩⟩**:** the advice is executed after a given point of the extended program,
- ⟨⟨**around**⟩⟩**:** the advice can decide whether to execute the matched join point, and if yes, at which point of the advice,
- ⟨⟨**replace**⟩⟩**:** the advice modifies a given point of the extended program implementation by replacing it, or by modifying the extended objects implementation,
- ⟨⟨**role**⟩⟩**:** the advice can be invoked on the objects that are extended by the aspect-class that the advice belongs to, moreover, the advice implementation can access the extended class attributes and the aspect-class attributes.

So far, a notation is defined for the definition of aspects and advices. A notation is still necessary for pointcuts, that is, for the designation of the point(s) in the extended class(es), where the advice needs to be executed. For this designation, the authors propose the use of stereotyped associations. A stereotype associated $\langle\langle$pointcut$\rangle\rangle$ leads from the aspect to the crosscutted class. Roles are used to designate both the exact location, where the advice is executed, and the advice itself. The role name ($r1$) at the aspect class denotes the advice, while the role name ($r2$) at the crosscutted class denotes the crosscutted method.

The form of $r2$ is *T[expr]*, where $T \in \{!, ?, \langle N\rangle, \langle U\rangle, \langle C\rangle, \langle R\rangle, \langle E\rangle, \sim, !\sim, \hat{}\ \}$ and *expr* is an optional expression that specifies the T set of points. ! denotes method invocation points, and it can be followed by a regular expression that matches the methods names or by a logical expression containing keywords like *ALL*, *CONSTRUCTORS*, *SETTERS*, *GETTERS* etc. A ? denotes the method execution point, followed by the same expressions as !. (For a complete introduction of this notation, please refer to [4].) Fig. 1 (a) depicts the notation of pointcuts introduced here, and Fig. 1 (b) introduces the notation of advices.

The second approach introduces the *extension of the UML metamodel* to support AOP constructs. However, such heavyweight extensions are complex to implement, especially in the case of a rapidly evolving technology, like AOP. Therefore, the authors of [5] follow a slightly different approach defining a metamodel for AspectJ, which can be used for forward and reverse engineering between AspectJ models and AspectJ programs. This approach requires much less effort if one is only interested in building a dedicated tool supporting AspectJ, rather than extending an existing UML CASE tool. For interoperability with other tools, it is sufficient to define a mapping of the metamodel to UML.

Fig. 2 depicts how the AspectJ metamodel is derived from the UML metamodel. First, the UML metamodel is specialized to express the static structure of the Java language. The Java metamodel is built tailoring the UML metaclasses to the Java 2 specification, eliminating irrelevant generality (unnecessary attributes and associations). Most of the remaining metaclasses use the same names as the corresponding UML metaclasses, except ArrayType (which corresponds to MultiplicityElement in the UML metamodel), Field (Property), Constructor (Method and Operation).

Fig. 3 depicts a modified version of the AspectJ metamodel. We modified the original metamodel proposed in the paper in order to be able to provide a graphical representation that is supported by standard UML modelling tools.

The AspectJ metamodel adds to the Java metamodel the concept of *pointcut*, *advice*, *inter-type declaration* and *aspect*. Pointcuts and advice affect program flow, inter-type declarations affect a program's class hierarchy and features, and aspects encapsulate these new constructs.

The modified AspectJ metamodel expands four Java metaclasses: Class, Generalization, Feature and Operation with new

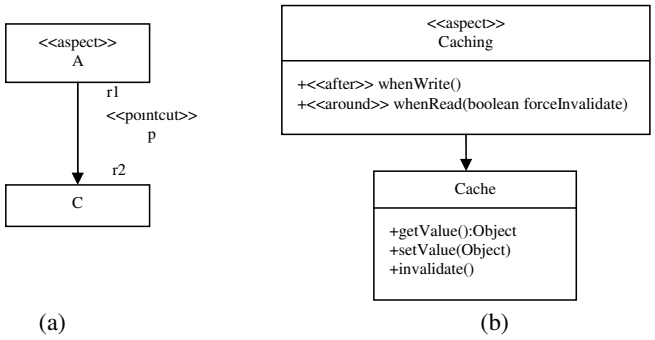


Fig. 1. A notation to depict AOP constructs in UML

associations, and it adds eight new metaclasses: Aspect, Pointcut, PrimitivePointcut, DeclareSoft, DeclareWarning, DeclareError, DeclarePrecedence and Advice.

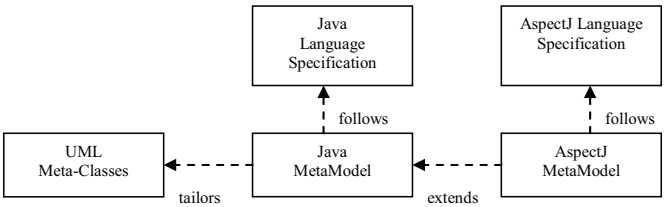


Fig. 2. The AspectJ metamodel extends the Java metamodel

The notation we propose is demonstrated in Fig. 4 through an example. This example is introduced below.

The core program consists of a single class *Communication* that has a *sendBuffer* attribute and a *send* command. The *send* command sends the *sendBuffer*'s contents to the specified output (in the running example, to System.out).

Before sending the message, encryption should be added in an aspect. The *EncryptAspect* introduces a new attribute, *encryptedBuffer* in the *Communication* class which will contain the encrypted message. An *around* advice, named *readBuffer* takes care to read the *encryptedBuffer* instead of *sendBuffer* whenever the *send* command is activated. The *before* advice *encrypt* encrypts the message, if the *send* command is called. (By encrypting the message on the calling of the *send* command, and replacing *get* access to the *sendBuffer* with access to *encryptedBuffer*, it is assured that the encryption takes place only once even if the buffer is accessed several times in the *send* command e.g. for calculating a checksum.)

Finally, a warning is declared using the *DeclareWarning* construct. The read access to *sendBuffer* in the *send* command was replaced by read access to the *encryptedBuffer*, therefore, modification of the *sendBuffer* is likely to be an error. If *sendBuffer* is modified in the *send* method, a warning is given by the compiler.

4 Separation of Concerns in Redundancy Management

As software systems are used in high availability and safety-critical systems (where the failure of the software can lead to major loss, or even injuries or death), the dependability of software is an important issue. The dependability can be increased by using more dependable components and/or applying redun-

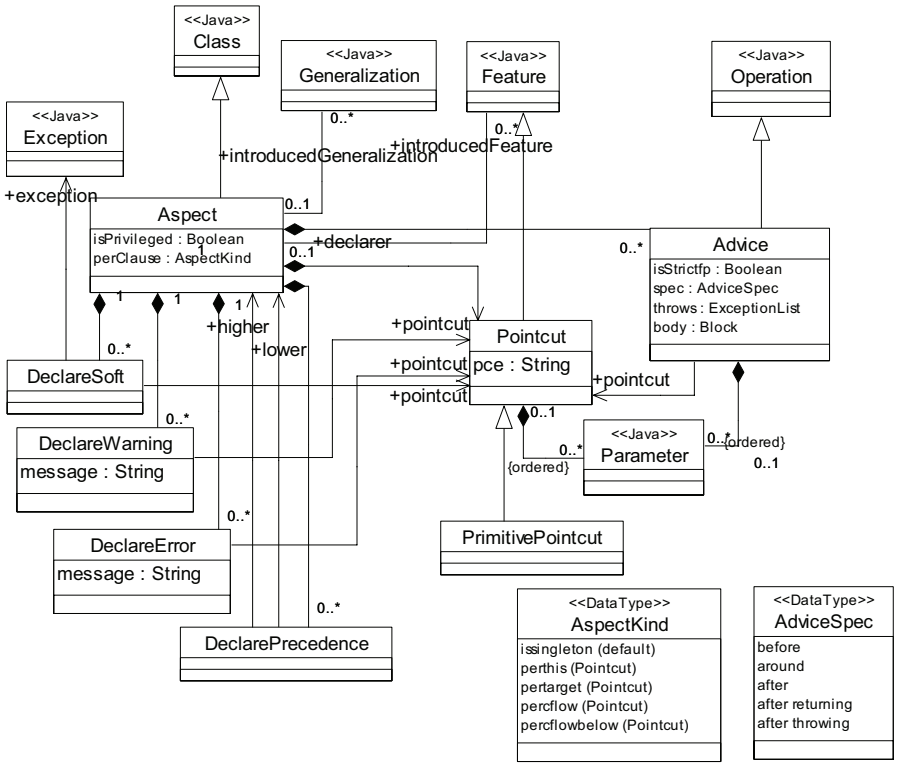


Fig. 3. The modified AspectJ metamodel

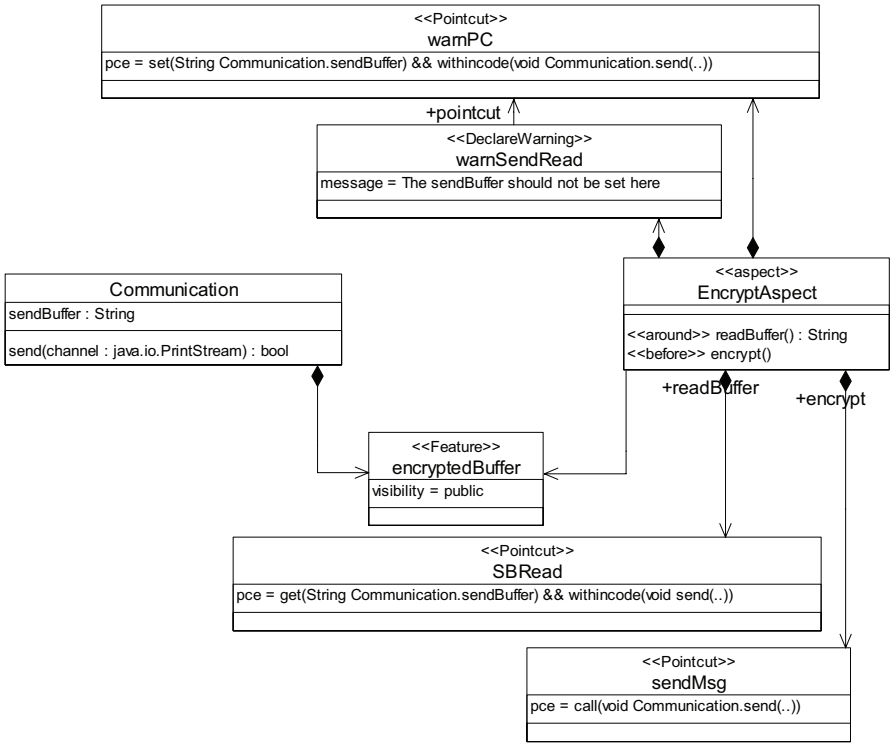


Fig. 4. Graphical notation

dancy. Neither way is effective on its own: increasing the quality of components is expensive and has its boundaries; and poor quality components will result in poor quality system, no matter what redundancy is applied.

In this paper, we focus on the application of redundancy, more specifically, how the redundancy management can be separated from the business logic.

The separation of redundancy management code at program code level is a well-researched area. Library calls to pre-defined mechanisms [11], reflection [12] and meta-object protocols [13] are mature and well-tried techniques that address the separation of dependability and functional requirements.

We examined the applicability of AOP for the separation of redundancy management in case of legacy code. We opted for AOP for the following reasons:

- In the case of library calls, the non-functional activities like redundancy management are collected into the library, but calls to the library functions are scattered in the original code. AOP provides a more clear separation by supporting the modularized implementation of the crosscutting concerns.
- In contrary to AspectJ, the previous techniques do not allow fine grade parameterization of the modifications, e.g. by supporting name-based, property-based, location- or caller-specific modifications.

AOP has already been examined from the viewpoint of separation of concerns in case of *concurrency* and *transactions* in [10]. The conclusion is that AOP is hard to use and its use requires great attention in those cases; however, the reason for this is that these concerns are "part of the phenomenon that the objects should simulate". That is, in case of concurrency and transactions, the authors of the paper were trying to make a *semantical separation*, they were trying to separate concerns that are *semantically coupled*.

In the case of redundancy management, a *syntactical separation* is needed; that is, the redundancy management is a *crosscutting concern*, and not a *part of the semantics of the logic*.

We have already examined the separation of redundancy management at the code level using AOP. In this paper, we examine how these concepts can be used at the modelling level to separate the modelling of the business logic and redundancy management, and to aid the automated construction of dependability models.

5 Using the AO Concept for Modelling Redundancy Structures

aspect-oriented modelling (AOM) aims at applying the AOP concepts *at the modelling level*, and providing a modularization of crosscutting concerns in the model. In the case of AOP, a *weaver* integrates the aspects and the core, and e.g. in case of AspectJ, produces an *integrated source code*. In the case of AOM, the core and the aspects are UML model pieces, and the weaver produces an *integrated model* based on the core and the aspect models. The integrated model can be used further as if it was a traditional UML model created by the designer, e.g. for code generation, further refinement or documentation. For analysis purposes, it is advantageous to deal with the separated core and aspect models, as we will present in Section 6.

Note that the clear distinction of core and aspect models helps the designer to separate functional and non-functional design (the functional architecture can be designed without dealing with the crosscutting concerns) and to re-use the aspect models. Moreover, changes in the functional design and in the crosscutting concerns can be performed independently, and traceability and assessment of these modifications is provided.

The modelling of aspect-oriented programs introduced the concept of aspect classes containing the advices, and specific notations for pointcuts. We will follow this approach of aspect modularization, but extend the code weaving to *model weaving* and separate a *weaving layer* in the model in order to support the re-use of advice.

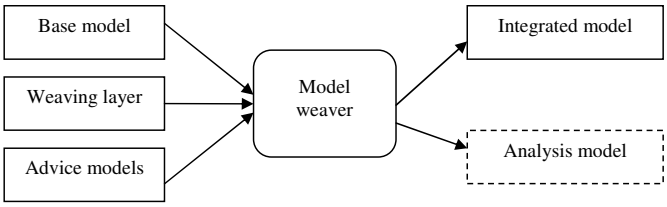


Fig. 5. aspect-oriented modelling process

Fig. 5 depicts our approach to the aspect-oriented design. The model consists of (1) a *base model*, which is the core concern, (2) *advice models*, which are the crosscutting concerns and (3) a *weaving layer* that provides a designation, how the base and advice models are to be integrated. The model weaver creates the *integrated model*, which can be input of further refinement or program source code generation. Additionally, on the basis of the base model, the advice models and the weaving layer, *analysis models* can be constructed (e.g. a dependability model for dependability analysis).

In case of Aspect Oriented Programming, the situation is rather straightforward: crosscutting concerns are encapsulated in individual modules and are woven into the core concern as designated by the pointcuts. Pointcuts are defined with literals according to a grammar defined in the language specification. This way, the definition of the core concern, the crosscutting concern and the pointcuts is homogeneous: all are defined in a *textual language*.

In the case of UML models, a graphical notation is used. Naturally, UML models also contain textual information (like the name of a class, but also stereotypes that contain information about the type of a class), but the model entities and their relations are expressed graphically. Therefore, in order to keep the readability of the models, a graphical notation is needed to express pointcuts. This enforces two opposite requirements in

case of AOM: in order to express relations between the components graphically, they need to be on the same diagram; which is conflicting with the aim of aspect orientation to handle these concepts separately.

This contradiction can be resolved by using the modularization techniques of UML. The base model, the advice models and the weaving layer are created in separate packages – but in the same model space. This allows the designer to represent the core concern on diagram C, an aspect on diagram A, and to create connections between the classes of C and A on a separate diagram, W, which is contained by the weaving layer. The model space built according to this profile forms the input of the *model weaver*, which constructs the *integrated model*. In the following, our approach is introduced through an example.

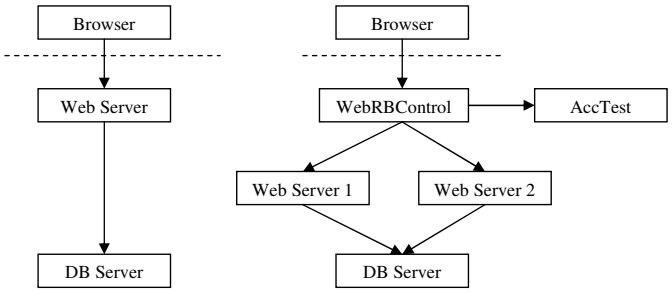
In this example, we focus on *fault tolerance* as crosscutting concern and later (in Section 6) aim at the dependability analysis of the system built from known components [6].

Our example is a web server, which accepts connections from the client browsers, performs some tasks and returns the results. The web server uses a dependable database server (DB Server). The dependability bottleneck in this configuration is the web server, which performs complex computations and therefore may be overloaded, or crashed and therefore unable to fulfil some or all of the requests. The example aims at making the web server redundant using the *Recovery block* fault tolerance scheme. Fig. 6 (a) depicts the basic system. The dashed line represents the border between the server and the client system.

Fig. 6 (b) depicts the redundant system. In the recovery block fault tolerance scheme, the component that is made redundant is replaced by a *recovery block controller* unit. This provides the interface to the system that it expects. In our case, the WebRBcontrol will accept the connections instead of the web server and provide the answers. This controller does not process the requests, it simply forwards them to one of the *variant*s. A variant is an implementation of the task to execute. There are a number of variants which can be used by the controller to perform the task. An *acceptance test* is also available, which checks if the answer seems to be correct. If yes, the result is returned, if no, and there is still a variant that has not yet executed this task, then this variant is executed. In our case, the variants are Web Server 1 and Web Server 2, and the acceptance test is rather simple: if a variant provides a syntactically correct answer within a given time, then it is considered correct. If a timeout occurs, the request is forwarded to the other server. The flowchart in Fig. 7 illustrates the recovery block scheme.

According to the aspect-oriented modelling approach, the Web Server and the DB Server belong to the core concern, that is, these objects form the *base model*. The fault tolerance structure, that is, the recovery block controller, the acceptance test and the additional variants belong to the crosscutting concern, that is, they form the *advice model*. The designation how these constructs are integrated belongs to the *weaving layer*.

In the following, our proposed notation for the representation



(a) Basic configuration (b) Redundant configuration

Fig. 6. The structure of the example system

of the weaving information in UML is introduced through this example. Note that the base model and the aspect model are common UML diagrams.

The top level of the design contains 3 kinds of packages: one package containing the core concern (stereotyped as *Core*), one containing the crosscutting concern (the advice model as a fault tolerance structure stereotyped as *FTS* in our case) and one package containing the weaving layer (stereotyped as *WeavingLayer* in Fig. 8).

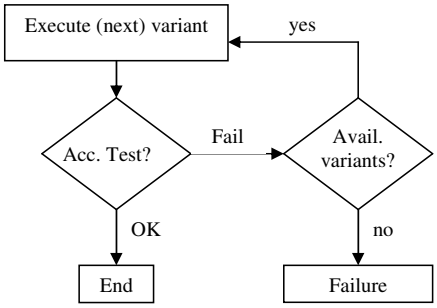


Fig. 7. The flowchart of the Recovery block pattern

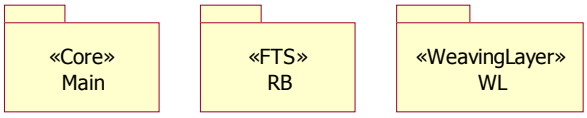


Fig. 8. The top level package contains the Core, the FTS and the WeavingLayer

As depicted in Fig. 9, the base model (the *Core* package) contains a single web server and a database server connected to each other. Fig. 10 shows the advice model (the *FTS* package), that is, in our case, the recovery block structure. It consists of a controller (RBControl), two variants (Variant1 and Variant2) and an acceptance test (AccTest).

All that in left is the weaving layer designating that we want the RBControl to replace the Web Server, and we want to use web servers as variants. Before defining the weaving layer, the following properties of our approach have to be emphasized:

- The advice is constructed in the form of a *design pattern* that is available in a library of patterns. In this way the re-use of the advice is directly propagated. Moreover, it is easy to

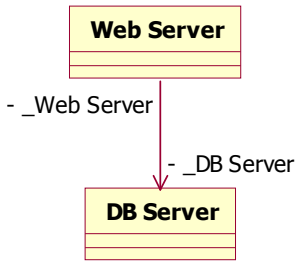


Fig. 9. The base model

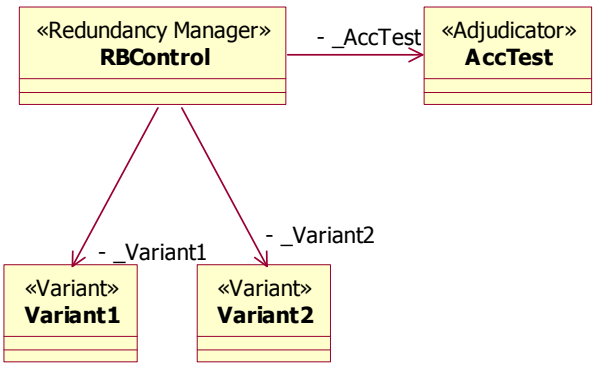


Fig. 10. The advice model

replace the advice model by another one if required (e.g. when the recovery block concept turns to be not optimal for the given application).

- Since the model of the advice is general, it does not contain implementation details that are specific to the core concern. Accordingly, the weaving layer has to contain all details that are required to integrate the advice with the core model. Besides the designation of the join points (i.e. pointcuts in the AOP terminology), the implementation of the abstract classes defined in the advice has to be provided as well.

Accordingly, Fig. 11 depicts the implementation of the variants, the redundancy manager and the adjudicator in the weaving layer. The Variant1, Variant2, RBControl and AccTest classes originate from the corresponding aspect model (⟨⟨FTS⟩⟩ RB in our case). Variant1 is implemented by Web Server 1, Variant2 is implemented by Web Server 2, the RBControl is implemented by WebRBControl and the AccTest is implemented by a TODetect time-out detector.

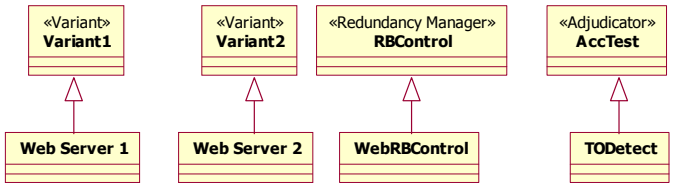


Fig. 11. Refinement of abstract classes

Fig. 12 depicts the pointcut itself, i.e., the *replaces* relation between the WebRBControl and the original Web Server. This relation designates that the WebRBControl plays the role of the Web Server in the integrated (woven) model. Therefore, the

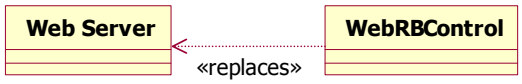


Fig. 12. Replacement of Web Server

Web Server will be removed from the original model during the weaving process, and all of its associations, relations (if any) will be inherited by WebRBControl. In our example, the association between the Browser and the Web Server will appear as an association between the Browser and WebRBControl in the integrated model.

Note that other relations (like e.g. the removal of a base class) can also be specified by a stereotyped UML dependency relation. Using this notation, aspect-oriented models can be created that can be woven into an integrated model using a model weaver.

6 Analysis of System Properties on the Basis of Aspect Oriented Models

The aspect-oriented design of information systems supports the creation of *analysis models* on the basis of the aspect-oriented model of the system. The approach is demonstrated in this paper by the systematic construction of a system-level dependability model (as the analysis model) on the basis of the structural UML model of the system.

The dependability model of a system is a mathematical model that describes the failure and repair processes of the components and the error propagation among them. The computation of the system level attributes like reliability and availability is performed on the basis of the analysis (sub)-models corresponding to these processes. Stochastic models like Stochastic Petri nets (SPN) having sophisticated tool support and solution methods are used for this purpose.

To allow the construction of the system-level dependability model, each component of the system is assigned a *failure* and a *repair* SPN subnet that models the failure and repair process of the given component. Each *error propagation path* between two components is assigned an error propagation subnet that describes the error propagation between the components. The construction of the dependability model is rather straightforward, if fault tolerance is not in scope, because components are assigned similar subnets that are customized using the appropriate dependability attributes available in the structural model (UML class diagram) [7].

However, fault tolerance applied in the system modifies this simple transformation. Since fault tolerance structures are used to detect and mask failures of components, the traditional error propagation model is not appropriate. A more involved subnet shall describe the non-trivial error propagation among subsystems consisting of redundant components and their clients. Typically, a *fault tree* (more exactly, the *mathematical model of a fault tree*) can be used to describe under which conditions the

failure of one or more components is propagated to a higher level. Note that the fault tree is specific to the applied fault tolerance pattern that determines the "management" of redundant components.

The systematic construction of the dependability model involves the following tasks:

- The dependability subnets of non-redundant components (including the error propagation among them) are assigned mechanically. As an example, failure subnet of a stateful component is presented in Fig. 13 (the possible subnets and their construction are described in detail in [7]). In this SPN subnet, three places represent the possible states of the component: a token in place H (healthy) represents a correct status, while a token in place E (error) represents that the component is in an erroneous state. The transition from H to E occurs with an intensity proportional to the fault occurrence. If there is a token in E, a token can be injected into place F (failure) which represents the case when the erroneous status of the component results in a failure which can be observed outside of the component. The stochastic parameters of the subnets are derived from the extensions (tagged values) introduced into the UML model. The connection of subnets is possible through the interface places H, E, and F that represent the healthy, erroneous and failure status of the components, respectively.

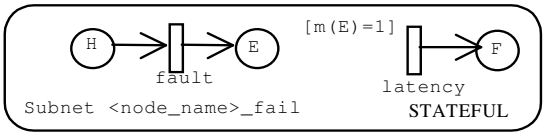


Fig. 13. Failure model of a component

- The fault tree-like error propagation subnets of the fault tolerance patterns (containing redundant components) are *defined a priori in the library of design patterns* together with the functional model of the pattern. A fault tree with an OR gate and the corresponding SPN subnet is illustrated in Fig. 14. Note that the SPN subnets in the library of patterns are defined using the UML notation according to the grammar defined by the *Petri net metamodel* (Fig. 15). In this way functional and analysis models can be handled in the same model weaving framework.

- The analysis model is constructed in parallel with the integrated UML model during the weaving process. The components of the core model are assigned the default failure, repair and error propagation subnets as described above. The weaving layer contains and modularizes all information (in the form of references, refinement relations and dependencies) which is needed to determine the mapping of the abstract classes of the redundancy pattern to the classes of the implementation. Following this mapping, the error propagation

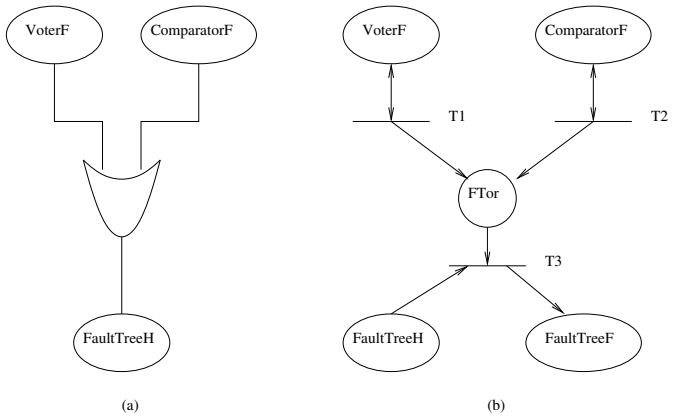


Fig. 14. A fault tree (a) and its SPN representation (b)

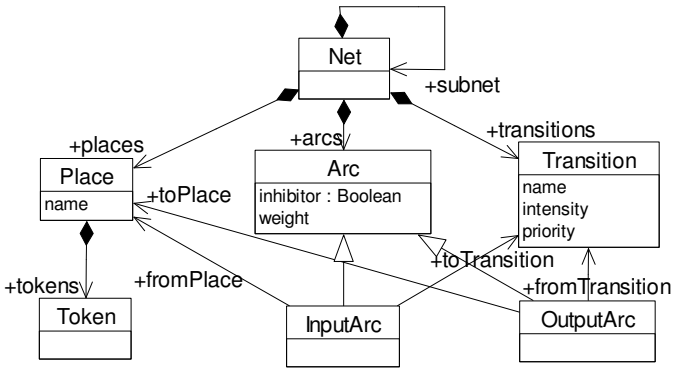


Fig. 15. The Petri net metamode

subnet belonging to the redundancy pattern can be connected to the interface places of the subnets of the proper implementation classes. This is exactly what the model weaver does when it creates the system level dependability model: when the redundancy pattern is woven into the integrated model, the subnets belonging to the newly introduced classes are also inserted into the model and the dependability subnet of the redundancy pattern replaces the default one. The interface places are connected according to the associations inherited and the mapping defined in the weaving layer.

If more detailed information or experience is available then a dependability expert can refine both the default subnets of the core model and the subnets of the implementation classes defined in the weaving layer. Without this manual refinement, the construction of the dependability model is performed fully automatically by the model weaver. Finally, the dependability model serves as the input of an appropriate analysis tool (e.g. SPNP).

7 The Workflow of Aspect Oriented Design and Analysis

The system design process starts with the construction of the core model by the designer of the business logic (7). Then the dependability expert can work on the weaving layer by linking together the implementation of the variants and selecting the fault tolerance structure.

The integrated model and the analysis model are constructed automatically in the weaving process. The dependability expert

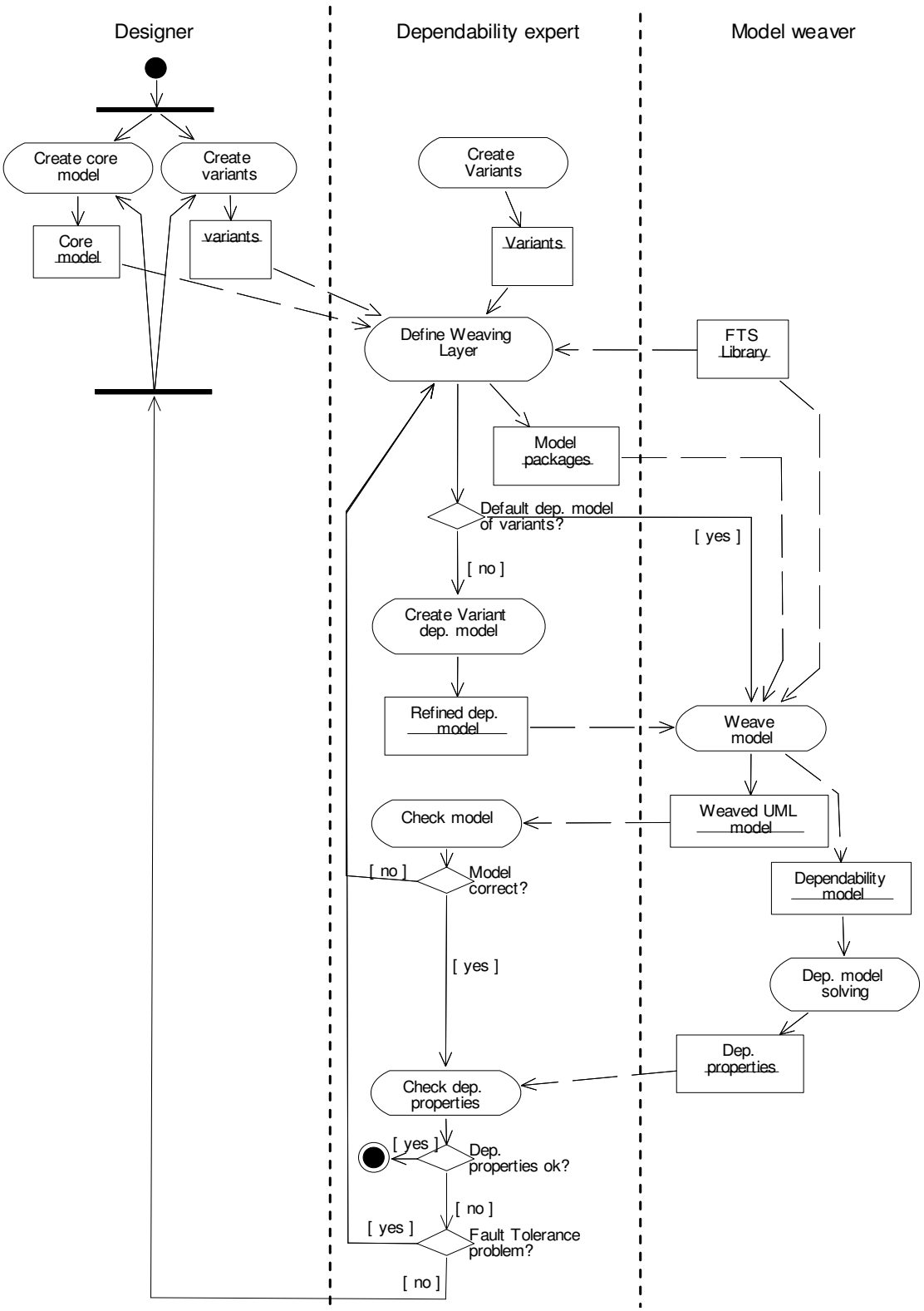


Fig. 16. Aspect-oriented design of system architecture with dependability analysis

can initiate the solution of the analysis model. Based on the results, the core model and/or the weaving layer can be refined or modified (e.g. a different fault tolerance scheme can be selected), and another iteration can be executed until the dependability requirements of the system are fulfilled.

8 Conclusion

In this paper, we first shortly introduced the concept of aspect-oriented programming and gave an overview of approaches that aim at modelling of aspect-oriented programs.

On the basis of this introduction, we presented our approach of aspect oriented modelling and analysis. We described a method of creating aspects at the modelling level and weaving them into a single integrated model. We extended this framework with the automatic creation of analysis models based on the aspect-oriented model and the analysis sub-models assigned to the advice. Finally, we introduced the workflow of aspect-oriented design and dependability analysis.

Our approach is characterized by the following advantages [6]:

- *Separation of functional and non-functional design*: The functional architecture is designed without dealing with fault tolerance and redundancy management issues. Design decisions related to fault tolerance are modularized in a separate model package called the weaving layer that is used to designate those locations of the architecture that need redundancy support and select the architecture pattern to be applied.
- *Reuse of fault tolerant architecture patterns and analysis sub-models*: A library of design patterns is provided. Moreover, the architecture patterns are assigned the corresponding analysis models.
- *Automatic construction of the integrated architecture and the analysis model*: Based on the information available in the weaving layer, our model weaver constructs automatically both the integrated model of the application (available for further refinements) and the corresponding dependability model (available for solution by external solvers). In this way fault tolerance mechanisms can be systematically analised when they are integrated into the system, supporting this way the selection of optimal solutions.

We provided prototype tool support for UML architecture diagrams (class and deployment diagrams). The notation for the weaving layer is defined by using the standard extension mechanism of UML (stereotypes), and the model weaver is implemented as a plug-in of the Rational Rose UML modelling tool. The result of the model weaving process is generated as a separate UML package. The dependability model is to be exported from the UML tools as an SPN analised by SPNP [8]. A few patterns of the design pattern library are already defined, but the library has to be extended in the near future.

References

1 **Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier JM, Irwin J**, *Aspect Oriented Programming*, Proc. European Conference on Object-Oriented Programming (ECOOP), Springer Verlag, 1997. LNCS 1241.

2 **Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG**, *Getting started with AspectJ*, Communications of the ACM **44** (Oct. 2001.), no. 10, 59-65.

3 **Laddad R**, *I want my AOP! Separate software concerns with aspect oriented programming*, January 2002, available at `http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect_p.html`. JavaWorld.

4 **Pawlak R, Duchien L, Florin G, Legond-Aubry F, Seinturier L, Maretlli L**, *A UML Notation for Aspect Oriented Software Design. AOSD 2002.*, available at `http://www2.umassd.edu/swsoc/workshops/aosd2002/asoduml.html`.

5 **Yan Han, Kniesel G, Cremers AB**, *Towards Visual AspectJ by a Meta Model and Modeling Notation. AOSD 2005.*, available at `http://dawis.informatik.uni-essen.de/events/AOM_AOSD2005/papers.shtml`.

6 **Domokos P, Majzik I**, *Desing and analysis of Fault Tolerant Architectures by Model Weaving* (2005). Accepted to the High Assurance Systems Engineering Conference (HASE 2005),.

7 **Majzik I, Pataricza A, Bondavalli A**, *Stochastic Dependability Analysis of System Architecture Based on UML Models*, Architecting Dependable Systems. LNCS-2677 (de Lemos R, Gacek C, Romanovsky A, eds.), Springer Verlag, 2003, pp. 219-244.

8 **Ciardo G, Muppala J, Trivedi KS**, *SPNP: stochastic Petri net package*, Proc. International Conference on Petri Nets and Performance Models, 1989.

9 **Ossher H, Tarr P**, *Communications of the ACM*: *Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software*, Vol. 44, October 2001.

10 **Kienzle J, Guerraoui R**, *AOP: Does it Make Sense? The Case of Concurrency and Failures*, Technical report (2002).

11 **Birman KJ**, *Replication and Fault Tolerance in the ISIS System*, ACM OS Review **19** (1985), no. 5, 79-86.

12 **Agha G, Frolund S, Panwar R, Sturman D**, *A Linguistic Framework for Dynamic Composition of Dependability Protocols*, 1993.

13 **Fabre JC, Nicomette V, Wu Z**, *Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming*, Proc. FTCS-25, 1995.