DRIP Catalyst: An MDE/MDA Method for Fault-tolerant Distributed Software Families Development[1]

Nicolas Guelfi[1], Reza Razavi[1], Alexander Romanovsky[2], Sébastien Vandenbergh[1]

[1]Software Engineering Competence Center
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
Luxembourg, L-1359– Luxembourg
{Nicolas.Guelfi, Reza.Razavi, Sebastien.Vandenbergh}@uni.lu

[2]School of Computing Science
University of Newcastle upon Tyne
UK
Alexander.Romanovsky@newcastle.ac.uk

Accepted for the OOPSLA & GPCE 2004 Workshop on Best Practices for Model Driven Software Development

Abstract

Coordinated Atomic Actions (CAAs) offer a structuring technique for cooperative exception handling in open distributed systems. The DRIP framework embodies in terms of a set of Java classes the CAAs. It has been proven to be successful in IST DeVa and DSoS Projects. This paper communicates DRIP Catalyst, a method for stepwise and rigorous development of complex, fault-tolerant distributed software families. This method is designed to facilitate reuse and instantiation of DRIP. It comprises an MDE process, a UML-based notation and an MDA development support tool. This tool is implemented as an extension of IBM/Rational XDE. DRIP Catalyst has been experimentally validated by developing two service-oriented, fault-tolerant, software family prototypes. It is also designed to allow formal verification of fault-tolerance applications properties.

Keywords: Distributed Object-Oriented Software, Concurrent Exception Handling, Fault Tolerance and Coordinated Atomic Actions, Architecture-centric software development, Model-Driven Engineering (MDE), Model-Driven Architecture (MDA).

[1] This work is supported by the Luxembourg Ministry of Higher Education and Research under the project n°MEN/IST/04/04.

1 Motivation

Distributed embedded software systems must satisfy an increasing set of quality requirements. For instance, consider a Home Care System for insulin therapy conforming to the Ambient Intelligence (AmI) vision [Ahola, IST]. Patients curry wearable devices that embody insulin pumps, thermometers and automated syringes that measure different vital signals and adjust the concentration of blood glucose with exogenous insulin admissions with the goal of achieving normoglycemia.

The function of this system remains more or less classical, i.e., to avoid the hypoglycemia episodes that can cause coma and death, and the hyperglycemia states that cause long term complications such as neuropathy, nephropathy, retinopathy and cardiovascular disease.

However, the quality requirements are much more important. For example, doctors in charge of the patients should be allowed to consult from distance the system to obtain the vital signals and to remote command the wearable devices in a safe, fault-tolerant and reliable manner. In addition, according to the AmI vision, the system should allow the patients to have as much as possible a normal life, e.g., be able to move freely around their home, taking care of the garden and talking with neighbors at their landing. Furthermore, the system should be adaptable and adaptive, that is it should be, in the one hand, easy to change the configuration of the system by adding and removing medical instruments, and, on the other hand, the system should adapt dynamically its behavior, e.g., the therapeutic treatment, to the evolutions of the state of the patient. The AmI vision insists indeed on personalized services to citizens.

An immediate consequence of such evolutions in the quality requirements of systems is an increasing need for new methods devoted to the development of families of systems. In addition, we believe that such methods, whatever their nature could be, model-driven or not, should remain architecture-centric. Indeed, as for example Rick Kazman observes it in [Kazman], "architecture-centric development is crucial to software-intensive systems: architecture promotes communication among stakeholders, captures early aspects of the design and finally by its abstraction level has the ability to be reused elsewhere".

Fortunately, there are Object-Oriented Frameworks [Johnson & Foote] that provide a valuable handle on complexity that underlies the design of a reusable solution for a family of problems. For instance, in the case of our Home Care System, such a framework would provide a ready-to-use solution that satisfies all enumerated requirements, though an important economy of time and resources.

Frameworks are however not in general accompanied with methods. There is then often an important gap between such a solution and cost-effective and timely creation of a family of complex and evolving software systems (relevant to the domain of the framework). Such a method can dramatically improve the framework instantiation process, the automated verification of software properties, and more generally the software lifecycle.

It then makes sense to investigate how to associate a method to an object-oriented framework to facilitate building a family of related software. Recall that a software product line is "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [Clements & Northrop]. Some examples of application families include Web-based systems, data warehousing, and application package integration. Then an architecture-centric method built around a framework provide a product-line architecture.

This has mainly motivated the investigation of an architecture-centric, model-driven method for a successful object-oriented framework called DRIP. Furthermore, there is a growing interest in the area of application-level fault tolerance and cooperative exception handling as the main paradigm for developing structured fault-tolerant architectures; and there is currently no standard-based method for such developments. This effort is conducted by the Software Engineering Competence Center of the University of Luxembourg (SE2C) in collaboration with University of Newcastle upon Tyne (UK), in the context of the CORRECT project. The goal is to develop a framework-specific MDE/MDA method, called DRIP Catalyst, by a combination of model-driven, generative and formal techniques, to support cost-effective, disciplined,

stepwise and high-level instantiation of DRIP. CORRECT focuses more particularly on service-oriented applications family.

The choice of MDE/MDA is motivated by the power of models and their use at different levels that allow separation of concerns, e.g., analysis, architecture design, implementation, in the one hand, and PIMs from PSMs on the other hand. It also captures and structures the information that is needed for generating running application by refinement and, if necessary, for formal verification of systems properties.

This position paper describes the current status of this work. Section 2 provides some background information. Section 3 states the problem. Section 4 describes our solution, the DRIP Catalyst method and illustrates it. Section 5 presents briefly some related works. Section 6 is devoted to the conclusions and perspectives of this work.

2 Background

CORRECT stands for rigorous stepwise development of Complex Fault tOlerant DistRibuted Systems: from ARchitEctural DesCription to Java ImplemenTation. The purpose of CORRECT is to support rigorous stepwise development of complex fault tolerant distributed systems through all phases of the life cycle, including software architecture design phase, with an intention to generate, by model transformation, reliable Java code. In particular, CORRECT plans developing a new UML-based language for formal description of fault tolerance properties and for supporting rigorous refinement of the system models in such a way that the properties are automatically guaranteed through both refinement steps and phases of the life cycle. A methodological guide for the engineering fault tolerant distributed systems based on this work will be developed.

The starting point of CORRECT is techniques for error recovery [Lee & Anderson]. There are two main classes of error recovery: *backward error recovery* (based on rolling system components back to the previous correct state) and *forward error recovery* (which involves transforming the system components into any correct state). The former uses either diversely-implemented software or simple retry; the latter is usually application-specific and relies on an exception handling mechanism. In general, the choice of fault tolerance error recovery techniques to be exploited for the development of dependable systems depends very much on the fault assumptions and on the system characteristics and requirements. There are two well-known techniques offering error recovery: *distributed transactions* and *atomic actions*.

Distributed transactions [Gray & Reuter] use backward error recovery as the main fault tolerance measure in order to satisfy completely or partially the ACID (atomicity, consistency, isolation, durability) properties. However, in spite of all its advantages backward error recovery has a limited applicability. Modern systems are increasingly relying on forward error recovery, which uses appropriate exception handling techniques as the chief technique [Cristian]. Examples of such applications are complex systems involving human beings, COTS components, external devices, several organizations, movement of goods, operations on the environment, real-time systems that do not have time to go back. Service-oriented architectures that are of special interest in CORRECT fall clearly into this category.

Atomic actions [Campbell & Randell] allow programmers to apply both backward and forward error recovery. The latter relies on coordinated handling of action exceptions that involves all action participants.

The Coordinated Atomic Actions (CAAs) concept [Xu et al.] results from combining distributed transactions and atomic actions. Atomic actions are used to control cooperative concurrency and to implement coordinated error recovery whilst distributed ACID transactions are used to maintain the consistency of shared resources. A CAA is designed as a set of participants or roles cooperating inside it and a set of resources accessed by them. In the course of the action, participants can access resources that have ACID properties. Action participants either reach the end of the action and produce a normal outcome or, if one or more exceptions are raised, they all are involved in their coordinated handling. If this handling is successful the action completes normally, but if handling is not possible then all responsibility for recovery is passed to the containing action where an external action exception is propagated.

The CAAs are then a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting components. They allow decreasing the overall system complexity and facilitating its development by structuring it in terms of nested recovery units, while enabling cooperative exception handling. This is why CORRECT adopts the CAAs as the main paradigm for developing structured fault tolerant architectures. The method planned by CORRECT is built on the basis of the DRIP which is a framework for implementing CAAs. The next section provides more details on DRIP.

3 Related work

Helping application programmers to reuse a framework without having to understand in details how it works is a long-term issue. Ralph Johnson observes that documenting frameworks for describing their purpose and design is a way to proceed [Johnson 1992]. However, communicating complex designs is hard [Beck & Johnson], and frameworks have in general complex designs. Design patterns [Gamma & al.]] have been shown helpful, but do not provide an extensive solution to this issue.

As for methods for developing CAA-based fault-tolerant applications, there is some initial work in [Beder *et al.*, Tartanoglu *et al.*, Lemos & Romanovsky, SRDS, Romanovsky *et al.*], but these proposals are not oriented on UML or on standard technologies; they are not suitable for applying refinement techniques (it was clearly not their intention). Finally, they are not specifically oriented towards Java. The UML Profile for QoS and Fault Tolerance [QoS] does not provide a method and does not provide a structuring solution like that of CAAs.

In the context of MDE/MDA, we have not found architecture centric development processes trying to integrating an object-oriented-framework. We have ourselves investigated similar approaches in the context of the FIDJI project[2] since 2001. This effort has resulted in the FIDJI approach and a particular definition of the notion of *architectural framework* [Guelfi, Perrouin]. According to FIDJI, an architectural framework comprises a model-based description of the architectural design of the framework, the framework's code, model refinement mechanisms (transformations) able to specialize the framework and a set of methodological guidelines driving the user through framework instantiation. This approach has been applied in developing an e-business auctioneering product-line called LuxDeal. Its application to the mobile world can be found in [Guelfi, Pruski, Ries].

The idea of applying model-driven engineering techniques as a means for supporting disciplined and stepwise instantiation of a framework is then not new in itself. However, there is a major difference between CORRECT and FIDJI approaches. FIDJI suggests producing UML models of the framework itself. Since, CORRECT suggest to use UML modeling to structure and represent only the extensions to the framework, in other words the way the framework is instantiated.

4 The DRIP Framework and lack of Method for its cost-effective reuse

The DRIP framework embodies in terms of a set of Java classes the CAAs. It has been developed by the University of Newcastle upon Tyne (UK). It has been proven to be successful in IST DeVa and DSoS Projects, as well as in many application areas outside those projects[3].

DRIP stands for Dependable Remote Interacting Processes. It builds on the notion of *dependable multiparty interaction* (DMI) [Zorzo & Stroud]. In a DMI several parties (objects or processes) "come together" to produce an intermediate and temporary combined state, use this state to execute a joint activity, and then leave this interaction and continue their execution outside it. The main properties of a multiparty interaction are synchronization upon entry of participants of the interaction; using a guard to check the preconditions to execute the interaction, hence the need for having synchronization upon entry; an assertion after the interaction has finished to check that a set of post-conditions has been satisfied by the execution of

[2] FIDJI work is supported by the Luxembourg Ministry of Higher Education and Research under the title I project n°MEN/IST/01/001.

[3] http://homepages.cs.ncl.ac.uk/alexander.romanovsky/home.formal/caa.html

the interaction; and atomicity of external data to ensure that intermediate results are not passed to the outside processes before the interaction finishes. These properties make DMIs an excellent vehicle for implementing reliable applications.

In order to support such implementations, Zorzo and Stroud have proposed within DRIP a general scheme for designing DMIs in a distributed object-oriented system [Zorzo & Stroud]. They extend the notion of multiparty interaction to include facilities for handling exceptions which allows dealing with failures in one or more participants of the multiparty interaction, and in particular concurrent exceptions and synchronization upon exit. Roles are implemented as remote objects, each of which can be installed on a different computer allowing the system to tolerate computer crashes (provided that the exception handler is capable of coping with this type of failure, and the exception handling roles are in different computers).

CAAs can be derived from DMIs by adopting a more restricted form of exception handling with a stronger exception handling semantics. DRIP is designed to support this derivation and can then serve for implementing CAAs. Since CORRECT focuses on the CAA-based architectures for Java applications, it was decided to use DRIP within it.

DRIP is however a white-box framework [Johnson & Foote] since its reuse requires knowing its internal design and implementation as described in [Zorzo & Stroud]. It is not accompanied by a method designed for facilitating its reuse, and does provide no way for relating code to the earlier phases of the life cycle. DRIP provides then a basis for our target systems, but, it is necessary to associate it with a method.

The problem addressed by this research is then providing methodological support for developing more easily and cost-effectively fault-tolerant distributed applications based on the CAAs, by reusing the DRIP framework. The next section describes our solution.

5 The *DRIP Catalyst* MDE/MDA Method

In addition to DRIP, DRIP Catalyst comprises a process, a UML-Profile, and a set of transformations, integrated in a tool. This section describes briefly the different components of this method at its current state of development as well as the architecture of the tool

5.1 FTT-UML Profile

As notation for modeling CAAs, CORRECT reuses currently the FTT-UML Profile: a UML-Profile for Fault-Tolerant Transactions defined in [Guelfi, Le Cousin, Ries] and illustrated in Figure 1 below. We have developed FTT-UML as a language for modeling dependable complex business processes using UML. FTT-UML eases understanding of processes by business analysts, their validation and verification. It allows also reusability of processes.

FTT-UML is designed to be directly integrated with COALA [Vachon & Guelfi], a syntactically and semantically well-defined fault-tolerant advanced transaction model based on CAAs. In other words, the COALA language provides a well-defined semantics of CAAs[4]. CAAs are then the central piece of this notation, making FTT-UML a good starting point for CORRECT.

[4] FTT-UML aims at providing to COALA a UML-based syntax adapted to business process modeling, and suitable for verification and validation.

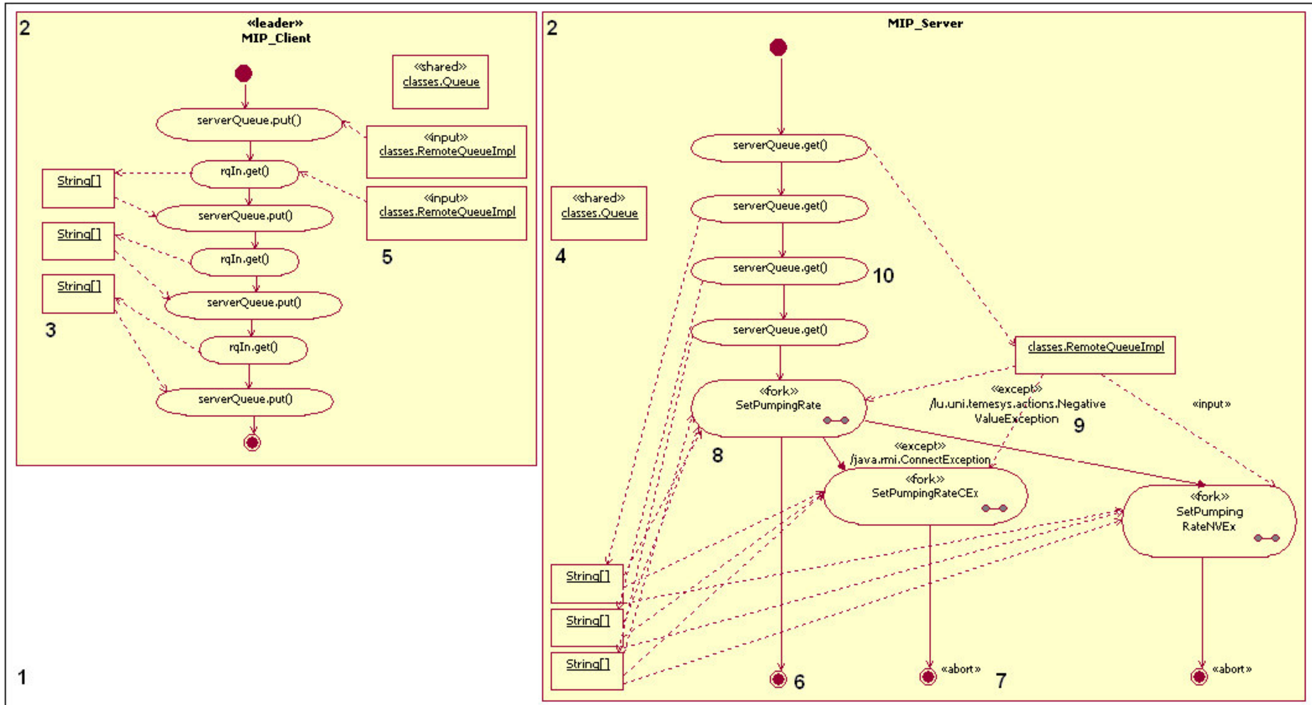

Figure 1: FTT-UML profile for modelling CAAs

Figure 1 above illustrates the different modelling elements offered by FTT-UML as follows:

1. A whole Activity Diagram represents one CAA.
2. Each Role is represented by a named swimlane. This swimlane can have the “leader” stereotype to express the fact that this leader will be in charge of declaring and instantiating all nested CAAs that appear inside the roles of the action.
3. A local object is represented by a named UML object that shows its type. It doesn’t have any stereotype.
4. A shared object is represented like a local object, with the “shared” stereotype.
5. An input (resp output) object is represented like a local object, with the “input” (resp “output”) stereotype.
6. A starting point represents the synchronization point for each role, at the beginning of a role. A final state represents the synchronizing point for each role, at the end of an action.
7. If a special outcome is needed, a stereotype is provided:
   a. “abort” in case of an abort outcome;
   b. “except” in case of an exceptional outcome. In this case, the name of the exception is also provided.
8. Nested CA Actions are represented by sub activities. Incoming links are representing:
   a. Inputs if they are linked to objects;
   b. Execution path. The role taken in the sub action is:
      i. The one provided by the link if available;
      ii. The one with the same name as the containing action if none is provided;
      iii. All roles at the same time if the sub action has the “fork” stereotype. In this case, it instantiates the sub actions and calls for its whole execution involving all its roles.
9. Each possible outcome of a sub action must provide an execution path:

a. A normal outcome is represented by a simple link going to the next activity or final state; (there can be only one normal outcome per action)
b. An abort outcome link has the "abort" stereotype and MUST link to a compensating sub action that is immediately followed by a final state; (there can be only one abort outcome per action)
c. An exceptional outcome link has the "except" stereotype, provides the name of the caught exception and MUST link to a compensating sub action that is immediately followed by a final state. (Design must provide one exception handler per possible exceptional outcome)

10. Activity nodes represent Method calls on objects. The name of this node refers to the following topology: "objectName.method()" where objectName is an accessible object (from the role point of view) and method is a visible method of this object. Arguments of this call are represented by the name of the objects inside the parenthesis, or by linking objects to the activity node.

This Profile has been used in several prototypes. It allows effectively to model CAAs, but it has an important disadvantage, that of conducting to models that are too close the Java syntax. This is why we currently use them for specifying platform-dependent design models. We are investigating a notation for platform-independent design models. The idea is that developers should use that notation, and FTT-UML should be mostly used to generate by transformation an intermediary compatible model, before generating code for different platforms.

5.2 Process

We have designed the process associated with DRIP Catalyst as a sequence of eight phases. These phases are organized around the specialization of the object-oriented framework via model transformations. Each phase takes as input the output of the previous phase. The input to the first phase is an unambiguous requirements specification. The study of requirements specification and their integration in this process is out of the scope of CORRECT. In particular we don't show how to manage tractability from requirements to design, implementation and tests. One of the options would however be to envisage a UML analysis model, i.e., a PIM that describes the application to be developed in business terms.

Here is a short description of the phases of this process:

1. **Problem-to-Solution Transition**: Transition from the requirements to the solution space by sketching the set of nested CAAs that satisfy the requirements. Each CAA is defined at this stage by its name, a short description, preconditions and postconditions. Output: Sketch of the solution in terms of CAAs.
2. **Platform-Independent Architectural Design**: The CAAs identified during the previous phase are grouped into logically coherent packages. One UML Class Diagram containing one UML package is created for each CAA package. Nesting of CAAs is allowed in the model by creating nested packages. Each leaf package represents a CAA and is linked to the model element that provides its platform-independent detailed design. For the moment this element is an Activity Diagram. Output: A set of UML Class Diagrams.
3. **Platform-Independent Detailed Design**: The modeling elements linked to the CAAs during the previous phase are fully specified using a UML-based notation. This notation is currently under development. In the meanwhile the FTT-UML is used. Output: A set of Stereotyped UML Activity Diagrams. Figure 1 above is itself an illustration of how FTT-UML allows modelling CAAs. For space reasons we don't detail the logic behind this design.
4. **Formal verification**: A fault-tolerant application should satisfy a set of dependability properties. For instance, we shall know that all exceptions are handled; if an exception is handled the external system behavior is equivalent to the system behavior without exceptions; and correct action nesting is in place. Using this phase a formal method that is under

development and takes advantage of the UML models, should be used to automatically check such properties. One of the possibilities that we investigate is an extension of COALA to adapt it to service-oriented architectures targeted by CORRECT. Output: Verified platform-independent detailed design model (PI2DM).

5. **PIM-to-PSM Transition**: Transition from PI2DM to a platform-specific detailed design model (PS2DM) by model transformation. For the moment the set of transformations to apply is empty since the PI2DM is strictly equivalent to the PS2DM. Output: A set of Stereotyped UML Activity Diagrams.
6. **PSM-to-code Transition**: Transition from PS2DM to the Java code by model transformation. For the moment we generate only Java code that specializes DRIP. But we believe that the same model should allow generating code for other comparable frameworks. The ability to choose between several platforms depends before all to the existence of a conforming framework. Output: Code that specializes the embedded framework; a prototype implementation of the application.
7. **Code completion**: Filling manually some parts of the generated code. Even if transformations can produce most part of the implementation model, some additions have to be made "by hand". We plan assisting the developer by indicating the parts of the code that need to be completed, in the case it happens that it is not possible or uninteresting to fully generate the code from models. Output: Code that can be compiled.
8. **Deployment**: Deployment of the application requires a set of configurations that are realized during this phase. Deployment phase can be facilitated be creating a wizard that generates the configuration files via transformations. Output: deployment guide and configuration files.

5.3 Model Transformation

Model transformation is currently only used at the sixth phase, PSM-to-code Transition, and more particularly for generating the Java code that specializes DRIP. The binding between PS2DM and DRIP is then realized by a mixture of Visitor and template-based techniques. The templates correspond to those parts of the DRIP extension points that are static. Visitor serves to fill those templates thanks to the information captured in the UML model.

The same set of transformation is successfully used for generating code for two case studies. The Home Care case described in the introduction, and the Travel Agency case [Zorzo *et al.*]. The full validation of our approach to model transformation requires however more experimentations. We are also studying suing a more specialized tool like Stratego [Stratego] or Generative Model Transformer [GMT] for better expressiveness and maintainability of transformations.

5.4 CORRECT MDA Tool

The methodological components described in the previous subsections are integrated together thanks to the UML CASE tool IBM/Rational XDE (JAVA version) [XDE]. We call the resulting software the CORRECT MDA Tool, a Domain-Specific MDA tool dedicated to distributed, CAA-based fault-tolerant applications.

The role of CORRECT MDA Tool is to support developers during all different phases of the above process. However, it currently provides support only for developing CAA models thanks to FTT-UML, and generating code for DRIP. Future releases of this tool should allow developing CAA models using a higher-level notation, and also to generate code for different object-oriented frameworks that embody CAAs.

As it is illustrated by Figure 2 below, this tool supports currently the model-to-code transformation following this approach:

1. Export the model to an XMI file
2. Convert XMI to an easier to use XML file (XSLT transformation)
3. Transform the XML file into a UML model "in-memory" representation, thanks to DOM
4. Transforming the UML model into an easier-to-use CAA Model
5. Travers the CAAModel by a Visitor and fill the DRIP code templates

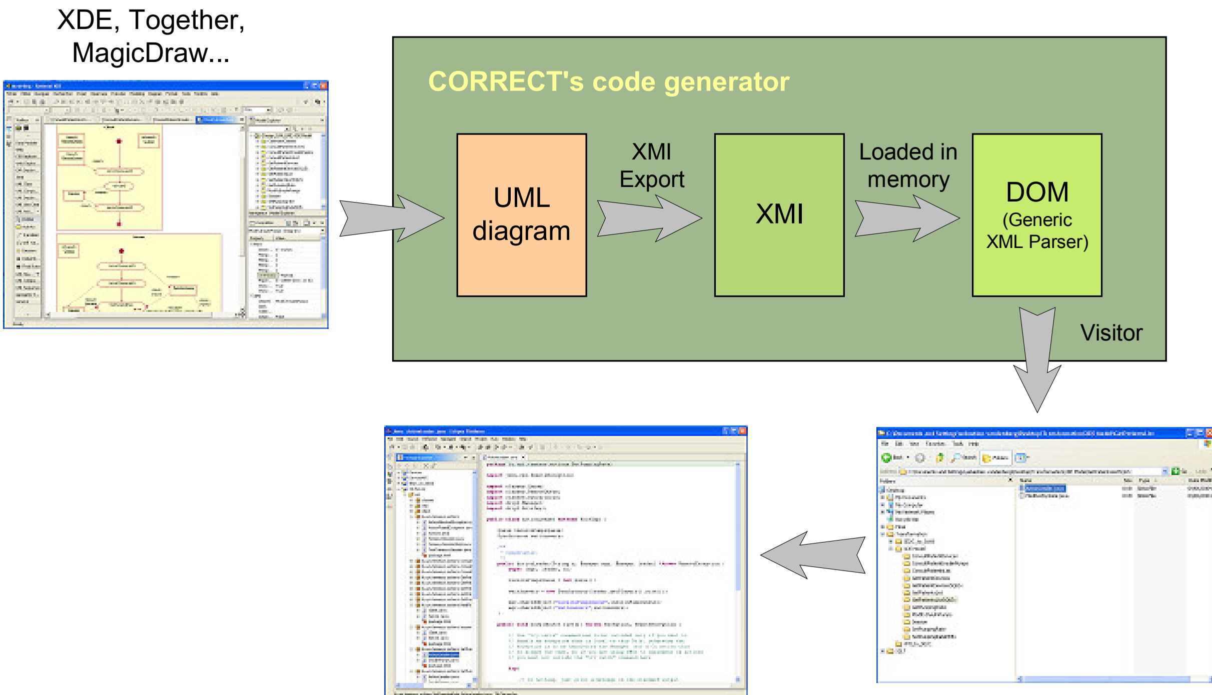

Figure 2: Model Transformation in the CORRECT MDA Tool

Programmers can in addition use in a compatible way Eclipse and its plug-ins, notably those for Web services like Lomboz.

5.5 Illustration

Figure 3 below illustrates the detailed design of the role Client in the Session CAA. It corresponds to the starting point of the system (entry point). It represents a session (the Client role) that is created and started every time a client connects to the system. This special action encloses all activities that the client executes, even if these activities are executed on the client side (JSP). This session action terminates when the client crashes or logs-off. By nesting CA actions and only CA Actions, we are certain that all the activity's chain is fault tolerant. No piece of code is executed out of a fault tolerant action.

On this figure, we can see how the case study works. It waits for a command (ActionValue). It then gives this value to the Server role through a Queue. It then makes a switch statement to decide what sub-action it has to call. After having executed this sub-action, it comes back to a wait position to close the cycle. This session has no final state since it always runs. We could stop it with a recognized interface exception if we would. The full detailed design of the Home Care and Travel Agency cases is given a separate document under progress.

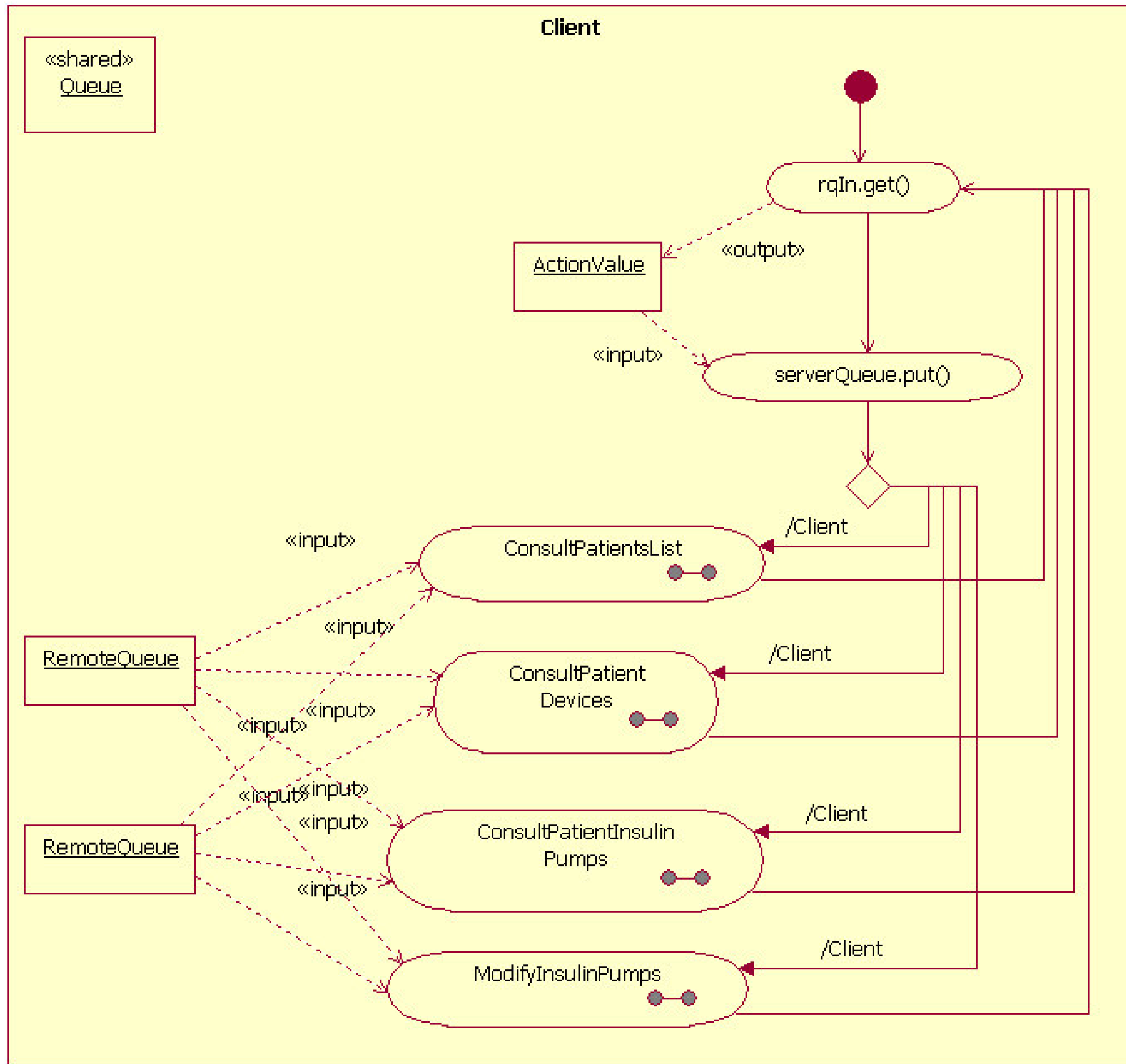

Figure 3: An example of a detailed CAA deign – The Session CA Action (Client Role)

We would like to insist on the fact that the UML models like that of Figure 3 above or Figure 5 below specify the specializations of the DRIP framework. For example, in Figure 6 below we can see the code for the Java class GPDSQLEx_MedRecSystem. This class is generated thanks to the UML diagram illustrated in Figure 7, and more precisely the role called GPDSQLEx_MedRecSystem (at the right of the figure). The generated code specializes the class RoleImpl of the DRIP framework. The overridden body method corresponds to one of the extension points of the DRIP.

The notation is still too close to the implementation language; this is why we work on a higher-level notation. But there is already a significant improvement in the way programmers reuse DRIP. The sequence of modeling and code generation below wants to illustrate for example how easier it is for the programmers to test new ideas and modify the application by changing the UML model of the application instead of working at code level. In a near future they will be able to verify also the fault-tolerance properties of their application, before generating the code. We will of course work to ensure the consistence between the verified model and the generated code.

Figure 4 below shows the essential part of the code of the Java class generated for the role called GPDSQLEx_MedRecSystem and illustrated in Figure 5. At this stage this role do simply nothing.

```
ConTEXT - [R:\Transformation\Temesys\GetPatientDevicesSQLEx\GPDSQLEx_MedRecSystem.java]
GPDSQLEx_MedRecSystem.java
15      {
16          super(mgr, leader, name);
17      }
18
19
20      public void body(java.util.Hashtable roleParameters)
21          throws java.lang.Exception, java.rmi.RemoteException
22      {
23          try {
24          } catch (Exception ex) {
25              System.err.println("Exception caught in: Temesys.GetPatientDevicesSQLEx.GPDSQLEx_MedRecSystem");
26              ex.printStackTrace();
27              throw ex;
28          }
29      }
30
31
32  }
33
Ln 1, Col 1    Insert    Sel: Normal    DOS    File size: 775
```

Figure 4: Java code generated for the GPDSQLEx_MedRecSystem role – step 1

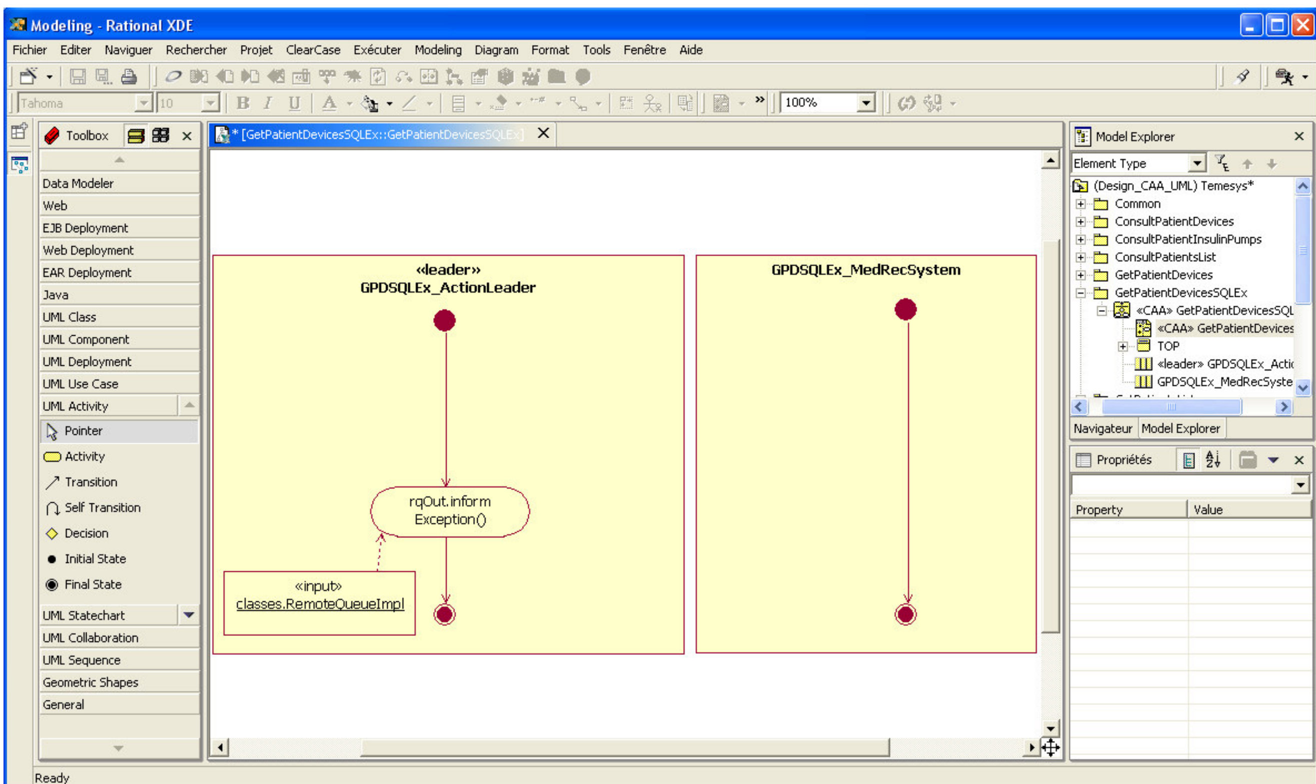

Figure 5: FTT-UML model for the GPDSQLEx_MedRecSystem role – step 1

Figure 6 below shows the code of the Java class generated for the same Role, but slightly modified, as illustrated in Figure 7.

ConTEXT - [R:\Transformation\Temesys\GetPatientDevicesSQLEx\GPDSQLEx_MedRecSystem.java]

```java
public class GPDSQLEx_MedRecSystem
    extends drip3.RoleImpl
{
    // Public attributes

    // Protected attributes

    // Private attributes
    private classes.Synchronous wait;

    public GPDSQLEx_MedRecSystem(String name, drip3.Manager mgr, drip3.Manager leader)
        throws java.rmi.RemoteException
    {
        super(mgr, leader, name);
    }


    public void body(java.util.Hashtable roleParameters)
        throws java.lang.Exception, java.rmi.RemoteException
    {
        try {
            // Shared object: wait
            wait = (classes.Synchronous)mgr.getSharedObject("wait");
            wait.synchronize();
        } catch (Exception ex) {
            System.err.println("Exception caught in: Temesys.GetPatientDevicesSQLEx.GPDSQLEx_MedRecSystem");
            ex.printStackTrace();
            throw ex;
        }
    }
```

Figure 6: Java code generated for the GPDSQLEx_MedRecSystem role – step 2

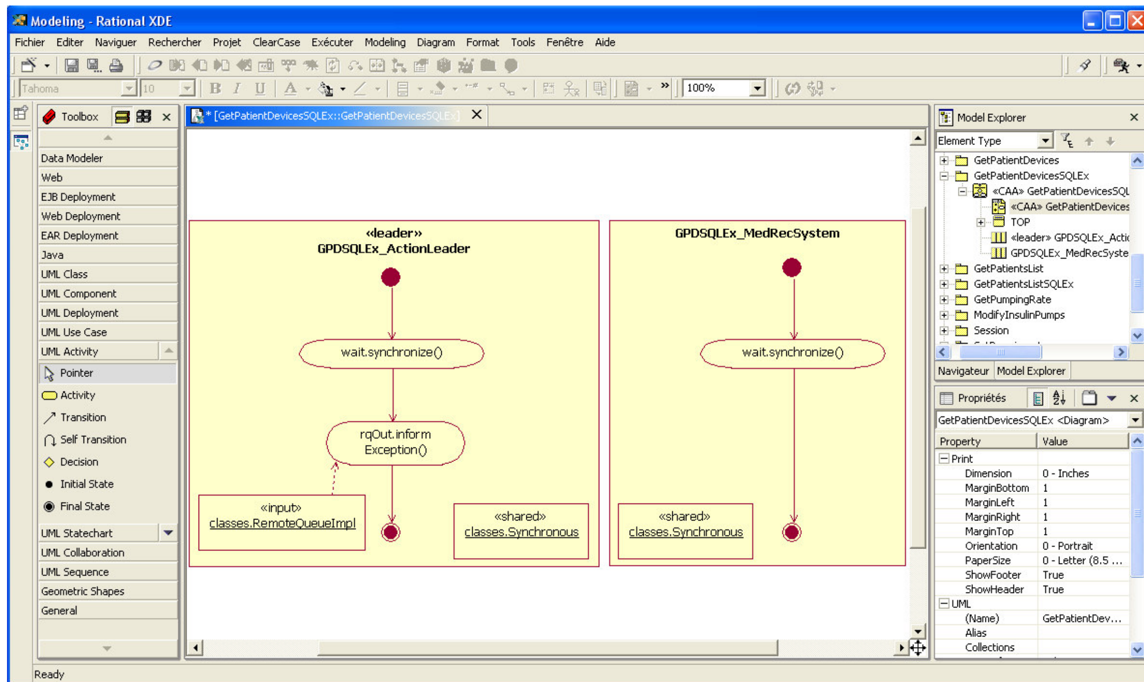

Figure 7: FTT-UML model for the GPDSQLEx_MedRecSystem role – step 2

The command line shell illustrated in Figure 8 is used for launching the transformation. The format for the command is “transform.bat path/to/your_xmi_file.xml”.

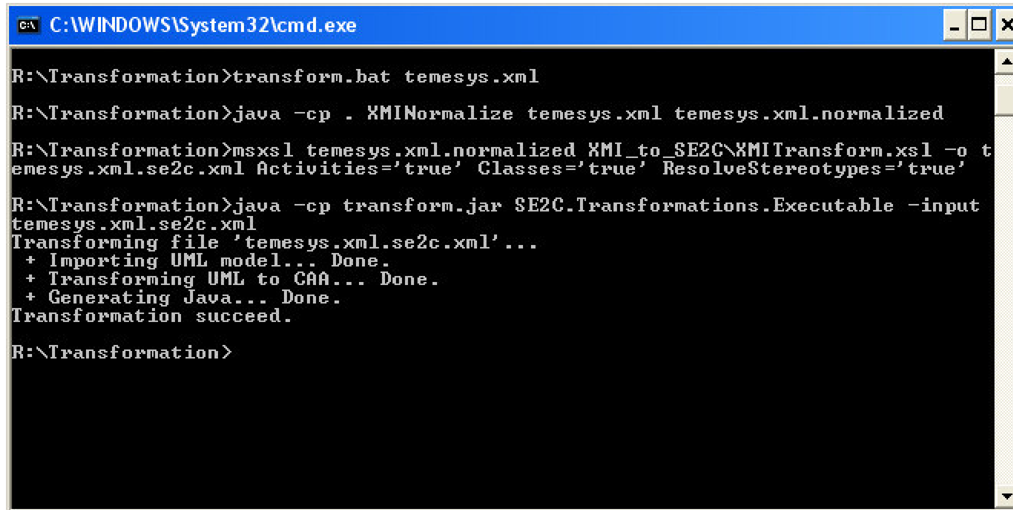

Figure 8: Example of command line shell launching transformations

6 Conclusion and perspectives

This paper presents an ongoing work on developing a new architecture-centric and model-driven method for stepwise development of fault-tolerant applications based on Coordinated Atomic Actions and the DRIP framework. A tool has been developed that embodies a UML Profile and a set of transformations. It allows generating code by model transformation that specializes the DRIP framework. An MDE chain composed of eight phases is also defined and is partially supported by the tool.

Architecture-centric MDE/MDA methods like that presented here offer an attractive solution for identifying the architecture of the PSM. The PSM is indeed generated to extend a well-defined architecture, according to well-defined rules. Frameworks offer a solid architectural foundation for code generation. Further, from the MDA perspective, using frameworks does not seem an obstacle to support solutions running on multiple platform, since in case this is really an issue, the same framework can be implemented in other languages.

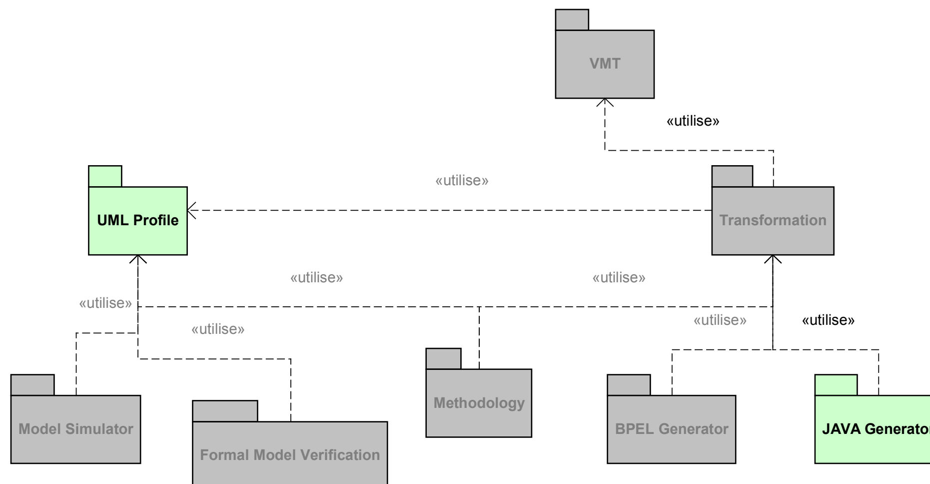

Figure 9: The block architecture of the ideal CORRECT MDA Tool

As it is illustrated in Figure 9, the CORRECT MDA Tool should ideally support not only several notations for platform independent and specific models, but also a model simulator, an engine for formal model verification, a process, and model transformation facilities for generating code on different platforms. We are currently investigating practical solutions for model verification and developing a notation for specifying platform-independent detailed design models.

We are also currently experimenting with our tool while implementing a more complex and extended case study. In particular, we are interested in the relationship between the set of transformations and the target family of the applications. Further, UML-FTT is currently defined as an extension to UML 1.5. It is being ported to UML 2.0, while improving its expressivity, in particular in the case of Web-services. This release of the profile will also be documented in a more standard way.

During the workshop we would particularly like to participate in discussions on methods for developing MDSD methods.

7 Acknowledgments

This work has benefited from a funding by the Luxembourg Ministry of Higher Education and Research under the project n°MEN/IST/04/04. A. Romanovsky is supported by FP6 IST RODIN project (n°511599). We would also like to thank P. Periorellis, A. Zorzo, A. Capozucca, and A. Berlizev for their collaboration on this project.

8 References

[Ahola] J. Ahola, Ambient Intelligence, ERCIM News 47(Ambient Intelligence), 2001

[IST] IST Advisory Group, Ambient Intelligence: from vision to reality - For participation in society & business, http://www.cordis.lu/ist/istag-reports.htm, 2003

[Kazman] R. Kazman, "Software Architecture", Handbook of Software Engineering and Knowledge Engineering, World Scientific Publishing, pp. 47-68, 2001

[Johnson & Foote] R. Johnson, B. Foote, "Designing Reusable Classes", Journal of Object-Oriented Programming 1(2), pp.22-35, 1988

[Clements & Northrop] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison Wesley, Reading, MA, USA, 2001

[Lee & Anderson] P. A. Lee, T. Anderson. Fault Tolerance Principles and Practice, volume 3 of Dependable Computing and Fault-Tolerant Systems., Springer - Verlag, 2nd edition, 1990.

[Campbell & Randell] R. H. Campbell, B. Randell. Error recovery in asynchronous systems. IEEE Transactions on Software Engineering, SE-12(8), 1986.

[Cristian] F. Cristian. Dependability of Resilient Computers, chapter Exception Handling. Blackwell Scientific Publications, 1989.

[Gray & Reuter] J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993

[Xu et al.] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu. Fault Tolerance in Concurrent Object-oriented Software through Co-ordinated Error Recovery. In FTCS-25, California, USA. IEEE CS Press, 499-509, 1995.

[Guelfi, Le Cousin, Ries] N. Guelfi, G. Le Cousin, B. Ries, Engineering of Dependable Complex Business Processes using UML and Coordinated Atomic Actions. Accepted for the International Workshop on Modelling Inter-Organizational Systems (MIOS'04), Larnaca, Cyprus, 2004.

[Zorzo, R. Stroud] A.F. Zorzo, R.J. Stroud. An Object-Oriented Framework for Dependable Multiparty Interactions. In OOPSLA-99. ACM Sigplan Notices, 34(10), 435-446, 1999.

[Vachon & Guelfi] J. Vachon, N. Guelfi. COALA: a Design Language for Reliable Distributed System Engineering. In Proceedings of International Workshop on Software Engineering and Petri Nets (SEPN'00), Aarhus, 2000.

[XDE] IBM Rational Software, "IBM Rational XDE - Extend your development experience", http://www-306.ibm.com/software/awdtools/

[Johnson 1992] R. E. Johnson. Documenting frameworks using patterns. ACM SIGPLAN Notices, 27(10):63–76, Oct. 1992. OOPSLA'92 Proceedings, Andreas Paepcke (editor). URL: http://st.cs.uiuc.edu/pub/papers/HotDraw/documenting-frameworks.ps.

[Beck & Johnson] K. Beck, R. Johnson. Patterns Generate Architectures. In the European Conference on Object-Oriented Programming 1994, pages 139-149. Springer-Verlag Lecture Notes in Computer Science # 821.

[Gamma & al.] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns---Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[Beder *et al.*] D.M. Beder, B. Randell, A. Romanovsky, C.M.F. Rubira. "On Applying Coordinated Atomic Actions and Dependable Software Architectures for Developing Complex Systems" in the 4th IEEE International Symposium on Object-oriented Real-time Distributed Computing, Margeburg, Germany, May, 2001, pp. 103-112.

[Tartanoglu *et al.*] F. Tartanoglu, V. Issarny, A. Romanovsky, N. Levy. Dependability in the Web Service Architecture. In: Architecting Dependable Systems, de Lemos, R., Gacek, C., and Romanovsky, A., (eds.) pp. 90-109.

[Romanovsky *et al.*] A. Romanovsky, P. Periorellis, A. Zorzo. Structuring Integrated Web Applications for Fault Tolerance, A.F. In Proceedings of the 6th International Symposium on Autonomous Decentralised Systems (ISADS 2003), Pisa, Italy, April 2003 pp. 99-106. IEEE CS Press 2003.

[SRDS] F. Tartanoglu, V. Issarny. A. Romanovsky. N. Levy. Coordinated Forward Error Recovery for Composite Web Services. In the 22nd Symposium on Reliable Distributed Systems (SRDS). Florence, Italy. 2003. pp.167-176.

[Lemos & Romanovsky] R. de Lemos, A. Romanovsky. Exception Handling in the Software Lifecycle. Int. J. Computer Systems Science and Engineering, 16, 2, pp.167-181, 2001.

[QoS] http://www.omg.org/technology/documents/modeling_spec_catalog.htm

[Guelfi, Perrouin] N. Guelfi, G. Perrouin, Using Model Transformation and Architectural Frameworks to Support the Software Development Process: the FIDJI Approach, 2004 Midwest Software Engineering Conference, Chicago, IL, USA, DePaul University, 2004.

[Guelfi, Pruski, Ries] N. Guelfi, C. Pruski, B. Ries, "A Study of Mobile Internet Technologies for Secure e-commerce Applications Development", Techniques and Applications for Mobile Commerce (TAMOCO) part of Multi-Konferenz Wirtschaftsinformatik 2004, Essen, Germany, p.194-2004

[Zorzo & al.] A. Zorzo, A.F., Periorellis, P. and Romanovsky. Using Coordinated Atomic Actions for Building Complex Web Applications: a Learning Experience, A. In Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-time Dependable Systems (WORDS 2003), Guadalajara, Mexico, 15-17 January 2003 pp. 288-295. IEEE CS Press 2003.

[Johnson & Foote] R. Johnson, B. Foote, "Designing Reusable Classes". Journal of Object-Oriented Programming, 1988.

[Stratego] http://www.stratego-language.org/Stratego/TransformationToolComposition

[GMT] Generative Model Transformer. http://www.eclipse.org/gmt/