Fault-tolerance for Component-based Systems - An Automated Middleware Specialization Approach

Sumant Tambe, Akshay Dabholkar, and Aniruddha Gokhale
Dept. of EECS, Vanderbilt University, Nashville
{sutambe,aky,gokhale}@dre.vanderbilt.edu

Abstract

General-purpose middleware, by definition, cannot readily support domain-specific semantics without significant manual efforts in specializing the middleware. This paper presents GRAFT (GeneRative Aspects for Fault Tolerance), which is a model-driven, automated, and aspects-based approach for specializing general-purpose middleware with failure handling and recovery semantics imposed by a domain. Model-driven techniques are used to specify the special fault tolerance requirements, which are then transformed into middleware-level code artifacts using generative programming. Since the resulting fault tolerance semantics often crosscut the middleware architecture, GRAFT uses aspect-oriented programming to weave them into the original fabric of the general-purpose middleware. We evaluate the capabilities of GRAFT using a representative case study.

Keywords: Fault tolerance, middleware specialization, generative programming, aspects, model-based.

1 Introduction

Research in middleware over the past decade has significantly advanced the quality and feature-richness of contemporary general-purpose middleware (*e.g.,* CORBA, J2EE and .NET) that provides remoting abstractions such as remote procedure call (RPC) and events for interprocess communication. Support for different non-functional properties, such as fault tolerance, real-time, transactions and security are now readily available within the middleware and its services. Application domains as diverse as industrial automation and business processing use these middleware for reliable and robust coordination and communication between software components.

Despite these advances, general purpose middleware have limitations in how many diverse *domain-specific* semantics can they readily support *out-of-the-box*. For example, fault tolerant CORBA (FT-CORBA) defines the infrastructure- and application-controlled styles of consistency management. In the former, the middleware provides consistency to the applications; however, without knowing the semantics of the data, this style of consistency management can at best be coarse grained. On the other hand, in the application-controlled consistency management style, the applications must provide all these capabilities thereby incurring additional development efforts and maintenance costs. Often, it is desirable for middleware to manage consistency based on the data semantics requirements of the applications.

Since different application domains may impose different variations in fault tolerance (or for that matter, other forms of quality of service) requirements, these semantics cannot be supported out-of-the-box in general-purpose middleware since they are developed with an aim to be broadly applicable to a wide range of domains. Developing a proprietary middleware solution for each application domain is not a viable solution due to the high development and maintenance costs.

Resolving this tension requires answering two important questions. First, how can solutions to domain-specific fault tolerance requirements can be realized while leveraging low cost, general-purpose middleware without permanently modifying it? An approach based on aspect-oriented programming (AOP) [10] can be used to modularize the domain-specific semantics as *aspects*, which can then be woven into general-purpose middleware using aspect compilers. This creates *specialized* forms of general-purpose middleware that support the domain-imposed properties.

Many such solutions to specializing middleware exist [9, 11], however, these solutions are often handcrafted, which require a thorough understanding of the middleware design and implementation. The second question therefore is how can these specializations be

automated to overcome the tedious, error-prone, and expensive manual approaches? *Generative programming* [4] offers a promising choice to address this question.

In this paper we present GRAFT (*GeneRative Aspects for Fault-Tolerance*), which incorporates these solutions to specialize general-purpose middleware with domain-specific fault tolerance semantics. GRAFT uses domain-specific modeling and model-driven engineering (MDE) [3,15] techniques due to their ability to intuitively capture and modularize domain-specific requirements, and their inherent support for generative programming.

GRAFT uses a two stage process. In the first stage, domain-specific models, describing applications, their structure, and their communications, are annotated with fault-tolerance requirements of the domain using our domain-specific modeling language called the Component Availability Modeling Language (CAML). Aspect-oriented model weaving [6] is used to automatically transform these annotated structural models into refinements that incorporate the fault tolerance requirements captured using CAML. In the second stage, generators associated with CAML synthesize aspect code that realizes the runtime fault tolerance behavior, which is then woven into the middleware code using an aspect compiler.

We show the feasibility of our approach in the context of a representative case study taken from a warehouse material handling system that uses the Component-Integrated ACE ORB (CIAO) – a C++ implementation of Lightweight CORBA Component Model (LwCCM) [12] – for implementing various software components. Our empirical results demonstrate the savings in efforts to specialize general-purpose middleware.

2 Motivational Case Study

To better present our GRAFT solution, we illustrate a case study that benefits from GRAFT to realize its fault tolerance requirements. Our case study is a warehouse *material handling system* (MHS). A MHS provides automated monitoring, management, control, and flow of warehouse goods and assets. A MHS represents a class of conveyor systems used by couriers (*e.g.,* UPS, DHL, and Fedex), airport baggage handling, retailers (*e.g.,* Walmart and Target), food processing and bottling.

Architecture. The software components in the MHS architecture can be classified as (1) *management* components, which make decisions such as where to store incoming goods, (2) *material flow control* (MFC) components, which provide support for warehouse management components by determining the routes the goods have to traverse, and (3) *hardware interface layer* (HIL) components, which control MHS hardware, such as conveyor belts and flippers.

Figure 1 shows a subset of the MHS operations, where a MFC component directs goods within the warehouse using the route BELT A→BELT B or the route BELT A→BELT C. Flippers F and F′ assist in directing goods from BELT A to BELT B and BELT C, respectively. Further, as shown in Figure 1, HIL components, such as Motor Controllers (MC1, MC2, MC1′, MC2′) and the Flipper Controller (FC, FC′), control the belt motors and flippers, respectively. The MFC component instructs the Flipper Controller component to flip, which in turn instructs the Motor Controller components to start the motors and begin transporting goods.

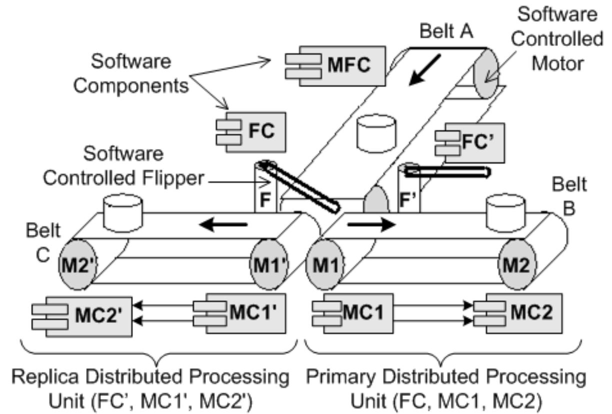

Figure 1. A Distributed Processing Unit Controlling Conveyor Belts

Domain-specific Fault Model. As goods are transported using different conveyor belts, faults could occur. Two broad kinds of faults are possible in the MHS system: (1) hardware faults, (*e.g.,* jamming of the flipper) and (2) software faults, (*e.g.,* MC or FC component crashes). Hardware faults in the MHS system are detected by their associated HIL components and communicated using application-specific software exceptions. Software faults, such as software component crashes, are detected by the clients of those components using system-level software exceptions generated by the underlying middleware. Both types of faults affect the reliable and correct operation of the MHS system, and are classified as *catastrophic* faults.

Domain-specific Failure Handling and Recovery Semantics. Failure recovery actions in MHS are based on *warm-passive* replication semantics. When catastrophic faults are detected in a MHS, the desired sys-

tem response is to shutdown the affected hardware assembly and activate a backup hardware assembly automatically. For example, when one of the motors of BELT B or flipper F fails, the MFC component should stop using the BELT B and route the packages via BELT C instead. The consequence of such a decision means that the HIL components associated with BELT B should be deactivated and those with BELT C as well as flipper F′ need to be activated.

The MHS thus imposes a *group*-based fault tolerance semantics on the software components controlling the physical hardware. If any one component of the group fails, the failure prevents the whole group from functioning and warrants a failover to another group. We call this group of components as a distributed processing unit (DPU) – in this case MC1, MC2 and FC for BELT B. Further, the clients of a DPU (*e.g.*, the MFC component) must failover to an alternative DPU if any of the components in the primary DPU fails.

System Design and Implementation Challenges. The software components of MHS are built using LwCCM and the associated CIAO middleware. Designing and implementing systems that support DPU group failure recovery semantics of a MHS using general-purpose middleware, such as LwCCM and its support for fault tolerance via FT-CORBA, is challenging because it lacks out-of-the-box support for (1) grouping multiple and distributed HIL components and treating them as a single unit of failure, (2) recovering a collection of HIL components together based on the group recovery semantics and redirecting all client MFC components to a replica group of HIL components.

Realizing these capabilities at application level impacts all the lifecycle phases of the application. First, application developers must modify their interface descriptions specified in IDL files to specify new types of exceptions, which indicate domain-specific fault conditions. Naturally, with changes in the interfaces, application developers must reprogram their application to conform to the modified interfaces. Modifying application source code to support failure handling semantics is not scalable as multiple components need to be modified to react to failures and provision failure recovery behavior. Further, such an approach results in crosscutting of failure handling code with that of the normal behavior across several component implementation modules.

An alternative approach to realizing this capability is to manually modify the general-purpose middleware and enhance it to recognize fault conditions and enforce failover by accounting for semantics, such as the DPU failover. Such modifications, however, are seldom restricted to a small portion of the middleware. Instead they tend to impact multiple different parts of the middleware. Naturally, a manual approach consumes significant development efforts and requires invasive and permanent changes to the middleware.

Overcoming these challenges requires an approach that automates middleware specialization and relieves applications from having to incur extra development effort. In particular, the following properties are desired of the proposed approach:

- **Property 1:** Application developers must not incur extra effort at design and development time to change IDL and system models to describe how application components are assembled and deployed. Moreover, any approach to capture the domain-specific requirements must be modularized to avoid scattering.
- **Property 2:** No manual efforts should be expended to incorporate the crosscutting changes to middleware while ensuring that no intrusive changes are made to the existing middleware. This implies, the domain-specific changes should remain decoupled from the original middleware, yet they must be made available at runtime.

3 GRAFT Process for Middleware Specialization

This section describes how GRAFT automates such domain-specific fault tolerance semantics within general-purpose middleware, while satisfying the properties mentioned before.

3.1 Overview of GRAFT

GRAFT is a two stage process to specialize middleware for domain-specific fault tolerance properties. Stage 1 in GRAFT leverages existing structural models of applications modeled as component assemblies, and annotates them with domain-specific fault tolerance requirements using our CAML language. The C-SAW [17] aspect-oriented model weaver is then used to transform the annotated models into those comprising new structural elements corresponding to the fault tolerance requirements (*e.g.,* replica). This transformation step is necessary to obtain platform-specific metadata (*e.g.,* XML deployment descriptors) for deployment and configuration engines from the output model of the transformation (See Section 3.2 for a detailed description.)

In stage 2, CAML models are used again to generated code that otherwise would have been written manually to carry out specialization of the middleware. The generated code is modularized using As-

pectC++ [18] language – an aspect-oriented extension to the traditional C++ language. Finally, GRAFT uses the AspectC++ [18] compiler to weave in the specialized code into the CIAO middleware (See Section 3.3 for a detailed description). The entire GRAFT process is orchestrated within our CoSMIC [5] MDE framework.

3.2 Stage 1: Design-time Support for Specialization

Specializing middleware for fault tolerance properties imposed by a domain requires means to externalize the requirements so that an automated tool can process them. Modifying IDL, application and/or middleware code to specify the requirements is not the right level of abstraction since all these approaches are crosscutting and invasive, which require significant modifications to and maintenance of different artifacts. Satisfying Property 1 from Section 2 requires that the application design and its structure (*i.e.*, the composition of its components) be shielded from the domain-imposed fault tolerance requirements.

Thus, it is desirable to specify these requirements using approaches that are intuitive, non invasive, and which promote automation. Model-driven engineering (MDE) [15], which uses domain-specific modeling languages (DSML) [7], provides the right level of abstraction at which such domain-specific requirements can be captured. Moreover, since MDE is a widely used approach to designing large-scale applications, it becomes a natural choice to specify these domain-specific requirements.

I. Specifying requirements using MDE: GRAFT provides a DSML called the Component Availability Modeling Language (CAML), which provides capabilities to annotate application structural models[1] with domain-specific fault tolerance requirements. CAML is developed using the Generic Modeling Environment (GME) [2], which provides a meta-programmable design environment for developing domain-specific graphical modeling languages. GME is capable of projecting different parts of a model on different graphical views. CAML leverages this capability to provide *separation of concerns* where application structural models are visually separate from domain-specific fault tolerance requirements. CAML's fault tolerance requirements are separated from the application's structural models by projecting them only in the *QoS view* but not in the *structural view* as shown in Figure 2. This property ensures that any domain-specific annotations of fault tolerance requirements to application structural models are represented as superimposition or an overlay, which preserves the original design and structure of the application.

[1] Structural models for component-based systems are built using our CoSMIC [5] modeling tool chain.

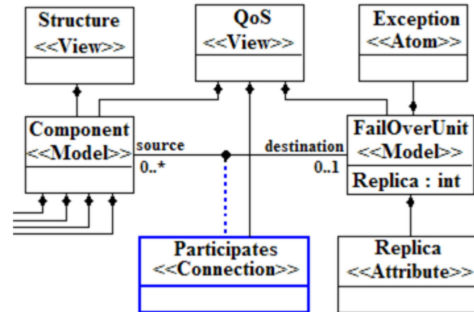

Figure 2. Availability Annotation Metamodel of CAML

To capture the DPU semantics of our case study from Section 2, we developed an annotation metamodel in CAML, which is shown in Figure 2. To modularize the DPU fault-tolerance requirements at the modeling level, CAML provides a key modeling abstraction called *FailOverUnit* that supports group recovery semantics for the components associated with the relation *Participates*. One or more components can be grouped together and treated as a DPU when they are associated with a single FailOverUnit.

FailOverUnit captures the degree of replication of a DPU as a unit using an integer attribute called *Replica*. Finally, the FailOverUnit also provides support for modeling system level and application-specific catastrophic exceptions because group recovery semantics are critically dependent on such observable exceptions. Notice how CAML ensures that the FailOverUnit is projected only within the *QoS view* while the *structural view* continues to be part of the functional concerns.

II. Automating structural changes based on modularized requirements:

Middleware specializations are required for all fault tolerant component instances in the system including replicas. Although CAML modularizes the domain-specific fault tolerance requirements by providing the annotation capabilities, these annotations by themselves do not produce extra component instances at run-time. The model interpreters in CoSMIC that synthesize deployment and configuration metadata for all the components, do not understand the annotations either. Therefore, from the perspective of the structural view, the application model is not complete unless the replicas of the protected components are made available. Thus the original structural models of the ap-

plication must be transformed using these annotated requirements, however, without the developer expending any effort to satisfy Property 1 in Section 2.

In our case study, such a transformation requires several steps including (1) duplicating models of the primary components participating in a FailOverUnit, and (2) duplicating their interconnections so that the necessary connections can be established at deployment time for the replica DPU. GRAFT uses aspect-oriented model weaving [6] support provided by the Constraint-Specification Aspect Weaver (C-SAW) [17] tool to transform the annotated structural models. The C-SAW weaver is a generalized model-to-model transformation engine for manipulating domain-specific models. C-SAW uses a language called Embedded Constraint Language (ECL) to specify transformations.

As shown in Figure 3, we developed a model-to-model transformation using C-SAW that takes a CAML model having annotated fault tolerance requirements as an input model, and generates a structural output model in response to the fault tolerance requirements. The new structural model is then traversed by existing model interpreters to produce metadata for packaging, assembling, deployment and configuration.

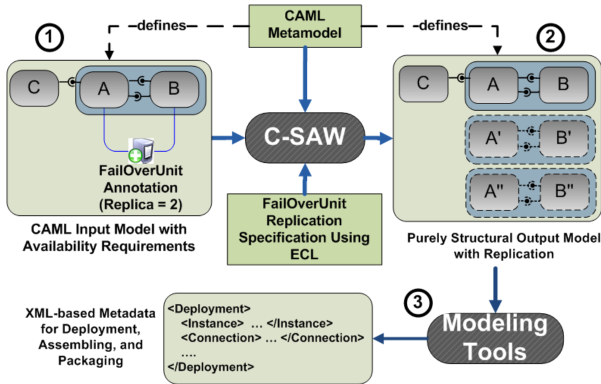

Figure 3. Automated Model Weaving Using C-SAW and FailOverUnit Replication Specification in ECL

Step 1 in Figure 3 represents a CAML model, which has fault-tolerance requirements modeled along with the application's structure. Step 2 in Figure 3 shows how a CAML model is automatically enriched using a transformation written using ECL. The transformation specification is parameterized and accepts the number of desired replicas as a parameter, which is specified as an integer attribute of a FailOverUnit in Step 1. Step 3 then uses existing model interpreters, which operate on the transformed models to produce platform-specific metadata.

The transformation is divided into multiple ECL strategies that perform two important steps. First, it creates clones of the participant components of a FailOverUnit. Second, it replicates the interconnections between the primary components into replica components. The result of these two steps is that structurally identical copies of the primary component models are created. This is necessary because the deployment and configuration tools do not distinguish between the primary components and the replica components.

3.3 Stage 2: Runtime Support for Specialization

The MDE tools [5] deploy the entire system and configure the middleware, however, they do not specialize the middleware. It is necessary for the middleware to be specialized using the domain-specific fault tolerance semantics specified in the MDE tools, without expending any manual effort. To address this challenge, GRAFT uses a deployment-time *generative* approach that augments general-purpose middleware with the desired *specializations*.

For our case study, GRAFT specializes the client-side middleware stubs. Client-side middleware stubs are used to communicate exceptions to client-side applications so that they can initiate appropriate recovery procedure in response to that. As mentioned in Section 2, these exceptions could be raised because of (1) hardware faults detected by the server or (2) software failure of the server side component itself. Both are examples of *catastrophic* exceptions, in response to which clients must initiate group recovery. To simplify developers' job, GRAFT generates code at deployment-time that augments the *behavior* of the middleware-generated stubs to catch failure exceptions, and initiate domain-specific failure recovery actions.

GRAFT provides a model interpreter, which (1) traverses the CAML model, (2) identifies the components that participate in FailOverUnits, (3) identifies the components that are clients of the FailOverUnit participant components, and (4) generates modularized source code that provides failure detection and recovery as shown by Step 1 in Figure 4. Depending upon the role of the component, two different types of behaviors are generated by the interpreter.

We have identified two different roles of components with respect to a FailOverUnit: (1) participants of a FailOverUnit (*e.g.,* FC component) and (2) non-participant client components that are directly communicating with one or more participants of the FailOverUnit (*e.g.,* MFC component). The participants of a DPU do not failover, however, clients of

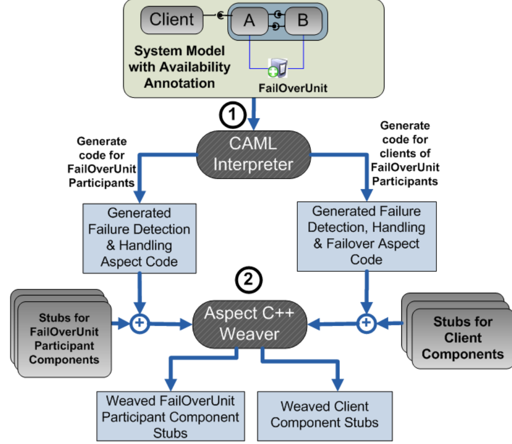

Figure 4. Automated Generation of Failure Detection and Handling Code

a DPU fail over to a replica FailOverUnit. To allow this difference in the behavior, failover code is generated only for the client components whereas the code for FailOverUnit participant components do not perform failover; instead they trigger failover in the client components of the FailOverUnit.

GRAFT encodes this difference in behavior by generating different AspectC++ code for each component associated with a FailOverUnit depending upon whether the component is a participant or a client. For participant components, for every method in the interface that can potentially raise a catastrophic exception, an *around* advice is generated that catches exceptions representing *catastrophic* failure and initiates a shutdown procedure for all the participant components. For the client components, however, a different around advice is generated that not only detects the failure and initiates a group shutdown procedure but also performs an automatic failover to a replica FailOverUnit.

To modularize and transparently weave the failure detection and recovery functionality within the stubs, GRAFT leverages Aspect-oriented Programming (AOP) [10] support provided by the AspectC++ [18] compiler. The CAML model interpreter generates AspectC++ code,[2] which is then woven by the AspectC++ compiler into stubs at the client side producing specialized stub implementations as shown by Step 2 in Figure 4. Finally, the specialized source code of the stubs are compiled using a traditional C++ compiler.

[2]Due to space restrictions we are not showing the generated aspect code.

4 Evaluation of GRAFT

We evaluate GRAFT by measuring the efforts saved to specialize middleware in the context of the MHS case study of Section 2. Additionally we also qualitatively validate the runtime behavior of the specialized middleware in meeting the fault tolerance requirements of the MHS case study.

Qualitative validation of runtime behavior. Figure 5 shows how the specialized stubs generated by GRAFT react to failures at runtime and provide group recovery semantics. To control the lifecycle of the components, the aspect code communicates with domain application manager (DAM), which is a standard deployment and configuration infrastructure service defined in LwCCM. It provides high-level application programming interface (API) to manage lifecycle of application components. Below, we describe the steps taken by GRAFT when a catastrophic exception is raised.

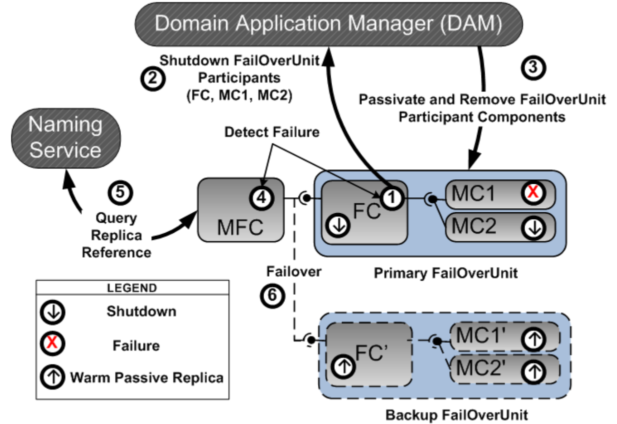

Figure 5. Runtime Steps Showing Group Recovery Using GRAFT

1. As shown in Figure 5, MFC component directly communicates with the FC component, which in turn communicates with MC1 and MC2 components. Consider a scenario where FC makes a call on MC1 and MC1 detects a motor failure and raises *MotorFailureException*. The exception is caught by the generated aspect code in FC indicated by (1) in Figure 5.
2. The specialized stubs in FC, initiate shutdown of the primary DPU by instructing the DAM to remove participating components of the primary DPU (FC, MC1, and MC2), including itself.
3. DAM instructs the containers hosting the primary DPU components (FC, MC1, and MC2) to passivate and remove the components.

	Fault-tolerance Modeling Efforts			Fault-tolerance Programming Efforts		
Component Name	**# of original connections**	**# of replica components**	**# of replica connections**	**# of try blocks**	**# of catch blocks**	**Total # of lines**
Material Flow Control	1 / 1	0 / 0	2 / 0	1 / 0	3 / 0	45 / 0
Flipper Controller	2 / 2	2 / 0	4 / 0	2 / 0	6 / 0	90 / 0
Motor Controller 1	1 / 1	2 / 0	2 / 0	0 / 0	0 / 0	0 / 0
Motor Controller 2	2 / 1	2 / 0	2 / 0	0 / 0	0 / 0	0 / 0

Table 1. Manual Efforts in Developing ITS Casestudy Without/With GRAFT

4. Removal of FC component triggers a system-level exception at the MFC component, which is again caught by the specialized stub at MFC-side.
5. The specialized stubs for MFC fetch a reference of FC′ from the naming service. The naming service is assumed to be pre-configured at deployment-time with lookup information for all the components in the system.
6. MFC successfully fails over to the replica DPU (FC′, MC1′, and MC2′) and resumes the earlier incomplete remote function call. Finally, FC′ communicates with MC1′ and MC2′ to drive the belt motors of the backup BELT C and continues the operation of MHS system without interruption.

Evaluating savings in effort to specialize middleware. Table 1 shows the manual efforts saved by adopting GRAFT's approach in designing and developing the MHS case study described in Section 2. The table shows that there is reduction in the efforts of modeling replica components and connections for all the four components. The declarative nature of CAML's FailOverUnit annotations and the automated model-to-model transformation thereafter, obviates the need for modeling the replica components and connections explicitly, resulting in a modular design of the MHS system.

A significant reduction in programming efforts is achieved due to automatic generation of code that handles failure conditions at runtime in the MHS system. The generated code for each component is different depending upon the number of remote interfaces used by a component, the number of methods in each remote interface, and the types of exceptions raised by the methods. The number of `try` blocks in Table 1 corresponds to the number of remote methods whereas the number of `catch` blocks correspond to the number of exceptions.

For example, when MFC component invokes a method of the FC component, 45 lines of aspect code is generated to handle group recovery semantics for that one function call alone. GRAFT's approach yields higher savings in modeling and programming efforts for larger, more complex systems, which may have hundreds of components with tens of them requiring fault-tolerance capabilities.

5 Related Work

Polze *et. al.,* [13], use AOP techniques to describe fault-tolerance as a non-functional component property and focus on the automatic generation and replication of protected components and fault tolerant middleware services based on aspect information and demonstrate service configuration through a graphical user-interface. GRAFT's approach overlaps considerably with this approach, however, GRAFT provides automatic support for *domain-specific* failure handling semantics to applications as opposed to common middleware services.

ACT [14] enables runtime adaptation by transparently weaving adaptive code in the ORBs at runtime, which intercepts and adapts the requests, replies and exceptions that pass through the ORBs. ACT uses a runtime approach, which is based on portable interceptors. Although ACT's generic and rule-based interceptor capabilities could be used to provide various failure management semantics at runtime, it provides no support for capturing design-, deployment-time domain-specific fault-tolerance requirements.

Sevilla *et. al.,* [16] describe an automatic code generation approach for distribution, fault tolerance and load balancing aspects of component based systems. GRAFT on the other hand provides mechanisms that specifically support group failure recovery semantics based on the model- as well as code-based weaving of component-based systems along with runtime monitoring infrastructure support.

Afonso *et. al.,* [1] propose a manual AOP-based approach for modularizing existing fault tolerance code from the legacy threaded applications. But the modularization and weaving must be done manually at application level whereas GRAFT automatically weaves the domain-specific failure handling code in general purpose middleware to provide transparent recovery through client-side failover.

JReplica [8] attempts to modularize the replication aspect of fault tolerance using AOP. JReplica extends

UML with a replication language to define replication policies at a finer domain-independent granularity whereas GRAFT supports domain-specific failure semantics such as group failover. However JReplica does not provide the middleware implementation for fault recovery whereas GRAFT automatically generates and weaves the necessary failure recovery code transparently in the middleware. A common feature between JReplica and GRAFT is that they both ensure that only the required method invocation paths are intercepted.

6 Conclusion

This paper presented GRAFT, which is a two stage generative approach to specialize general purpose middleware with domain-specific fault tolerance semantics. General purpose middleware are designed to support a wide range of application domains and can only provide generic solutions for non-functional properties, such as fault tolerance. When the domain imposes a rich set of fault tolerance requirements, GRAFT's approach is to automatically specialize the middleware using contemporary aspect-oriented languages to provide the desired failure recovery semantics without any intrusive modifications to the applications and middleware.

The capabilities of GRAFT were demonstrated in the context of an automated material handling system (MHS) case study for group-wide failure recovery semantics. The MDE capabilities of GRAFT including CAML are available in open source as part of our CoSMIC toolsuite at `www.dre.vanderbilt.edu/cosmic`.

References

[1] F. Afonso, C. Silva, N. Brito, S. Montenegro, and A. Tavares. Aspect-oriented fault tolerance for real-time embedded systems. In *ACP4IS '08: Proceedings of the 7th workshop on Aspects, components, and patterns for infrastructure software*, 2008.

[2] Ákos Lédeczi, Árpád Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[3] C. Atkinson and T. Kuhne. Model-driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[5] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. S. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt. Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. *The Journal of Science of Computer Programming: Special Issue on Foundations and Applications of Model Driven Architecture (MDA)*, 73(1):39–58, 2008.

[6] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan. An Approach for Supporting Aspect-Oriented Domain Modeling. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, 2003.

[7] J. Gray, J. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle. Domain-Specific Modeling. In *CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.)*, pages 7.1–7.20. CRC Press, May 2007.

[8] J. Herrero, F. Sanchez, and M. Toro. Fault tolerance aop approach. In *Workshop on Aspect-Oriented Programming and Separation of Concerns*, 2001.

[9] J. Jin and K. Nahrstedt. On Exploring Performance Optimizations in Web Service Composition. In *Middleware*, pages 115–134, 2004.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[11] S. Mohapatra, R. Cornea, H. Oh, K. Lee, M. Kim, N. D. Dutt, R. Gupta, A. Nicolau, S. K. Shukla, and N. Venkatasubramanian. A Cross-Layer Approach for Power-Performance Optimization in Distributed Mobile Systems. In *IPDPS*, 2005.

[12] Object Management Group. *Lightweight CCM FTF Convenience Document*, ptc/04-06-10 edition, June 2004.

[13] A. Polze, J. Schwarz, and M. Malek. Automatic generation of fault-tolerant corba-services. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, 2000.

[14] S. M. Sadjadi and P. K. McKinley. Act: An adaptive corba template to support unanticipated adaptation. In *Proc. of ICDCS. (2004)*.

[15] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[16] D. Sevilla, J. M. Garcia, and A. Gomez. Aspect-oriented programing techniques to support distribution, fault tolerance, and load balancing in the corba-lc component model. *nca*, 00:195–204, 2007.

[17] Software Composition and Modeling (Softcom) Laboratory. Constraint-Specification Aspect Weaver (C-SAW). www.cis.uab.edu/ gray/Research/C-SAW, University of Alabama, Birmingham, AL.

[18] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, 2002.