# Automated Construction of Dependability Models by Aspect-Oriented Modeling and Model Transformation

Péter Domokos, István Majzik

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Hungary, Magyar tudósok krt. 2.
{pdomokos, majzik}@mit.bme.hu

**Abstract.** In order to support the dependability analysis of a system under design in an early phase of the design process, so-called fault tolerance libraries can be created that contain both the architecture and the analysis model of a given fault tolerance pattern. The concepts of the Aspect Oriented Programming paradigm can be applied at the modeling level to designate how the patterns in the fault tolerance library and the architectural UML model can be integrated. A model transformation is applied to "weave" the architecture model and the fault tolerance scheme into an integrated model, and to transform it into a Stochastic Petri Net dependability model. This paper discusses the implementation aspects of model weaving and analysis model construction.

## 1 Introduction

Fault tolerance mechanisms in critical applications have system-wide effects determining non-functional parameters like reliability and availability. These mechanisms are not part of the business logic, therefore, the initial system architecture must be modified to add fault tolerance to an existing design by integrating aspects like error detection and redundancy management into the functional part of the system.

The Aspect Oriented Programming (AOP, [1]) paradigm aims at extending the OOP by modularizing *crosscutting concerns* that crosscut the boundaries of objects. AOP provides mechanisms to modularize the crosscutting concerns into separate modules and specify how the modules are integrated with the core concern. There are several approaches to AOP, among these Hyper/J and AspectJ are the most widely used ones. Both approaches support the modularization of concepts that have effect on one or more classes of the system model.

Hyper/J [6] is based on the concept of the multidimensional separation of concerns (MDSOC). The approach of Hyper/J to achieve MDSOC is called hyperspaces, while Hyper/J is a tool for its realization for Java.

Hyper/J permits 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 as 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.

Another feature of MDSOC is its ability to handle multiple decompositions of the same software simultaneously: some developers can work with classes, others with features etc.

The AOP in the AspectJ approach provides a mechanism to designate so-called *join points* in the program code (e.g. method calls, attribute reading/writing) and to execute additional code (so-called *advices*) at these points. Join points are designated using specific language constructs, *pointcuts*. An advice can be executed *before*, *after* or *around* (instead of) the original code at the join point. In case of around advices, the advice can determine whether the original code is executed.

A key difference between Hyper/J and AspectJ is that AspectJ supports augmentation of a single model, whereas Hyper/J supports integration of multiple models. In AspectJ, separately coded aspects are used to augment classes and methods. This is useful when a single aspect cuts across many classes, allowing a single, localized specification of scattered behaviour. Aspects augment classes, but cannot augment one another, so aspects are not composable. Aspects often cannot be understood without reference to the model, which may limit their reusability.

For fault tolerance, the introduction of new software elements at specified points of the system is necessary. That is, the object that provides a critical service is replaced or extended by a fault tolerance structure. For this purpose, the concepts introduced by AspectJ are more appropriate as there is no need to handle multiple decompositions of the same software.

In our approach, the concept of AOP as seen by AspectJ is applied to modeling. Fault tolerance is considered a crosscutting concern. According to the AOP's philosophy, the fault tolerance (FT) pattern is designed in a separate package (as an advice). A separate weaving layer is used to specify the integration of the system architecture model (the core concern) and the fault tolerance patterns (crosscutting concerns).

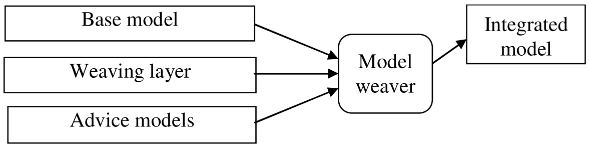

Fig. 1. Aspect-oriented model based design

Our approach to aspect-oriented design is depicted in Fig. 1. The base model, the weaving layer and the advice models are distinguished. In contrary to AspectJ (where aspects contain both the advices and the pointcuts), the advice models and the weaving layer are separated in order to support the reuse of advice models.

The advice models form the fault tolerance library. The advice models are in fact design patterns, that is, only the architecture of the fault tolerance structures is provided. The concrete implementation of the elements is not specified here, therefore, it is not specific to the core concern, which increases the reusability of the advice models.

The system designer is responsible for creating the base model and for specifying the integration of the base model and the FT patterns by creating the weaving layer. An automated model weaver constructs the integrated model.

The aim of dependability modeling is to construct an analysis model from the system model to estimate system-level dependability attributes. Parallel with the integrated model, the analysis model is also constructed based on the information available in the fault tolerance library.

The analysis model for dependability analysis at this architectural level consists of three kinds of analysis submodels: (1) the basic fault activation model of components, (2) the basic error propagation model between non-redundant components, and (3) the error propagation model between the FT pattern and the other parts of the model.

The basic fault activation model and the basic error propagation model are constructed automatically. The same component submodel is used for all components of the same type (stateful or stateless, software or hardware) and the same error propagation model for all relations, because it is only the failure and repair processes of a component that are of interest. These submodels are parameterized with the parameters of the component found in the base model.

However, the error propagation of a fault tolerance scheme can not be modeled with the same submodels, since its purpose is to modify the trivial error propagation e.g. by masking faults. Therefore, the analysis submodel of an FT pattern must be designed by a dependability expert (the designer of the pattern). The analysis model is attached to the library, and the analysis model using these submodels is automatically constructed in parallel with the integrated UML model, as depicted in Fig. 2. Based on the results of the analysis, the design can be refined or modified to compare different architectural solutions. The modifications can apply to the application of the fault tolerance structures, that is, the weaving layer can be modified; or even to the base model (system re-design). Based on the results of the dependability analysis, the FT library can also be modified e.g. by refining the analysis sub-models.

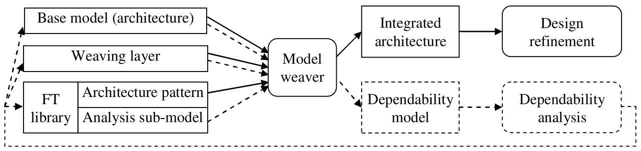

Fig. 2. Model weaving and dependability analysis

Our previous paper [2] introduced the idea of aspect-oriented modeling for dependability analysis. In this paper, the implementation of the model weaver and the transformation to Stochastic Petri Net (SPN)-based dependability model are described. The model weaver and the dependability model construction are implemented in a single *model transformation* (referenced later as model weaver called *uml2pn*).

The input is an XMI file produced by an UML modeling tool, e.g. Rational Rose. The UML model must conform to the notation specified in this paper.

Code generators are available for exporting the integrated UML model in XMI format (e.g. for Rational Rose), and for exporting the constructed stochastic Petri net in PNML [4] and CSPL [5].

## 2 Model Weaver

The model weaver is implemented in the VIATRA model transformation framework [3]. The transformation specification language is based on graph transformation, and is extended with ASM rules for the designing control flow of the transformation. ASM rules also provide a facility for code generation.

UML models are represented as graphs in the following way. The elements that appear as nodes in the UML metamodel are mapped into graph nodes, while the relations between them are mapped into graph edges.

Graph transformation works similarly to traditional Chomsky grammars, but it manipulates graph patterns. In VIATRA, a graph transformation rule can be specified by the specification of the left hand side and the right hand side (LHS and RHS) of the rule, or by the specification of the graph pattern (LHS) and the ASM rules that manipulate the pattern by adding new and/or old nodes and/or edges.

The model weaver is responsible for creating both the integrated UML model and the Petri net dependability model of the system. The weaving information is specified in the *weaving layer* to keep it separated from both the system architectural model and from the fault tolerance library. The weaving layer is a set of pointcuts in AspectJ terminology. It provides information about the integration of a fault tolerance pattern and the system architecture model by using references to model elements and FT pattern elements. If the same pattern is applied more than once, then the specification of each occurrence must be provided separately in the weaving layer.

The notation uses standard UML notations to specify the integration of the base and the advice models. This allows the use of any standard UML tool being able to produce an output file that can be parsed by VIATRA.

## 3 Basic Notation

In this section, a short introduction is given to the notation applied in the weaving layer to designate how the base model and the advice models join. In the current example, a pressure controller is shown which uses two valves to lower the pressure when needed. Should the first valve fail, the secondary valve is opened to avoid an accident.

Fig. 3. depicts the Recovery Block (RB) pattern. The recovery block consists of a controller that orchestrates the other elements, an acceptance test that tests the results of the variants.

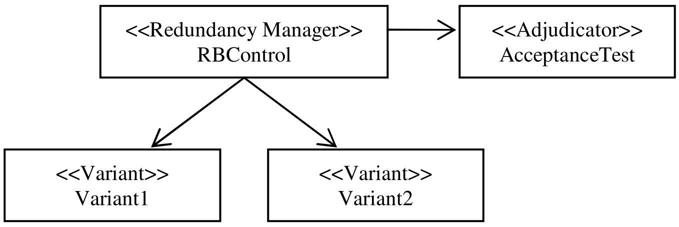

Fig. 3. The Recovery Block pattern

The classes provided by the fault tolerance library do not contain implementation details, because these are application specific. For this purpose, “implementation classes” are provided in the weaving layer that contain the application-specific information (like the concrete dependability attributes). The “implementation classes” are associated to the corresponding classes in the fault tolerance library using UML *generalizations*, see Fig. 4.

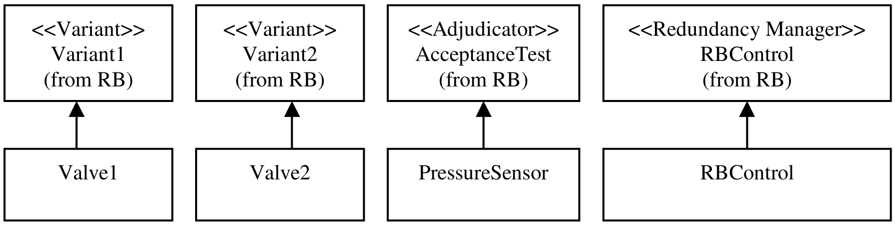

Fig. 4. The implementation classes of the RB elements
(the arrows denote generalization)

Fig. 5. depicts the notion of the pointcut. Dependencies stereotyped with *replaces* denote that the *Valve* is replaced with *RBControl.*

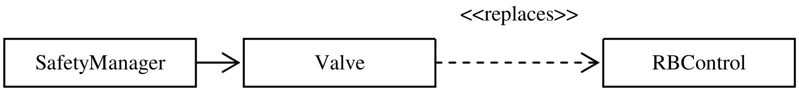

Fig. 5. Notation of a pointcut

## 4 Analysis Submodels in the FT Library

In our approach, the analysis submodels are Petri net models, therefore, the description of the analysis submodels in the fault tolerance library using UML notations requires a way to represent Petri nets in UML. The representation of Petri nets using UML notation is advantageous because both the UML models and the analysis models can be handled uniformly.

The Petri net metamodel shown in Fig. 6. defines the syntax of Petri nets. According to the metamodel, a Petri net contains places, transitions, and arcs connecting places and transitions. Places may contain tokens. To represent a concrete Petri net in UML, stereotyped classes are used to represent the objects (that is, places, transitions, arcs, tokens) appearing in the Petri net, and associations are used to express their connections. The Petri net elements also have attributes, e.g. places have a name.

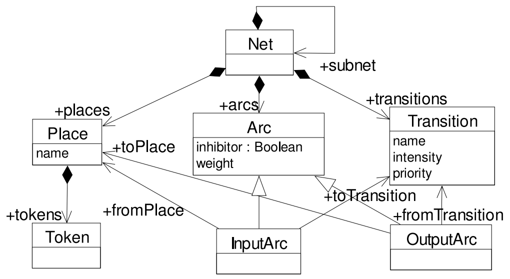

Fig. 6. The Petri net metamodel

The analysis submodels are assigned to the elements of the fault tolerance library using packages. A package is assigned to each component of the fault tolerance structure named identical to the component and stereotyped by *dep_mod*, see Fig. 7. Each component can be assigned a default analysis submodel, or the submodel can be replaced by one created by the dependability expert. In the former case, the appropriate package contains only the abstract *interface places* of the analysis submodel (H, E and F, see Section 5), or the entire Petri net model in the latter case.

The error propagation of the fault tolerance structure can be described e.g. by using a fault tree. The analysis model of the error propagation of the FT structure is described in an additional package stereotyped *interface*. The interface places of each submodel are identified by the package they come from and by their *name* attribute.

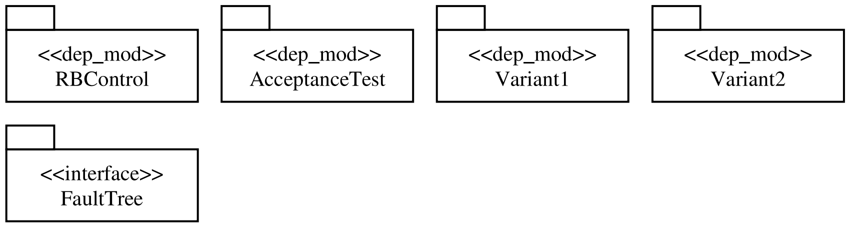

Fig. 7. Assigning the analysis submodels to the RB pattern

The user can also extend the fault tolerance library. The extension of the library with a new element includes both the creation of the UML pattern of the element and the construction of the analysis submodel.

In order to assign a refined analysis submodel to a component of the fault tolerance structure, a package must be created named identical to the element and stereotyped *dep_mod*. The analysis submodel must be created in the package using the UML notation based on the Petri net metamodel. The submodel must contain the interface places, because these will connect the subnet to other subnets of the dependability model.

In order to assign the default analysis submodel to a component of the fault tolerance structure, a package must be created named identical to the element and stereotyped *dep_mod*. Only the interface places of the subnet must be created in the package, which must be abstract.

In order to create the analysis submodel that represents the error propagation in the fault tolerance structure, a package stereotyped *interface* must be created. The Petri net submodel must be created in this package using the UML notation based on the Petri net metamodel. The subnet must contain the interface places, which represent the fault tolerance structure submodel when connecting to other parts of the dependability model. The dependability subnet of the FT structure is connected to the elements composing it (e.g. variants, redundancy manager) through their interface places. The interface places of the given element are referenced in the diagram. That is, the package of the interface places does not change, and the interface places will be connected to the dependability subnet using associations across packages, see Fig. 8. (In UML, model elements from other packages can be directly represented in a diagram and the modeling environment keeps track of their origin, see e.g. the "from RB" notation in Fig. 4.)

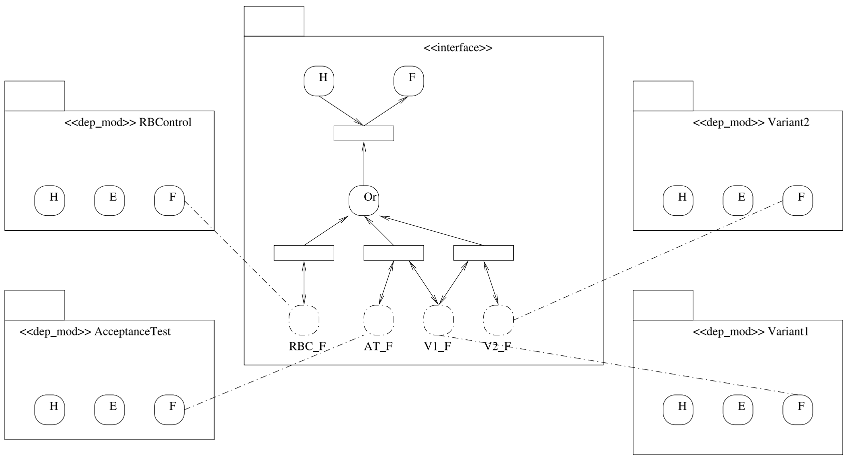

Fig. 8. Connecting the interface places of the FT components to the dependability model of the FT structure

As depicted in the figure, the package stereotyped *interface* contains part of the error propagation model of the recovery block pattern. The token is removed from the *healthy* place and put into the *failure* place if either the *RBControl*, the *AcceptanceTest* and *Variant1*, or *Variant1* and *Variant2* fail.

The dashed places in the *interface* package are the interface places of the appropriate components of the fault tolerance structure. They are only presented on the diagram, but in the model space, they are contained by their appropriate packages as indicated with the dashed lines. When the dependability model is constructed, the interface places of the FT structure components (shown in the corresponding packages) and their representation drawn with dashed line are mapped into one and the same model element.

## 5 Weaving Process

The transformation processes the weaving layer. The weaving layer consists of packages, each package defining an instance of the application of a fault tolerance structure. This separation is necessary because the same FTS may be applied at several points in the same design. The process is iterated for each package in the weaving layer.

First, it looks for the join point where the FT pattern must be inserted (denoted by a dependency stereotyped *replaces* in the weaving layer). The class to be replaced is the *supplier* of the dependency, while the *client* denotes the replacing class. More exactly, the replacing class is the child class of the *client*, since the conrete implementation class is denoted by UML generalization as seen in Fig. 4.

All connections of the original model element (the join point) are copied to the replacing element (that came from the pattern). The original element is not removed at this point, because it may participate in other parts of the weaving layer as well.

## 6 Construction of the Analysis Model

The next step is the construction of the dependability model associated to the integrated UML model. Each component in the system is assigned two kinds of Petri net subnets: (1) a failure subnet describing the failure process of the component, and (2) a repair subnet describing the repair process of the component. The following types of subnets are created:

- Default failure and repair subnets of the system components. The interface of the subnet are the places *H* and *F* indicating the healthy and the failure state of the component, and if the component is stateful, the interface also contains a place *E* indicating the erroneous state of the component. This default submodel is built-in into the weaver.
- Error propagation subnets between components. The error propagation subnets are created along associations between objects and model the fact that with a given probability, the failure of a component is propagated to another one that uses its services. This submodel is built-in into the weaver.
- User-defined failure and repair subnets of fault tolerance structure components. These subnets have the same interface places as the default failure and repair subnets, but model the behaviour of the component more precisely. It is defined in the appropriate package in the fault tolerance library.
- Interface of the fault tolerance structure. This submodel models the error propagation in the redundancy structure. It has the same interface as the default

failure and repair subnets. It is typically a Petri net model corresponding to a fault tree that describes under which conditions the failure of one or more components of the redundancy structure become visible.

During the construction of the dependability model, an instance of the corresponding subnet is created for each model element in the UML model. These subnets are integrated by the weaver in the following way. For each association that leads between two model elements *A* and *B*, an error propagation subnet is created. The interface places of the analysis model of *A* and *B* are connected to the error propagation subnet.

If a component *A* is connected to a fault tolerance structure *F*, then the error propagation subnet connects the analysis model of *A* and the interface of *F*.

During the construction of the analysis model of the fault tolerance structure, first the analysis models of the composing components, and the interface model of the fault tolerance structure are created. The next step is connecting the analysis models of the composing components and the FT structure. E.g., in the case of *RBControl*, the place *F* in the analysis model of *RBControl* is matched with *RBC_F* (see Fig. 8.) The weaver copies the arcs of *RBC_F* to the place *F* in *RBControl*, and removes *RBC_F*.

## 7 Code Generator

The integrated UML model is stored according to the UML metamodel, and the Petri net dependability model is stored according to the Petri net metamodel in the model space of VIATRA. This way, the model weaving and analysis model construction step are isolated from the code generation step. The code generator for a specific language can be implemented easily.

The first step of the code generation is the assignment of a unique identifier to each element in the model space. (This is necessary because XML representations need a unique identifier of model elements so that they can be referred to.)

The code generator must travel the model space, select the elements that are related to its output language (e.g. UML model elements for an UML code generator, or Petri net elements for a Petri net code generator) and generate the output code. The traversal of the model space is directed by the desired output format and by the structure of the model. The output file is written sequentially, which requires to travel the model space in the order the output file requires it.

Code generators are available for exporting the integrated UML model into XML format for Rational Rose, and for exporting the constructed Petri net into PNML [4] and CSPL [5].

## 8 Conclusion

In this paper, we introduced an approach to the automated construction of dependability models of architectural system models in an early phase of the design process using fault tolerance libraries specified using the approach of aspect-oriented modeling.

In this process, the dependability model subnets are stored together with the redundancy patterns in fault tolerance libraries. The integration of the system architecture model and the FT pattern is specified in a separate weaving layer, thus making the reuse of the FT library easy. The FT library does not contain concrete implementations of the fault tolerance structure's components, only the structure of the FT pattern and the associated analysis model is stored, thus, it is not specific to a concrete application. In a concrete application, these elements are instantiated to be tailored to the application.

Due to the separate design of the base model, the fault tolerance library and the weaving layer, it is easy to replace the applied FT pattern at a specific point, or to introduce FT patterns at a new point to analyse the system from the viewpoint of dependablity bottlenecks and to compare different solutions.

A drawback of the current version is that only one-to-one and one-to-many relations can be defined, that is, a component in the base system can be replaced by another component or by a structure; but there is no possibility to replace a structure in the base model. This does not introduce a serious restriction, since typically single classes are to be replaced (they designate a join point for the redundant subsytem). The current implementation consists of 15 graph patterns, 10 graph transformation rules and 7 ASM rules.

## References

1. Kiczales, G. et. al.: Aspect-Oriented Programming. Lecture Notes in Computer Science, Vol. 1241, Springer-Verlag (1997)
2. Domokos P., Majzik I.: Design and Analysis of Fault Tolerant Architectures by Model Weaving. Ninth IEEE International Symposium on High Assurance Systems Engineering, Heidelberg, Germany, 12-14 October 2005., 15-24
3. Generative Model Transformer project on the Eclipse homepage, http://www.eclipse.org/gmt
4. Billington, J., Christensen, S., Kees M. van Hee, Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri Net Markup Language: Concepts, Technology and Tools. Lecture Notes in Computer Science, Vol. 2679. Springer-Verlag, Application and Theory of Petri Nets 2003: 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27 2003., 483-505
5. SPNP User's Manual Version 6.0. http://www.ee.duke.edu/~kst/
6. Ossher, H., Tarr, P.: Using Multidimensional Separation of Concerns to (Re)shape Evolving Software. Communications of the ACM, Vol. 44., No. 10., October 2001.