Designing fault-tolerant component based applications with a model driven approach

Brahim Hamid, Ansgar Radermacher, Agnes Lanusse, Christophe Jouvray,
Sébastien Gérard and François Terrier

CEA, LIST
Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués
Boite 65, Gif sur Yvette, F-91191 France
email:{brahim.hamid,ansgar.radermacher,agnes.lanusse,christophe.jouvray,
sebastien.gerard,francois.terrier}@cea.fr

Abstract. The requirement for higher reliability and availability of systems is continuously increasing even in domains not traditionally strongly involved in such issues. Solutions are expected to be efficient, flexible, reusable on rapidly evolving hardware and of course at low cost. Model driven approaches can be very helpful for this purpose. In this paper, we propose a study associating model-driven technology and component-based development. This work is illustrated by the realization of a use case from aerospace industry that has fault-tolerance requirements: a launch vehicle.
UML based modeling is used to capture application structure and related non-functional requirements thanks to the profiles CCM (CORBA Component Model) and QoS&FT (Quality of Service and Fault Tolerance). The application model is enriched with infrastructure component dedicated to fault-tolerance. From this model we generate CCM descriptor files which in turns are used to build boot-code (static deployment) which instantiates, configures and connects components. Within this process, component replication and FT properties are declaratively specified at model level and are transparent for the component implementation.

Key words: Connector, CORBA Component Model, Distributed applications, Model-driven approach, Profile QoS+FT, Replication.

1 Introduction

A distributed system is a system which involves several computers, processors or processes which cooperate in some way to do some task. However, such systems require a specific treatment of faults. Faults may be hardware defects (link failures, crashes) or software faults which prevent a system to continue functioning in a correct manner.

In such systems, solutions are expected to be efficient, flexible, reusable on rapidly evolving hardware and of course at low cost. Model-driven engineering [19] provides a very useful contribution for the design of fault-tolerant systems, since it bridges the gap between design issues and implementation preoccupation. It helps the designer to concentrate on application structure and required behavior and permits to specify in a separate way non-functional requirements such as Quality of Service and/or fault-tolerance issues that are very important to guide the implementation process. The model(s) can be analyzed at a very early stage in order to detect potential misconceptions; and then, exploited by specific tools through several steps of model transformation and/or interleaving with platform models in order to produce the application components and configuration files.

In this paper, we propose a study associating model-driven approach and component-based development to design distributed applications that has fault-tolerance requirements. We focus on the run-time support offered by the component framework, notably the replication-aware interaction mechanism and additional system components for fault-detection and reference management. To illustrate the power of our approach we examine a test case from aerospace industry that has fault-tolerance requirements: a launch vehicle.

UML based modeling is used to capture application structure and related non-functional requirements thanks to two specialized extensions CCM (CORBA Component Model) [13] and QoS&FT (Quality of Service and Fault Tolerance) OMG profiles [15]. From this model we generate descriptor files (according to Deployment and Configuration standard (DnC) [14]). These descriptors are in turn used to configure a devoted infrastructure consisting of a container/component based architecture and to load configured components. Within this process, component replication and FT properties are declaratively specified at model level and are transparent for the component implementation.

The work is conducted in the context of a national project called *"Usine Logicielle"* [1]. This project is three-folded : modeling, validation and infrastructure/middleware support along with configuration support.

The rest of the paper is organized as follows. In the next section we present the model including the distributed computing systems, component model and the connector extension. In Section 3, we present briefly the proposed framework to implement fault-tolerance mechanisms. Section 4 describes the proposed methodology to design fault-tolerant distributed applications for component systems. We outline the profiles used on model level and describe the code generation and platform configuration process. In section 5 we review some related works. The last section summarizes and gives an outlook of future work.

2 Background

In this section, we outline two different aspects: the assumptions about the underlying computing system (mainly its network) and the component platform, namely the CORBA Component Model extended with the connector paradigm.

2.1 Distributed computing system model

A distributed system is a set of processes (or processors) and communication links. Processes communicate and synchronize by sending and receiving messages through

[1] This work has been performed in the context of the Usine Logicielle project of the System@tic Paris Région Cluster (http://www.usine-logicielle.org).

the links. The network topology is unspecified and each node communicates only with its neighbors. Two processes are considered as neighbors if and only if there is a link (channel) between them. We deal exclusively with connected topologies. A process can fail by crashing, i.e. by permanently halting. A process can also produce wrong computation results (e.g. due to spontaneous bit failures). Communication links are assumed to be reliable. The system is improved by failure detector modules. After a node fails, a dedicated protocol involving these modules notifies all neighbors of this node about the failure.

Networks are asynchronous in the sense that processes operate at arbitrary rates and messages transfer delay are unbounded, unpredictable but finite. We assume that message order is preserved. To implement failure detection, the dedicated protocol use a weak form of synchrony such as [1, 6].

2.2 Fault-Tolerance mechanisms

Fault-tolerance can be achieved by multiple mechanisms, for instance parity checking on memory on a hardware level. In the scope of this paper, furthermore to use fault detection functionality, we consider replication management. Obviously, replication relates to hardware as well as to software. With respect to hardware, it means that processing resources (nodes) and network links are replicated. With respect to software it denotes that the same component instance is deployed on multiple nodes. There are different well known variants of how redundant components may work, they fall in three main categories: all replicas can execute the same request and results are voted ("hot" or active with vote), only a single replica is active ("cold") or mixed policies where replicas are active but only one, the master sends its result. Indeed, the actual redundancy policy chosen for an application results from a compromise between powerful redundancy mechanisms offering better reliability at a high cost in terms of price, communication, size and weaker mechanisms in terms of recovery time but at lower costs. These considerations are particularly important in the domain of embedded systems and have driven our will to promote flexible design and implementation of such mechanisms.

In this experiment, the faults handled relate to hardware fault (node not responding) detected by the Fault Detector component through liveliness control as described below, and software error (no answer or wrong result from a replica detected by the voting mechanism). If a software error is detected on the result coming from a replica, the node on which this replica resides is desactivated and considered as faulty.

2.3 Connector extension of the CORBA Component Model(CCM)

Our work is based on the CORBA Component Model (CCM) extended with the connector paradigm. A main advantage of this model is its separation of business code located in the component from the non-functional or service code located within a container.

The CCM standard supports three different communication paradigms (port types): synchronous method calls based on CORBA (provided/required interface), event publishing and reception and the recently added streaming. One drawback is that the implementation of such communication mechanisms is generally fixed, i.e. a CCM implementation provides a single realization of the interactions between port types. This is quite restrictive, in particular for embedded systems requiring:

1. *Flexible interaction implementations*
2. *Additional communication models or variations of existing communication models*

There is no way to model this in a suitable way within the standard CCM model.

The limitations of this standard have driven us to propose an extension named the *eC3M* which introduces the concept of *Connector* in the context of *Component/Container* paradigm. This permits the definition of specific interaction semantics and to associate multiple implementations of a particular one when defining the deployment configuration. The connector extension to CCM has first been published in [18]. Here, we'll have a short look at it with a focus on specific connectors supporting the interaction with replicated components.

A connector has certain similarities with a component. It has a type definition consisting of ports providing or requiring interfaces and an implementation chosen at deployment time. The main difference is (1) its genericity – its interfaces are adapted to the component using it and (2) it is a *fragmented* entity: since the connection between a component and its connector is always a direct local call, each port of the connector is co-located with the component it is connected with.

3 Our Infrastructure

We propose a simple infrastructure based on a set of non functional components. It has similar elements as in FT-CORBA [11], but since these are realized as CCM components they are independent of an ORB, in particular the connector extensions allows for choosing different interaction implementations. The separation between components and containers in CCM allows to keep fault-tolerance aspects out of the business code. Only the container and the associated connector fragments (which can be seen as part of the container) manage FT aspects.

3.1 Fault-tolerance Framework

Here we show the set of non-functional (control) components used to support fault-tolerance and the run-time support, notably the replication-aware interaction mechanisms. To handle faults, we use the following control components:

1. *A fault detector (FD):* Each node is equipped with a fault detector to detect other faulty nodes. These components communicate with each other to build the list of faulty nodes. This component implements a fault detection protocol such as heartbeat or interrogation. In our framework, we use the following: at periodic rate, each fault detector (source node) performs broadcasting of aliveness requests to all other nodes (destination nodes). A requested destination node answers (or not) the source node. Thus, each fault detector node maintains the list of nodes and their states (alive, not alive).

2. *A fault tolerance manager (FTM):* The fault tolerance manager component performs reconfiguration to deal with detected faults [7]. It keeps tracks of ongoing status of replicas and defines fault processing. Reconfiguration is defined as the operation of transition from a source mode to a target mode when an event (faulty node) occurs. This is to keep the number of valid replicas, i.e after each failure occurrence, it checks that the number of valid replicas is higher than the minimum number of replicas. That is, the FTM changes the configuration of the system to satisfy the dependability requirements specified by the designer of the application at the design level.
3. *A replica manager (RM):* The role of this component is to store references of all replicated components (replicas) on a certain node. This component is not replicated, but deployed on each node. It handles a list of references to replicated components deployed in this node. It enables the creation /deletion replicas and their deployment in the case of dynamic reconfiguration.

Instances of these control components are activated on each node. However, the fault tolerance manager instance is in a leader mode on only one node, which may change dynamically when a faulty node event occurs.

3.2 Replication at a Connector Level

In the context of fault tolerance, a connection with a replicated component should perform group communication, i.e. the transparent communication with a set of replicas. Whereas this could be done with standard CCM and a specific CORBA implementation supporting group communication, it would be impossible to configure and control it (in case for instance of node failures) from standard CCM. As shown in the Section 2.3, the communication system is abstracted at a connector level. Since it is responsible for incoming and outgoing messages, it is an ideal place for the integration of replication protocol. Therefore, the user code interacts transparently with a group of replicas. Along with a replicated instance, the fragments of a connector are replicated as well.

Currently, we implemented an active replication ("hot") with vote mechanism as a proof of concept. In this variant, all replicas of a component instance are active at a given time and synchronize entries (optional) and results by a vote. We can separate the realization of a connector supporting this replication style into two phases. In the first, a unique request has to be distilled and sent to all replicas. In the second, the message is received by all replicas of the destination component and these (optionally) have to check that all got the same message.

Replicated components have a voter object in their container and a reference to this object is automatically passed to the connector fragment. The voter object is part of the run-time required for fault-tolerance. The code validates (*acknowledgeRequests*) the parameters with the other replicas by means of the voter object, before it sends a message to all replicas to the target object. The call of method *acknowledgeRequest* blocks until the result has been confirmed. If the current replica is leader, it sends the request to all replicas of the *server* fragment thanks to the *replica manager* instance in that node. Moreover, the *fault detector* instance is invoked to avoid sending request to the crashed node.

4 Designing Fault-Tolerant Distributed Applications (MDE Approach)

As described above, a simple redundancy management system can be implemented thanks to specific middleware components devoted to generic mechanisms such as fault-detectors, voters and so on...and specialized services implemented into connectors.

Here we describe how a MDE approach can help developers design their application and take full benefits of this infra-structure to build flexible efficient fault-tolerant component based applications. We present the approach chosen and the tools developed to support it.

Our laboratory *LISE* [2] has developed a tool that supports UML modeling (Papyrus UML [3]) based on the Eclipse environment. This tool suite provides a graphic UML modeling editor and code generation components (Java, C, C++). The tool supports also advanced UML profile management. We have developed additional plug-ins which generate CCM descriptor files from a model containing component instances with fault-tolerance requirements.

Our methodology is illustrated by means of a test case from aerospace industry that has fault-tolerance requirements: a launch vehicle. For simplicity, many functions of this test case have been omitted. Two components are identified:

- Calculation component $(Calc)$: this component makes some computations and then invokes the $display$ method provided by the interface of the $(Display)$: component.
- Display component $(Display)$: it is responsible of displaying the result of the computations done by the $Calc$ components. It provides $display$ method through its interfaces to be used by the $Calc$ components.

The sample application is described as follows: a calculation component is periodically activated by a timer; the result of the calculation is passed to a display component. Here, component $Calc$ is replicated three times and we use an active with voting replication style. For this application, dependability requirement is that it must tolerate one node crash.

4.1 Application modeling

Application modeling when dealing with component based approaches consists of describing components, their required and offered services and then define component instances and finally how these instances are connected to form the final system.

The modeling basis is UML on which a variant of the profile for CORBA Component Model (eC3M) is applied. The application is described in terms of components and provided and required interfaces; profile properties permit to complete the description so that complete IDL can be generated from the description. Assembly characteristics

[2] Laboratory of Model Driven Engineering for Embedded Systems, which is part of the CEA LIST.

[3] http://papyrusuml.org

and deployment information are also provided through the eC3M profile with stereotypes close to DnC concepts. From this information deployment plans can be generated for regular applications.

To handle fault-tolerant requirements we apply a complementary profile named *FT profile* which is composed of a subset of QoS&FT [15] and uses NFP (Non Functional Properties) sub-profile of MARTE [16] (standard UML profile for Modeling and Analysis of Real-Time Embedded systems). Stereotypes dedicated to fault-tolerance specify the fault-detection policy, replication management style, replica group management style etc.. Fig.1 shows the structure of this profile. Black ended arrows denote concept extension (stereotype ***FTInitialReplicationStyle*** is an extension of UML Class). White ended arrows are standard UML generalization relations.

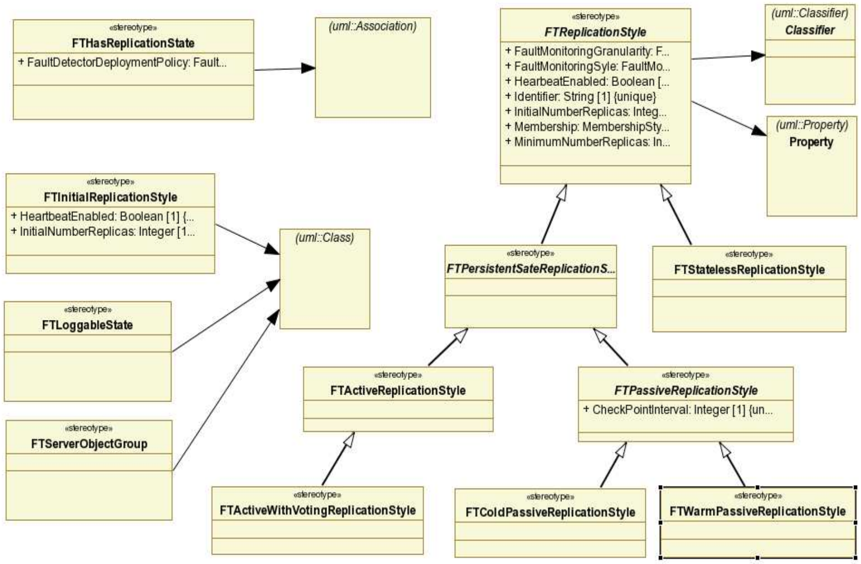
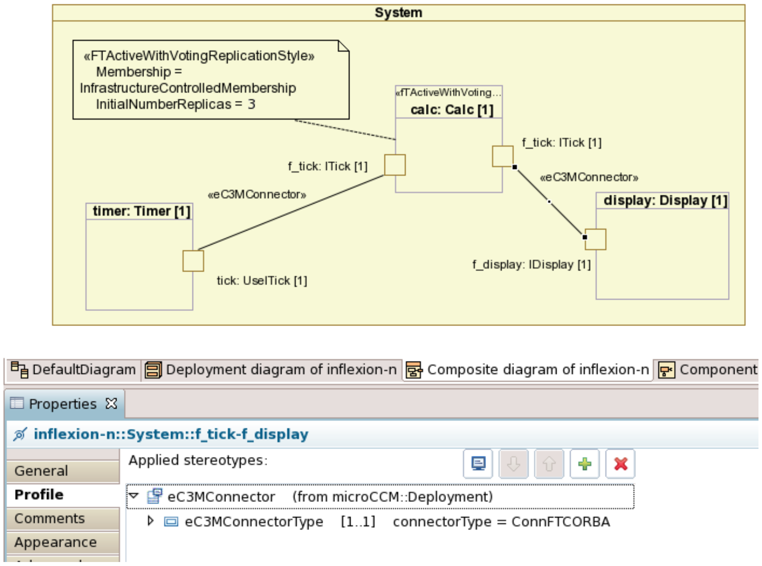

«stereotype»
FTHasReplicationState
+ FaultDetectorDeploymentPolicy: Fault...
(uml::Association)
«stereotype»
FTReplicationStyle
+ FaultMonitoringGranularity: F...
+ FaultMonitoringSyle: FaultMo...
+ HearbeatEnabled: Boolean [...
+ Identifier: String [1] {unique}
+ InitialNumberReplicas: Integ...
+ Membership: MembershipSty...
+ MinimumNumberReplicas: In...
(uml::Classifier)
Classifier
(uml::Property)
Property
«stereotype»
FTInitialReplicationStyle
+ HeartbeatEnabled: Boolean [1] {...
+ InitialNumberReplicas: Integer [1...
(uml::Class)
«stereotype»
FTPersistentSateReplicationS...
«stereotype»
FTStatelessReplicationStyle
«stereotype»
FTLoggableState
«stereotype»
FTActiveReplicationStyle
«stereotype»
FTPassiveReplicationStyle
+ CheckPointInterval: Integer [1] {un...
«stereotype»
FTServerObjectGroup
«stereotype»
FTActiveWithVotingReplicationStyle
«stereotype»
FTColdPassiveReplicationStyle
«stereotype»
FTWarmPassiveReplicationStyle

Fig. 1: The structure of the FT-Profile.

Once application components and interfaces have been defined, the system software architecture is described thanks to the UML composite diagram used to specify an assembly and hierarchical components. This diagram permits to determine what are the constitutive *parts* of the system and how they are inter-connected. Fig.2 shows the composite diagram corresponding to our sample application. The diagram indicates that the application consists of one component *calc*, one component *display* and one component *timer*. Connectors are defined between *timer* and *calc*, and *calc* and *display*. This is the description of the system without infrastructure components.



inflexion-n::System::f_tick-f_display

Applied stereotypes:

eC3MConnector (from microCCM::Deployment)

eC3MConnectorType [1..1] connectorType = ConnFTCORBA

Fig. 2: Applying fault-tolerance stereotypes.

Since we want to specify that redundancy is required we stereotype component *calc* with ***FTActiveWithVotingReplicationStyle*** stereotype and we indicate that membership policy is controlled by infrastructure and that initial number of replicas will be 3. In the same manner we indicate that connectorType of the connector between *calc* and *display* is ***ConnFTCORBA*** which means that a connector support for fault tolerance based on CORBA should be used (see next section).

From this model we can configure the final application, install binary files, generate appropriate connectors and configure specialized infrastructure services. This process follows several steps and uses different transformation tools described in the next section.

4.2 Code generation

Code generation is intended to support CCM implementation steps. This requires to generate : (1) CCM descriptor files from the model, (2) the code corresponding to a CCM implementation.

The first point concerns generation of component descriptors, a platform description and a *deployment plan*. A deployment plan contains information on the implementation and required artifacts (usually libraries), components instances, as well as allocation information (allocation of instances onto nodes), and connections between ports of these instances. The second one concerns the parsing of the deployment plan by a dedicated CCM implementation: *microCCM*. This framework is a tool set developed jointly with Thales which prepares application deployment from the analysis of the deployment plan. It produces a static deployment in which a *bootloader* file is generated for each node. This file contains code that instantiates components as well as connector fragments and performs the connections according to the deployment plan. Connector fragments are generated when necessary (this step is needed, since connectors adapt themselves to component interfaces, as shortly outlined in the previous section).

The following code (see Fig.3) gives a rough idea of the generated code contained within a connector fragment. In this case, fragment that is responsible for sending a result from the calculation component towards the display component.

```
void FTCORBA_IDisplay_client::display (CORBA::Float value)
{
    if (m_voter != NULL) {
        // calculate hash of request (used to simplify comparisons).
        Hash hash;
        hash.add (m_voter->getRequestNr ());
        hash.add (value);
        m_voter->acknowledgeRequest (hash.get ());
    }
    if (amILeader ()) {
        for (int i = 0; i<MAX_NR_OF_NODES; i++) {
            if (myRM->isOnNode ("DISPLAY",i) && !myFD->is_faulty_node(i))
              myRM->getObj ("DISPLAY",i)->display (value);
       }
    }
}
```

Fig. 3: Code of connector fragment associated with the node in which $Calc$ component is deployed.

4.3 Discussion

Overhead of connector fragments code : The following table provides an idea about the overhead of the connector fragments at some node. The figures are obtained for a prototype on a Linux PC. As said before, the bootloader file performs the instantiations and configuration of components and connector fragments and the connections between these. The connector fragments use a naming scheme that correspond to their name followed by the interface to which they adapt to and followed finally by the port name within the connector type. The overhead of a connector supporting fault-tolerance is relatively small, in general it depends on the number of operations and their parameters. The voter run-time adds about 11 Kb.

Efficiency, evolutivity, reusability : In order to use another replication style it suffices to (1) adapt our infrastructure to deal with such a replication style, i.e. provide a connector and (2) specify the use of this connector by means of a stereotype attribute of a connection on model level (as shown in Fig.2). A re-generation of the descriptor files and the connector generation will take this change automatically into account.

text	data	bss	dec	filename
13788	12	828	14628	gcc_linux_mico/obj/bootloader.o
372	4	1	377	gcc_linux_mico/obj/CCM_hooks.o
2936	4	1	2941	gcc_linux_mico/obj/CORBA_IFault_Detector_client.o
2339	4	1	2344	gcc_linux_mico/obj/CORBA_IFault_Detector_server.o
3245	4	1	3250	gcc_linux_mico/obj/FT_CORBA_IDisplay_client.o
1271	4	1	1276	runtime/FT/gcc_linux_mico/obj/ReferenceSet.o
11458	5	1	11464	runtime/FT/gcc_linux_mico/obj/Voter_impl.o

Fig. 4: Overhead of the connector fragments corresponding to the proposed implementation for a prototype on a Linux PC.

Thus, multiple deployment variants can be easily produced and tested (benchmarked) and optimized to find a suitable solution.

5 Related Work

Some CORBA implementations provided proprietary fault tolerance mechanisms such as OmniORB, Orbix and Orbacus. They are based on an embedded set of "contact details" within an interoperable object reference (IOR). These solutions are vendor specific and not interoperable. Therefore, the OMG standardizes fault-tolerant mechanisms (short FT-CORBA) [12] within the CORBA specification. The replication manager interface is the core of the FT-CORBA infrastructure, inheriting from three interfaces that deal with object groups, a generic factory and the fault-tolerance properties. The latter is also referred to by the FT-profile outlined in this paper. A full implementation of the FT-CORBA specification tends to be "big", therefore it is not implemented by many ORBs, in particular not by ORBs that are tailored for small and medium embedded systems.

AQuA (Adaptive Quality of service Availability, see[17] and [9]) is incompatible with FTCORBA. Fault-tolerance is obtained by active or passive replication and requires reliable group communication. It allows developers to specify the desired level of dependability, through the configuration of the system according to the availability of resources and the faults occurred. This system uses QoS contract as in Quality Objects [20]. The group communication service is based on Ensemble [8].

The AFT-CCM (Adaptive Fault-Tolerance) model [5] is based on CCM and treats fault-tolerance as a specific QoS requirement. For each component with fault-tolerance requirements, an AFT manager is created. This seems to be quite costly, but enables the modification of QoS parameters at run-time such as the replication coordinator implementing the replication technique (one component for each replica). A prototype of this system was built using OpenCCM (http://openccm.objectweb.org) running under ORBacus. Only passive replication style was implemented since active replication style requires group communication mechanisms that are not supported in the used ORB. Another approach for CORBA components replication is studied in [10]. This approach uses interceptor objects that accomplish replication management: each replicated component is associated with an interceptor object. In the AFT-CCM, a generic connector is used to avoid the implementation of a new interceptor object for each new component.

The MEAD (Middleware for Embedded Adaptive Dependability) group has proposed a fault-tolerant CCM in cooperation with Raytheon. This extension uses additional descriptor files containing deployment rules and container descriptions that specify the fault-tolerance properties of the application. The link between components and FT services (including fault monitoring, checkpoint (log) components) is done at the container level. There is a separation between logical and physical assembly in CCM process: for example, the number of replicas is logical and the placement is a physical concern. This deployment is achieved using an assembly manager/deployer that is installed at each host. Both active and passive replication styles are supported by the proposed extension using the extended virtual synchrony model [2]. This model guarantees that events are delivered in the same order at each node.

Different modeling approaches can be followed, several specialized description languages have been defined and are well adapted to describe system implementation (AADL and its error annex [3, 4]), EAST ADL which focuses particularly on the specification of allocation constraints, or some dedicated languages devoted to the development of critical systems based on formal techniques and synchronous calculus (as in the SCADE tool). But none of these approaches are well suited to the Container/Component paradigm.

The main difference between the fault-tolerant CCM approaches above and our approach is the focus on a specification based on UML and a standardized profile (QoS+FT). Another difference is that we integrated the fault-tolerance mechanism into a generic CCM extension. Note that our connectors replace interaction tailoring via interceptors that are used by other approaches to enable transparent replication.

6 Summary and Future work

We have shown that fault-tolerant applications can be generated directly from a specification of the architecture (component assembly & deployment) in UML and component descriptions as well as their implementation. The whole approach is largely based on standards: UML with CCM as well as a fault-tolerance profile and the execution middleware based on CCM. The extension of the middleware renders it more flexible and enables the transparent support for group communication. Unlike other approaches, the connector extension of the middleware is not a specific extension for fault-tolerance – fault tolerance is merely a good example of the enhanced flexibility that can be achieved within this component approach. It is then possible to implement distributed applications onto heterogeneous platforms running under different operating systems and communication stacks at low cost and with high implementation efficiency. The application are currently runs on a PC using Linux and on a GR-XC3S-1500 LEON development board using RTEMS OS (a Posix compliant). The latter is used to show that our approach may be used easily to design embedded systems.

The next steps are primarily a support for an automatic re-configuration of the application, for instance the transition between a nominal and a reduced-functionality mode. Re-configuration mechanisms in a non-FT context are already implemented by the project partner Thales; and recently we propose a model driven approach to help specify reconfigurability issues [7]. The challenges of the integration include for instance the replication of the component performing the reconfiguration steps. Another objective for the near future is to implement other replication styles than the active with vote and to examine footprint and performance overheads in detail.

References

1. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed system. *Journal of the ACM*, 43(2):225–267, July 1996.
2. T. Dumitras, D. Srivastava, and P. Narasimhan. Architecting and implementing versatile dependability. In *Architecting Dependable Systems III (WADS)*, volume 3549 of *Lecture Notes in Computer Science*, pages 212–231. Springer, 2005.
3. P. Feiler and A. Rugina. Dependability Modeling with the Architecture Analysis & Design Language (AADL). Technical report, 2007. CMU/SEI-2007-TN-043.
4. P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, 2006. CMU/SEI-2006-TN-011.
5. J. Fraga, F. Siqueira, and F. Favarim. Adaptive Fault-Tolerant Component Model. In *Ninth IEEE international workshop on Object-Oriented Real-Time Dependable Systems*, 2003.
6. B. Hamid. *Distributed fault-tolerance techniques for local computations*. PhD thesis, University of Bordeaux 1, 2007.
7. B. Hamid, A. Lanusse, A.Radermacher, and S. Gérard. Designing reconfigurable component systems with a model based approach. In *Workshop on Adaptive and Reconfigurable Embedded Systems, APRES'08 (to appear)*.
8. M.G. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.
9. A. Kobusinska, J. Kobusinski, and M. Szychowiak. An Analysis of distributed platforms applying replication mechanisms. Technical Report Report RA-014, Poznan University of Technology, 2001.
10. L. C. Lung, F. Favarim, G. T. Santos, and M. Correia. An Infrastructure for Adaptive Fault Tolerance on FT-CORBA. In *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 504–511, Washington, DC, USA, 2006. IEEE Computer Society.
11. OMG. *CORBA Core specification, Version 3.0.3*, 2004. OMG Document formal/2004-03-12.
12. OMG. *CORBA Core specification, Version 3.0.3*, 2004. OMG Document formal/2004-03-12.
13. OMG. *CORBA Component Model Specification, Version 4.0*, 4 2006. OMG Document formal/2006-04-01.
14. OMG. *Deployment and Configuration of Component Based Distributed Applications, v4.0*, 2006. OMG document ptc/2006-04-02.
15. OMG. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*, 5 2006. OMG Document formal/2006-05-02.
16. OMG. *UML Profile for MARTE*, 2007. OMG document ptc/07-08-04.
17. Y. Ren, M. Cukier, and W.H. Sanders. An adaptive algorithm for tolerating value faults and crash failures. *IEEE transaction on parallel an distributed systems*, 2:173–192, 2001.
18. S. Robert, A. Radermacher, V. Seignole, S. Gérard, V. Watine, and F. Terrier. Enhancing interaction support in the corba component model. In *From Specification to Embedded Systems Application*.
19. D. Schmidt. Model-driven engineering. *in IEEE computer*, 39(2):41–47, 2006.
20. J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *in Theory and Practice of Object Systems*, 3(1), 1997.