MDDPro: Model-Driven Dependability Provisioning in Enterprise Distributed Real-time and Embedded Systems *

Sumant Tambe[1], Jaiganesh Balasubramanian[1], Aniruddha Gokhale[1], and Thomas Damiano[2]

[1] Vanderbilt University, Nashville, TN, USA
Email: {sutambe,jai,gokhale}@dre.vanderbilt.edu
[2] MITRE Corporation

Abstract

Service oriented architecture (SOA) design principles are increasingly being adopted to develop distributed real-time and embedded (DRE) systems, such as avionics mission computing, due to the availability of real-time component middleware platforms. Traditional approaches to fault tolerance that rely on replication and recovery of a single server or a single host do not work in this paradigm since the fault management schemes must now account for the timely and simultaneous failover of groups of entities while improving system availability by minimizing the risk of simultaneous failures of replicated entities. This paper describes MDDPro, a model-driven dependability provisioning tool for DRE systems. MDDPro provides intuitive modeling abstractions to specify failover requirements of DRE systems at different granularities. MDDPro enables plugging in different replica placement algorithms to improve system availability. Finally, its generative capabilities automate the deployment and configuration of the DRE system on the underlying platforms.

Keywords: Dependability Design Tools, Model-Driven Engineering, Generative programming, Real-time SOA systems

1 Introduction

Dependability is a crucial design consideration for mission-critical distributed real-time and embedded (DRE) systems, such as avionics mission computing, and supervisory control and data acquisition (SCADA) systems. DRE systems development processes are increasingly adopting the service oriented architecture (SOA) design principles due in large part to the availability of real-time component middleware platforms, such as the Lightweight CORBA Component Model (LwCCM) [1]. The SOA approach when applied to DRE systems gives rise to

* This work is supported in part or whole by subcontracts from LMCO ATL and BBN for the DARPA Adaptive and Reflective Middleware Systems Program.

what we term *enterprise DRE systems*, which are a loose coupling of interacting real-time and embedded services that are composed, assembled, deployed and configured on the underlying platforms to realize the end-to-end functionality. With the newer SOA-style design, however, new challenges emerge in the design of dependability management solutions for enterprise DRE systems, which stem from the following limitations of contemporary mechanisms:

Limitations of existing dependability mechanisms. A substantial amount of research in dependable distributed computing has predominantly concentrated on providing fault tolerance solutions to intrinsically homogeneous, two-tier client-server systems with mostly request-response semantics or cluster-based server systems with transactional semantics. These research artifacts most often assume single language and single platform systems, which when incorporated in middleware platforms form point solutions, limit reuse, and are too restrictive for enterprise DRE systems.

Lack of support for mixed-mode dependability semantics. DRE systems of interest to us require mix mode dependability wherein parts of the system may require ultra high availability calling for solutions that require active replication schemes while other parts of the systems may demand passive forms of replication to overcome issues with non-determinism.

Lack of support for variable failover granularity and failure risk management. In enterprise DRE systems, traditional approaches to fault tolerance that rely on replication and recovery of a single server process or a single host are not sufficient since the fault management schemes must now account for the timely and simultaneous failover of groups of entities while also improving the system availability by minimizing the risk of simultaneous failures of groups of replicated entities.

Lack of intuitive and scalable dependability provisioning tools. Standardized middleware solutions to dependability, such as FT-CORBA [2], provide a *one-size-fits-all* approach, which do not support the different properties, such as mixed-mode dependability semantics, required by enterprise DRE systems. Moreover, dependability provisioning in DRE systems tend to use imperative, programmatic mechanisms which are tedious, inflexible, non reusable and error prone, and cannot scale to large enterprise DRE systems, where heterogeneity of the underlying platforms is the norm.

To address the challenges outlined above, design-time tools that can automate the dependability provisioning problem for enterprise DRE systems are needed. This paper describes *MDDPro* (Model Driven Dependability Provisioning), which is a Model-driven Engineering (MDE) [3] tool for design-time dependability provisioning in enterprise DRE systems. We demonstrate

- how the intuitive modeling capabilities in our tool can model fault tolerance elements in DRE systems at different granularities,
- how system availability can be enhanced by applying replica placement decision algorithms on the models, and
- how generative programming capabilities in the tool can be used to rapidly and reliably provision dependability in DRE systems.

The rest of the paper is organized as follows: Section 2 describes the challenges in designing the dependability provisioning tool for enterprise DRE systems; Section 3 describes the design and implementation of our dependability provisioning tool; Section 4 describes related research; and Section 5 provides concluding remarks and directions for future research.

2 Design Considerations for Automated Dependability Provisioning

Several factors must be considered when developing a dependability provisioning tool, such as MDDPro, for enterprise DRE systems. In this section we use a sample enterprise DRE system as a guiding example to outline the requirements of such a design-time tool.

2.1 Enterprise DRE System Case Study

Figure 1 illustrates a sample enterprise DRE system drawn from representative domains, such as avionics mission computing or shipboard computing, where variables of interest are sensed by the sensor equipment, which are software controlled and fed to a set of planners who determine the appropriate control action to be taken, and subsequently relay this information to the actuator software components.

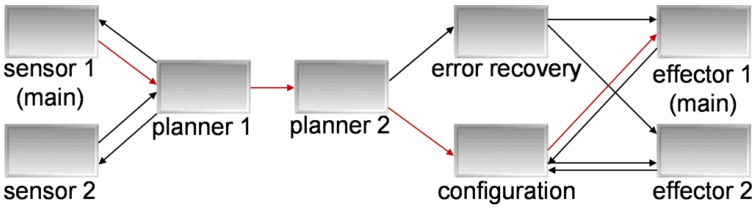

Fig. 1: A Sample Enterprise DRE System

Enterprise DRE systems are often deployed over heterogeneous platforms, which consist of multiple different networks, hardware and several layers of software. We consider the fact that failures may occur in any of these entities. For example, node failures, operating system crashes, middleware broker process failures, and even network link failures are common. In our current discussion we do not consider multiple cascaded failures.

Quite often the critical functionality of enterprise DRE systems is spread across multiple components. For example, the planning activity in Figure 1 is spread across two planning components, which could be deployed in separate application servers on different hosts. Since these distributed set of components form a unit of critical functionality, for high availability and even for the correct

operation of the system, it may be required that all such components in the critical path be protected against failure.

Moreover, if any of these individual components fail, it may not be sufficient to recover only the failed component but rather the failover should recover a group of critical components. This is because failure recovery takes finite amount of time and therefore by the time the failed functionality is recovered, the system may lose some critical system events. Therefore, it is highly desirable in such situations to failover to another replica of the protected group of the components although the failure may occur in only a single component. Thus, the fault recovery granularity can be much larger than the system elements affected by the single failure. The functionality and the topology of replica workflow could be different from that of the primary set of components to account for graceful degradation.

Risk management and availability considerations in enterprise DRE systems involve how individual or groups of critical components are replicated and deployed. Effective deployment of replica (or replica groups) minimize the risk of simultaneous failures in individual replica groups thereby improving the availability of the system.

2.2 Design Considerations

Using the enterprise DRE system case study illustrated in Figure 1 and the dependability management requirements outlined above, we now describe the design considerations for an automated dependability provisioning tool for enterprise DRE systems. In the following we describe the desired characteristics of such a design tool.

1. **Variable granularity of system protection:** Enterprise DRE systems are composed of several independently deployable assemblies of components that communicate together in a workflow fashion to carry out the system's functionality. Quite often the unit of modularity in the system design is larger than a single deployed component and results in some critical functionality of the system being spread across multiple components and/or assemblies. As outlined in the case study, in terms of the availability perspective, the entire critical functionality which is spread across multiple components must now be protected from failures. Moreover, failure of any one component in the workflow now implies the failure of the entire flow. In such a situation, the system must failover to a redundant workflow as opposed to a single component. One strategy for the failover mechanism could be to allow graceful degradation. The functionality of the replica components may not be the exact duplicate of the original. For example, the replica component can possibly implement an algorithm that is less resource hungry compared to the primary.

   A design-time tool must allow the specification of these requirements of enterprise DRE system. Section 3.2 describes how MDDPro provides intuitive abstractions to capture these dependability requirements of enterprise DRE systems.

2. **Mixed-mode dependability requirements:** Enterprise DRE systems are large-scale and comprise several different components each of which accomplishing specific tasks of the entire system functionality. Some parts of the system may require ultra high reliability mandating active replication schemes. However, due to the overhead associated with active replication and the non determinism issues [4, 5] involved in active replication, it may be necessary to restrict the use of active replication to a small part of the enterprise DRE systems. Other parts of the system may then use other forms of replication, such as passive replication, or depend on simple restart mechanisms depending on the criticality of the component and available resources in the system.

   The design-time tool must enable enterprise DRE system developers to capture these mixed-mode dependability semantics of the system. When combined with the granularity of protection units and other performance requirements of the system, this provisioning task becomes complex to perform manually using ad hoc and programmatic techniques. Section 3.2 describes how MDDPro provides intuitive abstractions to capture mixed-mode dependability requirements of enterprise DRE systems.
3. **Effective replica deployment for maximizing availability:** As alluded to above, enterprise DRE systems may have a number of different protected units of functionality that are assembled together to form the system. Moreover, different parts of the system may use different replication schemes. Considering both these requirements, it is now necessary to introduce redundancy in the system that accounts for the units of protection used and the replication styles used. Redundancy in the system improves system availability, however, high levels of reliability are realized only when replicas are placed in such a way that the risk of simultaneous failures of replicas is minimized. Effective replica placement also impacts several other performance characteristics of the entire system. For example, effective replica placement may be necessary to maintain a bounded and fast state synchronization among the replicas.

   A design-time tool can be used to ensure that the system simultaneously satisfies multiple QoS requirements such as performance, predictability and availability, by incorporating deployment state space search algorithms that automatically find effective deployments. This feature boils down to the general problem of constraint satisfaction. Optimality is a harder problem than constraint satisfaction, however, we do not consider it yet in our design. Section 3.3 describes how we have designed our MDDPro tool that can plug in different replica placement algorithms that find effective deployments for enterprise DRE systems.
4. **Automated provisioning of dependability:** Even though the modeling techniques can help capture dependability requirements while replica placement algorithms can provide effective deployment decisions, these must ultimately be realized in the context of the underlying hosting platforms, such as the component middleware. Component middleware often use XML meta-

data that describes how components of an enterprise DRE system should be hosted in the middleware and how they must be connected to each other. For large-scale systems, the amount of metadata becomes very large and ad hoc techniques, such as handcrafting these descriptors becomes infeasible and error prone.

Dependability provisioning makes this task harder since the metadata must now account for the protection units and provisioning the multiple replication schemes within the enterprise DRE system. This requires substantial degree of middleware configuration by allocating different resources end-to-end. Replication adds to the number of connections that must be established between the different protection units and their replicas. The replication style makes this task even harder. For example, when active replication is used, the middleware must be configured to use a group communication substrate that is used by the communication between replicas. On the other hand, in passive replication, the secondary replicas must be provisioned on the middleware to accept periodic state updates from the primary. Section 3.4 describes how generative programming [6] techniques used within our MDDPro tool automates the metadata generation to provision dependability for enterprise DRE systems within the middleware platforms.

Solution Approach. Model Driven Engineering (MDE) [3] is a promising approach to provision the dependability requirements for enterprise DRE systems because it raises the level of the abstraction of system design to a level higher than third-generation programming languages by providing a scalable and intuitive abstractions that are closer to the domain. The *model-per-concern* paradigm within MDE alleviates system complexity because it abstracts away the irrelevant details from the developer's current "view" of the system. Generative tools provided by MDE approaches can seamlessly integrate multiple views of the system and produce a consistent set of metadata used by underlying hardware/software platforms for configuration. The MDDPro tool described in this paper is therefore based on the MDE approach.

3 Dependability Provisioning using Model-driven Engineering

In this section we describe the design and implementation of our MDDPro design-time, automated dependability provisioning tool, which uses a model-driven engineering (MDE) approach in its design and satisfies the requirements of such a tool outlined in Section 2.2.

3.1 Overview of Enabling Technologies

Before delving into the details of our design-time dependability provisioning tool, we first provide an overview of the enabling technologies we have leveraged to develop MDDPro.

MDDPro has been developed in the context of the CoSMIC (Component Synthesis with Model Integrated Computing) [7] MDE toolsuite. CoSMIC is an open source MDE tool suite used to simplify the development of component-based DRE applications focusing particularly on the assembly, deployment, configuration, and validation of component-based enterprise DRE systems. CoSMIC comprises a collection of *domain-specific modeling languages* (DSMLs), which define the concepts, relationships, and constraints used to express domain entities [8], and generative programming capabilities that automate the different development concerns of DRE systems.

The different capabilities in CoSMIC including the MDDPro tool described in this paper have been developed using the Generic Modeling Environment (GME) [9]. GME is a metaprogrammable modeling environment that enables domain experts to develop visual modeling languages and generative tools associated with those languages. The modeling languages in GME are represented as metamodels. A metamodel in GME depicts a class diagram using UML-like constructs showcasing the elements of the modeling language and how they are associated with each other.

A key CoSMIC DSML developed in GME is the *Platform Independent Component Modeling Language* (PICML) [10], which enables graphical manipulation of modeling elements, such as component ports and attributes. PICML also performs various types of generative actions, such as synthesizing XML-based deployment plan descriptors defined in the OMG Deployment and Configuration (D&C) specification [11]. CoSMIC provides the *Component QoS Modeling Language* (CQML), which is a mapping of the platform-independent PICML models to models that are specific to the lightweight CORBA Component Model. Figure 2 illustrates the CQML model for the enterprise DRE system case study from Figure 1. Our MDDPro tool is an enhancement to the CQML DSML and its generative capabilities.

3.2 Modeling Dependability Requirements in MDDPro

We now describe how the MDDPro tool addresses Requirements (1) and (2) described in Section 2.2. CQML allows modelers to annotate the system elements modeled with platform-specific details and different quality of service (QoS) requirements as shown in Figure 2. MDDPro is responsible for the dependability QoS attributes in CQML. The artifacts that can be annotated are component instances, component implementations, connections between component ports, component assemblies, among others.

MDDPro allows an enterprise DRE system deployer to model the dependability requirements in the QoS view of the DRE system as shown in Figure 3. The QoS view leverages the basic structure of the DRE system in terms of the component instances in an assembly, component ports and their inter connections. It allows FT elements to be modeled orthogonally to the system components and therefore achieves separation of dependability concerns from the primary system composition and functionality concerns.

The following modeling elements are supported within MDDPro:

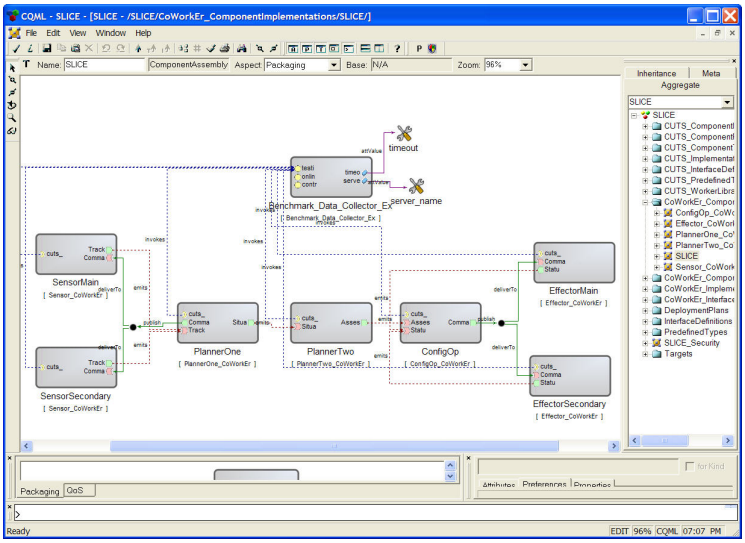

Fig. 2: CQML Model of the Enterprise DRE System Case Study

- **Failover units (FOUs),** which enable control over the granularity of protected system components, such as software components, component assemblies, or entire component workflows. Failure of any one element belonging to a FOU is treated equivalent to the failure of all the elements in the FOU and the system effectively "fails over" to another replica of the FOU. This modeling abstraction not only captures the failover granularities of system entities, but also the degree of replication for each FOU and other systemic requirements, such as the periodicity of liveness monitoring for FOUs. The degree of replication is represented as a pair of numbers representing minimum and maximum number of replicas. The programming language artifacts that implement the replica components could be different from that of the primary components allowing graceful degradation of the functionality if the dependability solution desires it.

  Frequently, the liveness of distributed components is monitored using a "heart beat" protocol. The frequency of the heartbeat is one configurable parameter in the liveness monitoring, which can be configured in MDDPro. The heartbeat itself is configurable in two ways: *push model* or *pull model*. Thus, the directionality of the heartbeat can also be configured in MDDPro. In Section 3.4 we show how modeling of FOUs enable us to automatically synthesize and configure liveness monitoring components as well as heartbeat producing components. Conceptually, a FOU is an abstraction to capture the availability requirements at the control plane of the dependability solution.
- **Replication groups (RGs),** which allows capturing the replication requirements of software components within a FOU. These models specify replication strategies, such as active, passive or other variants, and the state

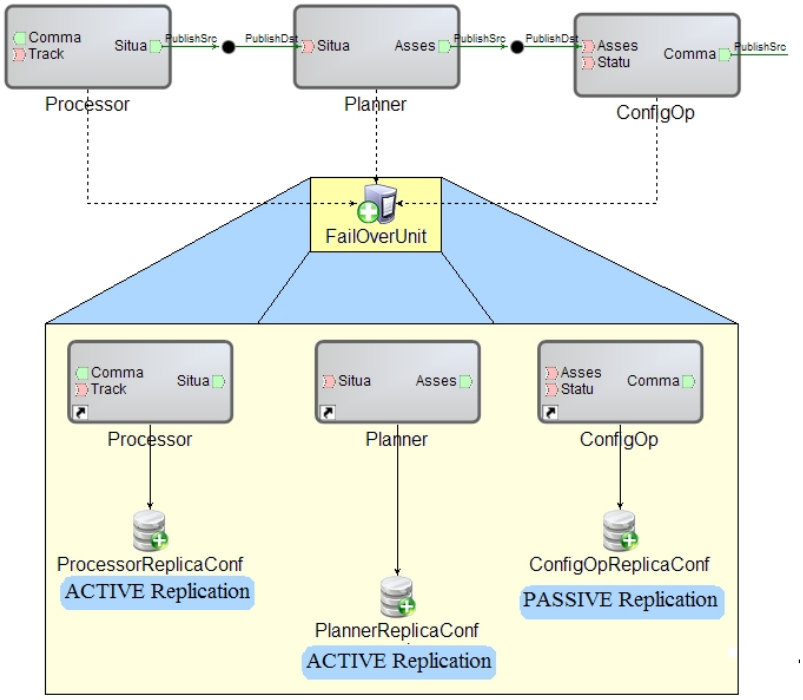

Fig. 3: Availability Requirements Modeling in CQML

synchronization policies for components. A replication group captures the configuration parameters related to the data plane of the deployment solution. Multiple replicas of the system components synchronize their state with each other as per the configuration of the data plane. For example, data synchronization frequency of the replicas is configurable. Moreover, the topology of state synchronization among replicas is also a data plane level configuration issue handled in MDDPro.

- **Shared Risk Groups (SRGs),** which defines one way of grouping of the resources in the target network of the applications that share a risk of simultaneous failure. Application components share a risk of simultaneous failure by virtue of the failure of the resources they share, such as processes, nodes, racks or even data centers on which they are hosted. Risk factors are determined by assigning the metrics, such as co-failure probabilities to a hierarchy of the network resources in a risk group that affects the availability of the system. The computation of the co-failure probabilities themselves is beyond the scope of this paper and is assumed to be done apriori using reliability engineering methodologies.

  The primary purpose behind modeling the shared risk groups and their respective co-failure probabilities is to facilitate automated deployment deci-

sions of the components in the system such that the probability of failure of entire system is minimized thereby increasing the availability. One way of reducing the co-failure probability is to increase the physical distance between the nodes where the components are deployed. Here, the physical distance can be thought of as the distance from a remote host or a remote blade or a remote data center and so on. An advantage of using distance metric is that it is simpler and quite intuitive than co-failure probability. In Section 3.3 we show how the shared risk group model is used by the MDDPro model interpreter to determine a suitable and effective deployment that satisfies the availability requirements and minimizes risks of simultaneous failures. In our prototype implementation of the algorithm we use the simpler distance metric to guide the decision of the replica placement.

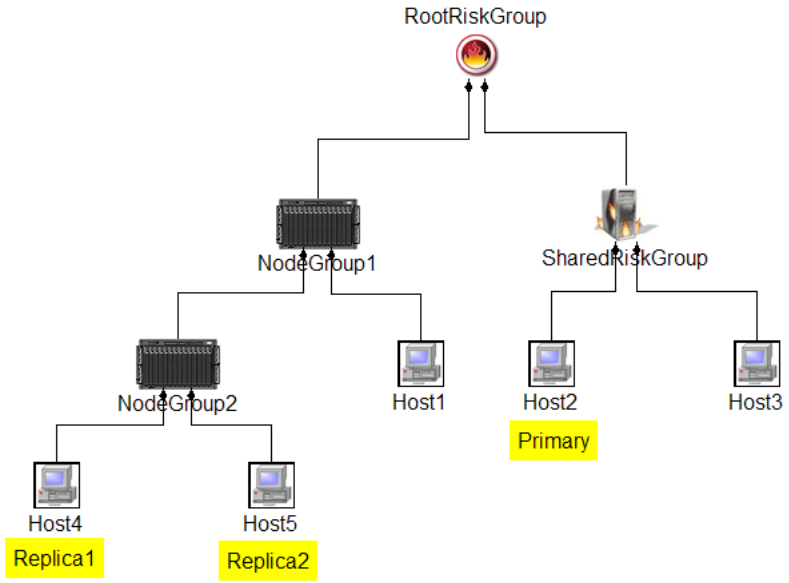

Fig. 4: Shared Risk Group Hierarchy Modeling in CQML

The Figure 4 shows a model of the Shared Risk Group hierarchy. Hosts 1 to 5 are part of a domain and are contained under a common "RootRiskGroup" at the top. A RootRiskGroup represents comparatively larger structures such as a ship or an entire building. All the hosts in the domain share a common risk of failure of the largest composing structure represented by a RootRiskGroup. We limit the scope of our dependability solution at that level. The RootRiskGroup is further divided in to smaller units of Shared Risk Groups as shown in the figure. For example, Host1, Host4 and Host5

share a common risk of a failure of the NodeGroup1 but failure of NodeGroup2 that consists of Host4 and Host 5 does not affect Host1.
The distance between hosts is simply computed as the number of tree edges between two hosts. For example, the distance between the Host2 and Host3 is 2. Similarly the distance between the Host2 and the Host4 or Host5 is 5. Based on such a Shared Risk Group hierarchy, deployment decisions are taken to maximize the distance between the primary component and its replicas as shown in the Figure 4

3.3 Improving Availability via Effective Replica Placement

Requirement (3) in Section 2.2 states that the dependability solution for enterprise DRE systems must minimize the risk of simultaneous failures of replicated functionality. This requires effective replica placement algorithms, where replication is provided for protection units that are modeled as failover units described in Section 3.2.

MDDPro uses GME's plugin capabilities to add model interpreters. One such model interpreter addresses the replica placement problem. The placement model interpreter provides a strategizable framework that can use different constraint-based algorithms to determine an effective replica placement plan to minimize the co-failure probability of the system as a whole.

Formulation of replica placement problem instance in MDDPro. In one instantiation of the formulation of the replica placement problem within our strategizable model interpreter, we use mathematical vectors to represent the distance of the replicas from the primary component. If the primary component has N replicas, then we form N orthogonal vectors, where each vector represents the distance from the primary component node in terms of hops captured in the shared risk group hierarchy. The magnitude of the resultant vector of the N orthogonal vectors is used to compare different deployment configurations and to find the one that satisfies the constraints.

In this formulation of the placement problem algorithm, we have taken care to avoid generation of some obviously undesirable deployment configurations of the system. For example, it does not allow deployment configuration where all the replicas of a component are located in the same host. This is obviously undesirable in dependable enterprise DRE systems because placing multiple replicas in the same host increases the risk of simultaneous failure of replicas.

Prototype heuristic algorithm using the distance metric. The prototype placement algorithm that we have developed maximizes the distance of the replicas from the primary replica but the pair-wise distance between replicas themselves can be small. In other words, the replicas themselves can group together in closely located hosts that are farthest from the primary host. Such a deployment configuration is skewed and undesirable. To alleviate the problem we apply a penalty function to the resultant magnitude of the vector. The penalty function gives more precedence to uniform deployments than highly skewed deployments. The penalty function that we have used is a simple standard deviation of the distances of individual replicas from the primary component. We can

generate better configurations by penalizing highly skewed deployment configurations heavily compared to the more uniform deployment configurations.

For example, consider two resultant vectors $v1\{4, 4, 4\}$ and $v2\{1, 1, 8\}$ having 3 dimensions. Although the magnitude of v2 is much greater than v1, the deployment configuration captured in v1 is more desirable than v2 because the replicas are spread across more uniformly around the primary unlike v2. The heuristic algorithm for the prototype implementation of the deployment algorithm is illustrated in Listing 1.

```
1. Compute the distance from each of the replicas to the primary for a placement.

2. Record each distance as a vector, where all vectors are orthogonal.

3. Add the vectors to obtain a resultant.

4. Compute the magnitude of the resultant.

5. Use the resultant in all comparisons (either among placements or against a threshold)

6. Apply a penalty function to the composite distance (e.g. pairwise replica distance)
```

Listing 1: Replica Placement Heuristics

3.4 Automated Dependability Provisioning via Generative Programming

The model interpreters and generative tools in MDDPro use the dependability requirements captured in the models for synthesizing metadata used to provision dependability for enterprise DRE systems. In order to realize such an automation in the provisioning process several artifacts of dependability must be addressed: (a) the designer of the dependable system has to annotate the desired degree of replication of the protected components in the model, (b) the generative tools have to process the replication requirements and produce deployment metadata that reflects the number of physical software components that will actually be deployed but not necessarily be represented in the model, (c) derive the complex connection topology interconnecting the generated components, which is dictated by the degree and style of replication of the primary component as well as replication requirements of the components it interacts with, and (d) generating the fault-tolerance infrastructure components that produce a periodic heartbeat as well as monitor the liveness of the replicated components.

Deployment metadata generation framework As noted in Section 3.1, the real-time component middleware platforms used to host the enterprise DRE

systems use standardized XML-based metadata descriptors to describe the deployment plans of the entire system, which the runtime system uses to actually deploy the different components of the system. Our challenge involved enhancing the metadata descriptors to include dependability provisioning decisions. For this goal to realize, MDDPro's generative capabilities had to be integrated with the existing generators available in CQML without obtrusive changes to existing capabilities. This approach ensures that generators for QoS issues beyond dependability, such as security, can seamlessly be integrated with CQML.

To address these concerns, we have developed an extensible framework called *The Deployment Plan Framework* that allows augmentation of metadata generation "on-the-fly" as it is being generated. The framework exposes a fixed set of hooks to be filled in by the developer of the existing and any new CQML model interpreters including the MDDPro model interpreters. The main job of the deployment framework is to generate the standardized metadata describing the components, their implementations, their inter-connections and so on. Additionally, it invokes predefined hook methods implemented by different QoS model interpreters of CQML. The MDDPro interpreter implements a subset of a large set of different possible hook methods. The hook methods "inject" autogenerated standardized metadata in response to the availability requirements captured in the model. The metadata generated on-the-fly blends into the other standardized metadata.

This architecture allows large scale reuse of earlier code base that deals with the basic structure and composition capabilities of PICML/CQML. The developer producing QoS enhancements to the existing modeling capabilities of CQML need not be concerned with the other complexity of the framework and the format of the standardized descriptors, but simply add/modify the metadata for the QoS dimension they are addressing. Our MDDPro model interpreter exploits these capabilities of the Deployment Plan Framework to "inject" three different kinds of metadata.

1. **Replica component instances** of the primary protected component depending upon replication degree annotated in the model. For example, if replication degree of an FOU is 3, then two replicas of the primary FOU are created. Thus, two replicas of each component in the FOU are effectively added by the interpreter.
2. **Component connection metadata** is injected based on the replication style and degree of replication. The incoming connections to the protected components are marked with special annotations so that the run-time system can use suitable implementations to realize them. One such possible annotation is IOGR, i.e. Interoperable Object Group Reference. IOGR is a part of the FT-CORBA [2] standard.
3. **Deployment metadata** is the assignment of components to computing resources available in the system. This metadata includes information for all the primary protected components, their replicas and the dependability infrastructure components (e.g. Heartbeat components).

Handling complex connections As shown in Figure 5, shows the effect of the replication style and the degree of replication on the complexity of the connection establishment. In an unprotected system, the Processor component and the Planner component have exactly one connection between them. The Figure 5 captures the multiplicative increase in the number of connections when both, the Processor component and the Planner component, are protected using active replication. Each Processor component, primary as well as its replica has to make three connections to each member of the Planner replica group because the degree of replication of the Planner fail over unit (FOU) is three. In general, if the source component of the connection is replicated M times and the destination component is replicated N times then the number of connections grow by a factor of M x N.

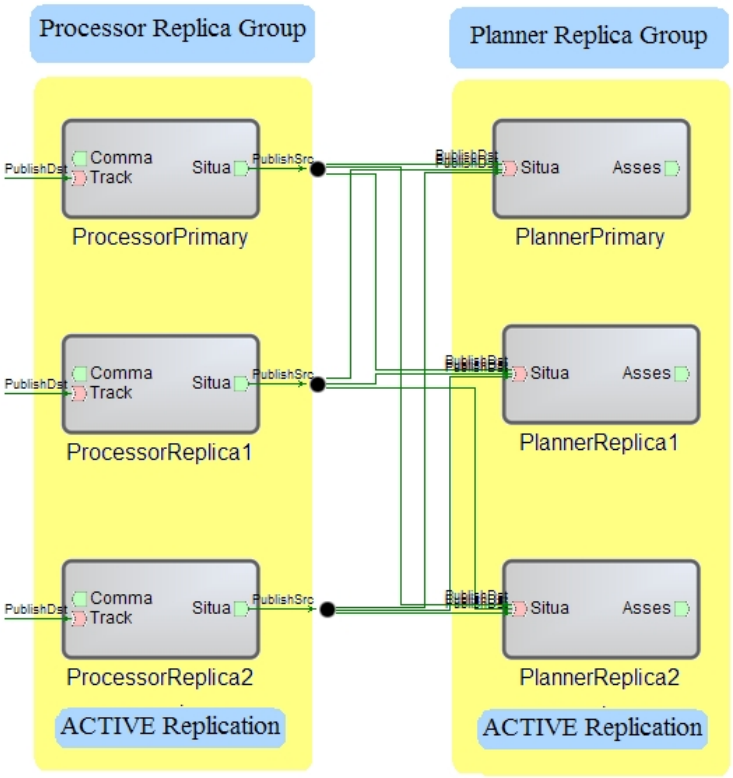

Fig. 5: Complexity of connection generation

Note that the diagram only indicates the necessary number of connections the middleware has to establish when components are deployed. These connection may or may not actually be used to send requests across because it really

depends upon where request/reply suppression is in place. Nevertheless, the component container has to prepare for any unforeseen failures and has to establish connections *apriori* in order to avoid the latency of connection establishment later when failures occur. The model interpreter that we have developed completely hides away the complexity of modeling the component replica instances and the connections between them.

Automatic generation of liveness monitoring infrastructure The model interpreter also generates the infrastructure components necessary for liveness monitoring of the protected components. It uses the availability requirements in the models to generate supporting run-time components to realize ready-to-deploy, robust, and fault-tolerant enterprise DRE systems. This includes generating, configuring, and deploying the status monitoring and fault recovery components without the need for the application developer having to model/develop them explicitly.

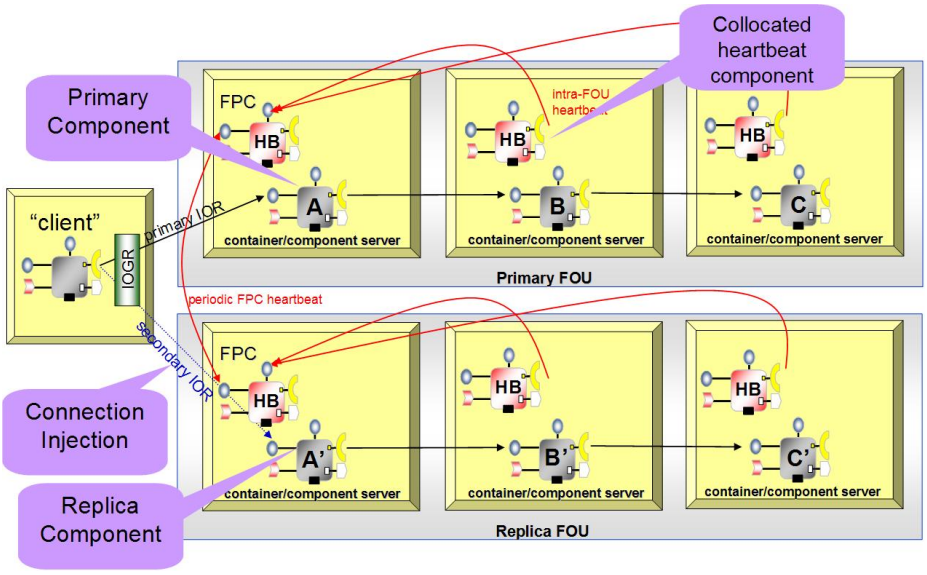

Fig. 6: Generated Deployment of Dependability Infrastructure Elements

The generated architecture shown in Figure 6 has two important components: the heartbeat (HB) component and the Fault Protection Center component (FPC). The purpose of the HB components is to send a periodic heartbeat beacon to the FPC or respond to the periodic liveness poll request received from the FPC. The FPC is the central controlling component that ensures the liveness of the protected components using either pull or push model of the heartbeat beacon. The HB components are collocated with the protected components. The

underlying assumption is that the HB component and the protected component would fail simultaneously in the face of a failure. The central FPC component is also replicated to avoid single point of failure. Multiple copies of the FPC components send heartbeat beacons among themselves to ensure that FPC themselves are alive and are doing continuous liveness monitoring of the system.

As shown in Figure 6, every protected component has its own collocated HB component and there is one FPC for every FOU. All the HB components belonging to one FOU send heartbeat to its corresponding FPC. Multiple simultaneously active FOUs have equal number of FPCs, which communicate with themselves to prevent single point of failure.

The heartbeat frequency at which the liveness indications are sent between HBs and FPCs is configurable in the model. The advantage of this architecture is that the infrastructure components for liveness monitoring can be auto-generated using generative technologies. The necessary deployment metadata required to collocate the HB components with their respective protected components and to establish the connections between HB components and the FPC components is auto-generated by the model interpreter from the requirements. Moreover, the metadata that captures the configuration of HB components such as push/pull model and heartbeat frequency is auto-generated for every HB component.

4 Related Work

Although there has been substantial research in dependability mechanisms and algorithms over the past several decades, applying modeling and generative techniques to automate dependability provisioning has recently caught researchers' attention. In this section we compare our work on model-driven engineering of dependability with related research.

The CORRECT project [12] describes a project that is looking at applying step-wise refinement and OMG's Model Driven Architecture [13] to automatically generate Java code used in a fault tolerant distributed system. The project uses UML to describe the software architecture in both a platform-independent and platform-specific form. Model-to-model transformations are used to incrementally enrich the models with platform-specific artifacts until the Java skeleton code is generated. MDDPro, on the other hand, is designed to automatically generate the complete source code (not just the skeletons) for the component liveness monitoring infrastructure that detects exceptional conditions.

[14] focuses on a research that provides platform-independent means to support reliability design following the principles of a model driven architecture and approach. The research aims to systematically address dependability concerns from the early to the late stages of software development by expressing dependability architectures using profiles. Design profiles are mapped to deployment domains, where the reliability configurations of how the components communicate and are distributed is explained. Unlike [14], MDDPro uses an extensible way to automatically generate platform specific metadata and programming language artifacts that realize parts of the dependability provisioning solution.

[15] focuses on a research that uses UML to model application architectures for composite web services. The UML representation is based on BPEL, and extensions are added to characterize the fault behavior of the elements comprising the web services. Model transformations are used to map the UML models to Block Diagrams, Fault Trees and Markov models to analyze the dependability characteristics of the composite web services. On the other hand, our approach in MDDPro enhances the productivity of the system developers rather than system dependability analysts.

Although our research on MDDPro has similar goals, we use the concept of domain-specific modeling languages, which provides more richer and semantically powerful modeling concepts than the general-purpose modeling elements provided by UML. Additionally our framework allows plugging in multiple different model interpreters that can synthesize metadata for multiple different middleware platforms provide deployment planning.

5 Conclusions

This paper describes how model driven engineering (MDE) can be used to simplify and automate dependability provisioning in enterprise distributed real-time and embedded (DRE) systems. We describe the capabilities of the MDDPro (Model Driven Dependability Provisioning) MDE tool which we have built as part of the CoSMIC tool suite. Our work is suitable for component-oriented systems that have multiple different quality of service requirements and which are deployed and configured via declarative mechanisms. Both these traits are common to systems that use the service oriented architecture. In the remainder of this section we describe the lessons we learned in this effort and our future work in this realm.

Lessons Learned and Future Work

Capturing availability requirements in terms of degree of replication, replication style at the modeling time and generating component infrastructure components increases productivity to a great extent but many unresolved challenges still remain.

- **Availability model analysis** is useful to determine the effect of the availability requirements on other QoS aspects of the system. Our prototype implementation of MDDPro is simplistic because it neglects the effects on system resource consumption due to replication. Unconstrained increase in the degree of replication of the protected components in the system may result in excessive resource consumption and may adversely affect other QoS guarantees of the system such as timeliness and CPU load. An analysis technique needs to be in place that would help the system designers take correct decisions about the system availability without adversely affecting the resource consumption and other QoS characteristics of the system.

- **Run-time adaptation** of the fault-tolerance infrastructure as well as the replicated application components is highly desirable in enterprise DRE systems because these systems usually exhibit modal behavior. System functionality as well QoS priorities may change as the mode of operation of the system changes. Our approach to the availability modeling is static in nature and depends on the availability of the target domain information and their associations with each other in terms of co-failure probability. Although the placement model interpreter does take deployment decisions at design time using a strategizable constraint-solver framework, it does not make the system adaptive at run-time. Runtime monitoring subsystems such as RACE can be used to implement a general purpose resource constraint-solver framework at runtime, not unlike the one we have in our design-time placement model interpreter. Such a framework would make intelligent (re)deployment decisions based on changing environment (failures, resource consumption) and modes of the eDRE systems.
- **Ensuring state consistency** across replicas of components or FOUs in a general is a challenge. Our availability model abstracts away the details of the fault monitoring part of the FT subsystem and generates component based infrastructure automatically for precisely doing that. However, state synchronization and ensuring state consistency across replicated components of the system is a hard problem. The primary challenges in this space are capturing and provisioning a variety of state synchronization mechanisms because different component developers may implement different mechanisms as they see fit. Several different ways of ensuring state synchronization are used, for example, central repository/database-based approach, transmission of periodic state updates using point-to-point communication or multicast communication. Modeling the topology transmission of state update messages is also important in case of non repository-based techniques because the runtime failover critically depends on the order in which replica components receive state updates.

All artifacts described in this paper are available in open source from the CoSMIC web site (`www.dre.vanderbilt.edu/cosmic`).

References

1. Object Management Group: Lightweight CCM FTF Convenience Document. ptc/04-06-10 edn. (June 2004)
2. Object Management Group: Fault Tolerant CORBA Specification. OMG Document orbos/99-12-08 edn. (December 1999)
3. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer **39**(2) (2006) 25–31
4. Pascal Felber and Priya Narasimhan: Experiences, Approaches and Challenges in building Fault-tolerant CORBA Systems. Transactions of Computers **54**(5) (May 2004) 497–511
5. Priya Narasimhan and Tudor Dumitras and Aaron M. Paulos and Soila M. Pertet and Charlie F. Reverte and Joseph G. Slember and Deepti Srivastava: MEAD:

support for Real-Time Fault-Tolerant CORBA. Concurrency - Practice and Experience **17**(12) (2005) 1527–1545
6. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading, Massachusetts (2000)
7. Gokhale, A., Schmidt, D.C., Natarajan, B., Gray, J., Wang, N.: Model Driven Middleware. In Mahmoud, Q., ed.: Middleware for Communications. Wiley and Sons, New York (2004) 163–187
8. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-Integrated Development of Embedded Software. Proceedings of the IEEE **91**(1) (January 2003) 145–164
9. Ledeczi, A., Bakay, A., Maroti, M., Volgysei, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. IEEE Computer (November 2001) 44–51
10. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. Elsevier Journal of Computer and System Sciences (2006) 171–185
11. Object Management Group: Deployment and Configuration Adopted Submission. OMG Document mars/03-05-08 edn. (July 2003)
12. Capozucca, A., Gallina, B., Guelfi, N., Pelliccione, P., Romanovsky, A.: CORRECT - Developing Fault-Tolerant Distributed Systems. ERCIM News **64**(1) (2006)
13. Object Management Group: Model Driven Architecture (MDA). OMG Document ormsc/2001-07-01 edn. (July 2001)
14. G.Rodrigues: A Model Driven Approach for Software Systems Reliability. In: In the proceedings of the 26^{th} ICSE/Doctoral Symposium, May 2004 - Edinburgh, Scotland, ACM Press (May 2004)
15. Zarras, A., Vassiliadis, P., Issarny, V.: Model-Driven Dependability Analysis of Web Services. In: Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04), Agia Napa, Cyprus (October 2004)