Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Development: a Case Study

Jules White, Douglas Schmidt, Aniruddha Gokhale
{jules, schmidt, gokhale}@dre.vanderbilt.edu
Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, USA

Abstract

Autonomic computer systems aim to reduce the configuration, operational, and maintenance costs of distributed applications by enabling them to self-manage, self-heal, self-optimize, self-configure, and self-protect. This paper provides two contributions to the model-driven development (MDD) of autonomic computing systems using Enterprise Java Beans (EJBs). First, we describe the structure and functionality of an MDD tool that formally captures the design of EJB applications, their quality of service (QoS) requirements, and the autonomic properties applied to the EJBs to support the rapid development of autonomic EJB applications via code generation, automatic checking of model correctness, and visualization of complex QoS and autonomic properties. Second, the paper describes how MDD tools can generate code to plug EJBs into a Java component framework that provides an autonomic structure to monitor, configure, and execute EJBs and their adaptation strategies at run-time. We present a case study that evaluates how these tools and frameworks work to reduce the complexity of developing autonomic applications.

Keywords: Autonomic Computing, Component Middleware, Model-driven Development.

1 Introduction

Autonomic computing challenges. Developing and maintaining enterprise applications is hard, due in part to their complexity and the impact of human operator error, which have shown to be a significant contributor to distributed system repair and down time [2]. The aim of autonomic computing is to create distributed applications that have the ability to self-manage, self-heal, self-optimize, self-configure, and self-protect [1], thereby reducing human interaction with the system to minimize down-time from operator error. Although the benefits of autonomic computing are significant [1], the pressures of limited development timeframes and inherent/accidental complexities of large-scale software development have discouraged the integration of sophisticated autonomic computing functionality into distributed applications. Some enterprise application platforms offer limited autonomic features, such as such as Enterprise Java Bean (EJB) [3] application servers clustering capabilities, though they tend to have large development teams and long development cycles.

A key challenge limiting the use of autonomic features in enterprise applications is the lack of design tools and frameworks that can (1) alleviate the complexities stemming from the use of *ad hoc* methods and (2) generate code that is correct-by-construction. Some infrastructure does exist, such as IBM's Autonomic Computing Toolkit [4], which focuses on system-level logging and management. System-level autonomic toolkits are inadequate, however, for fine-grained autonomic capabilities, which are essential to fix problems early before an entire application must be restarted.

To address the limitations with system-level autonomic toolkits, *component-level* autonomic frameworks are needed to reduce the effort of developing autonomic applications. Component-level autonomic properties support fine-grained healing, optimization, configuration, monitoring, and protection than system-level toolkits. For example, a mission-critical command and control system for emergency responders should be able to shutdown/restart application component logic selectively as it fails, rather than shutdown/restart the entire application. With existing autonomic infrastructure based on the system-level , the failure of a key component triggers a restart of the entire application [5]. In contrast, a component-level autonomic framework could provide mechanisms to restart only the point of failure.

Creating applications with either system or component-level autonomic frameworks requires moving large amounts of state data, analysis data, actions plans, and execution commands between components. These types of applications also require careful weaving of monitoring, analysis, planning, and execution logic into the functional components of the system. Analysis of the autonomic aspects of the application, such as checking whether the right state is being monitored by the right components, is a tedious and error-prone process.

Simplifying autonomic system development via MDD techniques. *Model-driven development* (MDD) [6] tools are a promising means of reducing the cost associated with these activities. Models of autonomic systems developed with MDD tools can be constructed and checked for correctness (semi-)automatically to ensure that application designs meet autonomic requirements. Tools can also be used to generate the various capabilities to move data, coordinate actions, and perform other autonomic functions.

To address the need for component-level autonomic computing – and to avoid *ad hoc* techniques that manu-

ally imbue autonomic qualities into distributed applications – we have created the *J3 Process,* which is an open-source MDD environment that supports the rapid design and implementation of autonomic applications. J3 consists of several MDD tools and autonomic computing frameworks, including (1) *J2EEML*, which captures the design of EJB applications, their quality of service (QoS) [6] requirements, and the autonomic adaptation strategies of their EJBs via a domain-specific modeling language (DSML) [7], (2) *Jadapt*, which is a J2EEML model interpreter that analyzes the QoS and autonomic properties of J2EEML models, and (3) *JFense*, which is an autonomic framework for monitoring, configuring, and resetting individual EJBs [8].

This paper describes the structure and functionality of J2EEML and shows how it simplifies the development of autonomic systems by providing notations and abstractions that are aligned with autonomic computing, QoS, and EJB terminology, rather than low-level features of operating systems, middleware platforms, and third-generation programming languages. We also describe how (1) Jadapt generates EJB and Java code from J2EEML models to ensure that autonomic applications meet their specifications and to reduce implementation time and (2) JFense provides a set of reusable autonomic components that allow developers to plug-in EJB applications and focus on autonomic logic, rather than the glue for constructing the autonomic system. Finally, we evaluate how the J3 Process reduces the complexity of developing an autonomic EJB application used as a case study to evaluate our MDD tools and processes.

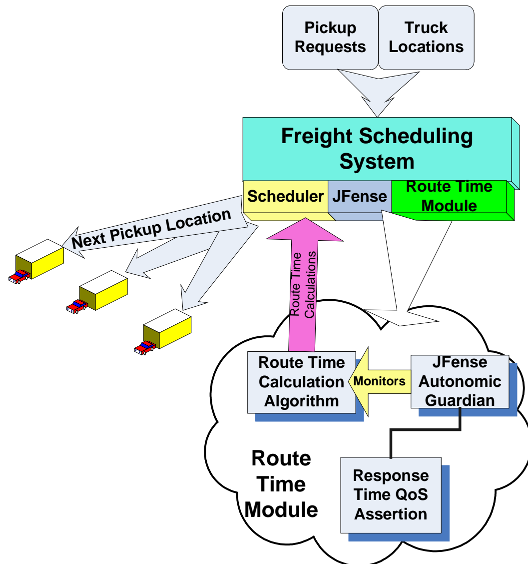

Figure 1: A Multi-Layered Autonomic Architecture for Scheduling Highway Freight Shipments

Our case study centers on an EJB-based system that schedules highway freight shipments using the multi-layered autonomic architecture shown in Figure 1. The system has a list of freight shipments that it must schedule. It uses a constraint-optimization engine to find a cost effective assignment of drivers and trucks to shipments.

A central component in Figure 1 is the *Route Time Module* (*RTM*), which determines the route time from a truck's current location to a shipment start or end point. The *RTM* uses a geo-database and the GPS coordinates from the truck to perform the calculation. This module is critical to the proper operation of the optimization engine. A heavy load is placed on the *RTM*, so it is crucial that it maintains its *QoS assertions*, such as maintaining a maximum response time for the *RTM* of 100 milliseconds. QoS assertions are properties that the system can introspectively measure about itself to determine whether the measured value for the property is beneficial to the system. These measured *QoS goals* allow the system to decide whether it is in a good state and predict whether it will continue to remain in a good state.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes the MDD J3 Process for developing autonomic EJB applications; Section 3 gives an overview of J2EEML and describes key challenges we faced when developing it; Section 4 quantifies the reduction in manual effort achieved by applying the J3 Process on our highway freight shipment case study; Section 5 compares our work with related research; and Section 6 presents concluding remarks.

2 The J3 Process for Autonomic System Development

The *J3 Process* contains the following MDD tools and component middleware frameworks that address the challenges of developing autonomic EJB applications:

- **J2EEML**, which is a DSML-based MDD tool tailored for designing autonomic EJB applications that uses visual representations to model domain-specific abstractions. J2EEML provides a formal mapping from QoS requirements to application components.
- **Jadapt**, which is an MDD tool that produces many artifacts required to implement autonomic EJB applications modeled in J2EEML. Jadapt generates code that meets the J2EEML specifications and also reduces the amount of code that application developers must write manually.
- **JFense**, which is an autonomic framework that provides components for monitoring, analysis, planning, and execution. Developers can use these components to avoid writing custom autonomic frameworks. JFense can be configured to meet the autonomic requirements of a variety of EJB applications.

This section focuses on the design and function of J2EEML and illustrates how it can be used to create structural models of EJB applications.

2.1 Overview of J2EEML

J2EEML is a DSML that enables EJB developers to construct models that incorporate autonomic and QoS

concepts as first-class entities. J2EEML itself is developed using the *Generic Modeling Environment* (GME) [9], which is a general-purpose MDD environment that we use to simplify the creation of *metamodels* that characterize the roles and relationships in the autonomic computing domain, and *model interpreters* that generate many artifacts required to implement autonomic EJB applications. J2EEML captures the relationship between QoS assertions and application components to address key design challenges of developing autonomic applications. For example, J2EEML helps developers understand which components to monitor in their EJB applications by enabling them to visualize and analyze the relationships between components and QoS assertions.

Developers use J2EEML to capture the design of autonomic systems and the mapping of components to QoS assertions in four phases: (1) they create a structural model of the EJBs composing an autonomic system, (2) they create models of the QoS properties that the system is attempting to maintain, (3) they map these QoS properties to the specific beans within the system that the properties are measured from, and (4) they design courses of action to take when the desired QoS properties are not maintained. This modeling process captures the structure of the system, how the QoS properties are related to the structure, and what adaptation should occur if a QoS property is not within an acceptable range.

2.2 Modeling EJB Structures with J2EEML

The first piece of a J2EEML model is its *EJB structural model*, which describes the components of the system that will be managed autonomically. This model defines the beans that compose the system and captures the EJB specifics of each bean, including JNDI names, transactional requirements, security requirements, package names, descriptions, remote and local interface composition, and bean-to-bean interactions. An EJB structural model is constructed via the following steps:

1. Each session bean is added to the model by dragging and dropping session bean atoms into the J2EEML model. Developers then provide the Java Naming and Directory Interface (JNDI) name of the bean, its description, and its state type (i.e., stateful or stateless).
2. For each session bean, a model is constructed of the business methods and creators supported by the bean by dragging and dropping method and creator atoms. Each method and creator atom has properties for setting the visibility, whether it is part of the local interface, remote interface or both, and the transactional requirements, the security role requirements, and the input/output. Figure 2 shows a model of the remote interface composition of the *TruckStatusModule* from the case study described in Section 1. The model contains methods for modifying the truck's schedule and their transactional/security requirements.

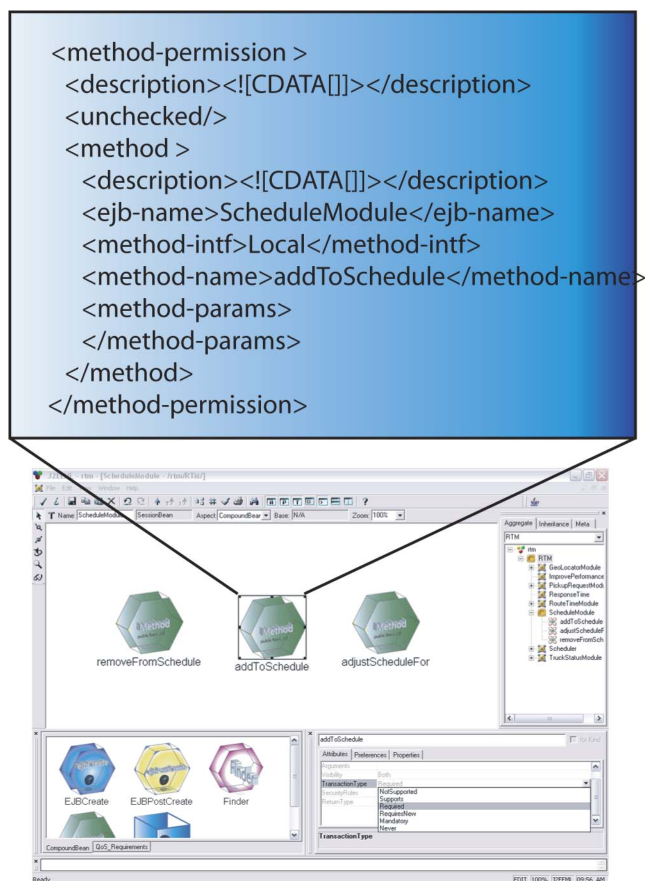

Figure 2: J2EEML Remote Interface Composition Model for the *TruckStatusModule*

3. Entity beans are dragged and dropped into the model to construct the data access layer. These beans are provided a JNDI name/description and properties indicating if they use container managed persistence (CMP) or bean managed persistence (BMP).
4. Persistent fields, methods, and finders are dragged and dropped into the entity beans. Each persistent field has properties for setting visibility, type, whether it is part f the primary key, and its access type (i.e., read-only or read-write).
5. Relationship roles are dragged and dropped into the entity beans and connected to persistent fields. These relationship roles can be connected to other relationship roles to indicate entity bean relationships.
6. Connections are made between beans to indicate bean-to-bean interactions. Capturing these interactions allows Jadapt to later generate the required JNDI lookup code for a bean to obtain a reference to another bean.

After these six steps have been completed, the J2EEML model contains enough information to represent the composition of the EJBs.

Figure 3 shows a J2EEML structural model of the highway freight scheduling system. In this figure, each bean within the freight scheduling system has been mod-

eled via J2EEML. Interactions between the beans are also modeled, thereby allowing developers to understand which beans interact with one another. Figure 3 also illustrates snippets of the XML deployment descriptor and Java class generated for the *Scheduler*.

To support decomposition of complex architectures into smaller pieces, J2EEML allows EJB structural models to contain child EJB models. Beans within the these children show up as ports that can receive connections from the parent solution. This design allows developers to decompose models into manageable pieces and enables different developers to encapsulate their designs.

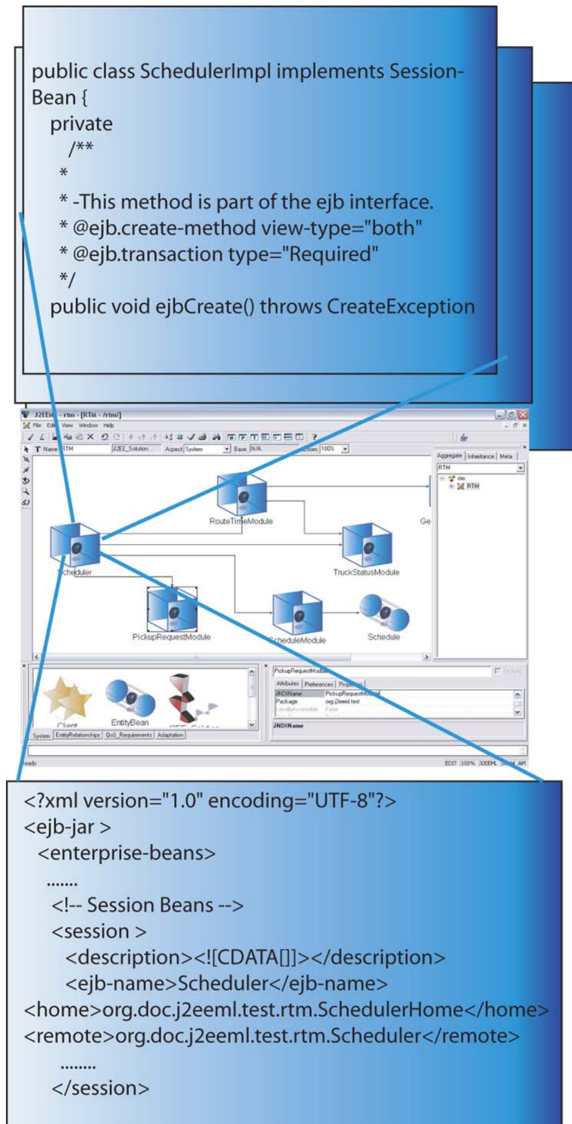

Figure 3: J2EEML Structural Model Showing Bean-to-Bean Interactions

For our highway freight scheduling example, we constructed a structural model of each bean required for the *Route Time Module*, constraint-optimization engine, truck status system, and incoming pickup request system, as shown in Figure 3. The model also includes information on the entity beans used to access the *truck location* and *pickup request* databases.

Using J2EEML provides several advantages in the design phase, including (1) visualization of beans and their interactions, component security requirements, system transactional requirements, and interactions between beans, (2) enforcement of EJB best practices, such as the Session Façade pattern [10], which hides Entity beans from clients through Session beans , and (3) model correctness checking, including checks for proper JNDI naming. J2EEML's visualization benefits significantly decreased the difficulty of understanding system structure and interactions. The correctness checking and enforcement of best design practices facilitated rapid creation of both a correct-by-construction and well-designed solution.

3 Designing J2EEML to Address Key Concerns of Autonomic Computing

Autonomic applications require four elements to achieve their assertions: *monitoring, analysis, planning, and execution* [1]. These elements form a *controller* that observes and adapts the application to maintain its assertions. This section describes how the monitoring, analysis, and planning aspects of autonomic systems present unique challenges when designing and building the J2EEML and shows how we addressed each challenge. To focus the discussion, we use the *Route Time Module* (*RTM*) shown in Figure 1 as a case study to illustrate key design challenges associated with autonomic systems.

3.1 Monitoring

Monitoring is the phase in autonomic systems where applications observe their own state. Since this state information is used in later phases to control system behaviors it is crucial that the right information be collected at the right times without adversely impact system functionality and QoS. The following are key design challenges faced when developing the monitoring aspects of autonomic systems:

Challenge 3.1.1: Providing the ability to specify the large range of data that can be monitored by the system. Developers of autonomic systems must address how to self-monitor key data, e.g., by capturing CPU and memory utilization, exceptions thrown by the applacation, or error messages in a log. The model for specifying what information to capture from the system must be flexible and support a range of data types. The model must also be extensible and support unforeseen future data types that might be needed later.

A core concept behind J2EEML is that an autonomic EJB application can measure properties of its current state introspectively and determine if the property values indicate the application is in a beneficial state. J2EEML models the properties it measures via *QoS assertions*,

which determine which properties an autonomic system can measure about itself introspectively and analyze to determine if the properties are in an acceptable assertion range. Each assertion provides properties for setting its name and description. Developers can drag and drop these assertions into J2EEML models.

The J2EEML QoS assertions model is critical for understanding an autonomic system's QoS properties, how they can be measured, what their values should be, and how degradations in them can be corrected. Understanding QoS assertions is also crucial to designing the structural architecture of EJB applications and understanding how they meet those assertions. Capturing and mapping QoS requirements to the appropriate structural architecture have traditionally used natural language descriptions, such as "the service must support 1,000 simultaneous users with a good response time." Due to the lack of an unambiguous formal notation, such descriptions are prone to different interpretations, which result in architectures that do not meet the QoS requirements. Choosing an EJB architecture that best fits the QoS requirements can be complex and error-prone since specification ambiguity and hidden architectural tradeoffs make it hard to choose the appropriate design.

For example, deciding whether to use remote interfaces for a J2EE implementation of a service can have a substantial impact on end-to-end system QoS. Remote interfaces allow distribution of beans across servers, which can increase scalability. Distribution can also increase latency, however, since requests must travel across a network or virtual machine boundaries.

With the *RTM* in our case study, one QoS assertion is the average response time. This QoS assertion states that the system will measure all requests to the *RTM* and track the average time required to service each request. If the calculated average response time exceeds 50 milliseconds, the assertion is false, indicating that the *RTM* is taking too long to respond, otherwise the assertion is true, indicating that the *RTM* is responding properly.

Figure 4 illustrates a J2EEML model of the scheduling system and the association of the *RTM* to the *ResponseTime* QoS property. This model shows J2EEML's ability to model QoS properties as aspects [17] that are applied to a component. When the model is interpreted and the Java implementation generated, the association between the *RTM* and *ResponseTime* assertion will lead to the appropriate monitoring code being generated in the *RTM*'s implementing class.

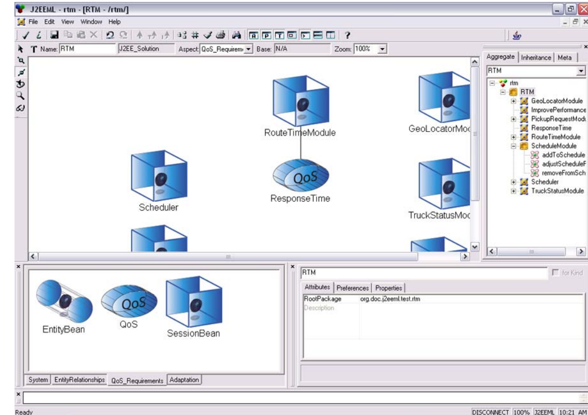

Figure 4: J2EEML Model Associating the *ResponseTime* QoS Assertion with the *RouteTimeModule*

Challenge 3.1.2: Building a system to specify where monitoring logic should reside in the system. The decision of what to monitor directly affects where the monitoring logic will reside. To monitor a log for errors, the logic could be at any level of the application, such as a central control level. For observing exceptions or the load on a specific subcomponent of the application, the monitoring logic must be embedded more deeply. In particular, developers must position the monitoring capability precisely so that it is close enough to capture the needed information, but not so deeply entangled in the application logic that it adversely affects performance and separation of concerns.

In our freight scheduling case study, we must ensure separation of concerns in the application design and find an efficient means of monitoring. The monitoring logic for the *RTM*, however, should not be entangled with the route time calculation logic. Moreover, the time to monitor each request should be insignificant compared to the time to fulfill each route request.

After the structural and assertion models are completed, developers can use J2EEML to map QoS assertions to EJBs in the structural model. This mapping documents which QoS assertions should be applied to each component. It also indicates where monitoring, analysis, and adaptation should occur for an autonomic system to maintain those assertions. For example, to determine the average response time of the *RTM*, calls to the *RTM*s route time calculation method must be intercepted to calculate their servicing time. The relationship between the *RTM* bean and average response time assertion in the model indicates that the *RTM* bean must be able to monitor its route time calculation requests.

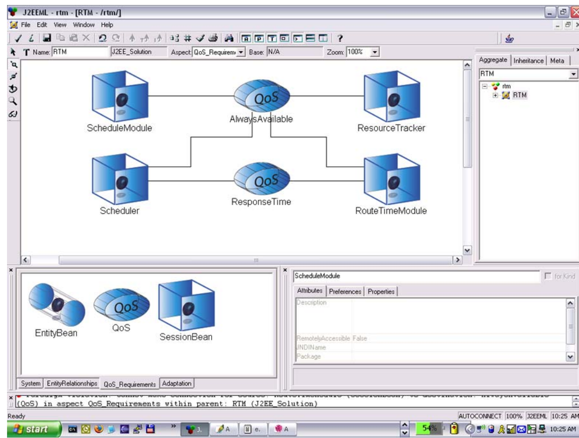

Figure 5: J2EEML Mapping of QoS Assertions to EJBs

J2EEML supports aspect-oriented modeling [11] of QoS assertions, i.e., each QoS assertion in J2EEML that crosscuts component boundaries can be associated with multiple EJBs. For example, maintaining a maximum response time of 100 milliseconds is crucial for both the *RTM* and the *Scheduler* bean. Connecting multiple components to a QoS assertion, rather than creating a copy for each component, produces clearer models. It also clearly shows the connections between components that share common QoS assertions. Figure 5 shows a mapping from QoS assertions to EJBs. Both the *RTM* and the *Scheduler* in this figure are associated with the QoS assertions *ResponseTime* and *AlwaysAvailable*. The *ResourceTracker* and *ShipmentSchedule* components also share the *AlwaysAvailable* QoS assertion in the model.

Components can have multiple QoS assertion associations, which J2EEML supports by either creating a single assertion for the component that contains sub-assertions or by connecting multiple QoS assertions to the component. If the combination of assertions produces a meaningful abstraction, hierarchical composition is preferred. For example, the *RTM* is associated with a QoS assertion called "AlwaysAvailable" constructed from the sub-assertions "No Exceptions Thrown" and "Never Returns Null." Combining "Minimum Response Time" and "No Exceptions Thrown," however, would not produce a meaningful higher-level abstraction, so the multiple connection method is preferred in this case.

3.2 Analysis

Analysis is the phase in autonomic systems that takes state information acquired by monitoring and reasons about whether certain conditions have been met. For example, analysis can determine if an application is maintaining its QoS requirements. The analysis aspects of an autonomic system can be (1) centralized and executed on the entire system state or (2) distributed and concerned with small discrete sets of the state. The following are key challenges faced when developing an autonomic analysis engine:

Challenges 3.2.1: Building a model to facilitate choosing the type of analysis engine and Challenge 3.2.2: Building a model to facilitate choosing how the engine should be decomposed and/or distributed. To choose a distributed vs. monolithic analysis engine, the tradeoffs of each must be understood. Concentration of analysis logic into a single monolithic engine enables more complex calculations. However, for simple calculations, such as the average response time of the *RTM* component, a monolithic engine requires more overhead to store/retrieve state information for individual components than an analysis engine dedicated to a single component. A monolithic analysis engine also provides a central point of failure. A key design question is thus where analysis should be done and at what granularity.

A model to facilitate choosing the appropriate type of analysis engine must enable developers to identify what data types are being analyzed, what beneficial information about the system state can be gleaned from this information, and how that beneficial information can most easily be extracted. It is important that the model enable a standard process for examining the required analyses and determining the appropriate engine type.

To create an effective analysis engine, developers must determine the appropriate number of layers. A key issue to consider is whether an application should have a single-layer vs. multi-layered analysis engine. At each layer, the original monitoring design questions are applicable, i.e., what should be monitored and how should it be monitored? A model to enable these decisions must clearly convey the layers composing the system. It also must capture what analysis takes place at each layer and how each layer of analysis relates with other layers.

In the context of our highway freight scheduling system, a key question is whether its autonomic layer analyzes the *RTM*'s response time or whether a layer above the *RTM* should do it. At each layer, the analysis design considerations are important too, e.g., what information the system is looking for in the data, how it finds this information, and how this can be better accomplished by splitting the layer. For example, a developer must consider whether every request to the *RTM* should be monitored to determine if the *RTM* is meeting its minimum response time QoS. Conversely, perhaps only certain types of requests known to be time consuming should be monitored. Another question facing developers is how the *RTM*'s monitoring logic sends data to its analysis engine.

Developers can use J2EEML to design hierarchical QoS assertions to divide-and-conquer complex QoS analyses. A hierarchical QoS assertion is a assertion that is only met if all its child assertions are met. In terms of QoS assertions, this means that all the child QoS assertions must hold for the parent QoS assertion to hold. With respect to the *RTM*, the QoS assertion *GoodResponseTime* only holds if both the child QoS assertions *AverageResponseTime* and *MaximumResponseTime* also hold. This hierarchical composition is illustrated in Fig-

ure 6, where *GoodResponseTime* is an aggregation of several properties of the response time.

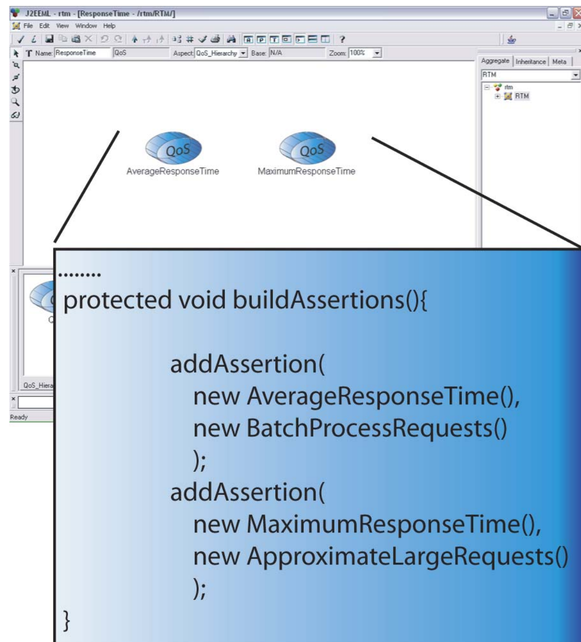

Figure 6: J2EEML Hierarchical Composition of ResponseTime QoS Assertion

Modeling QoS assertions hierarchically enhances developer understanding of what type of analysis engine to choose. A small number of complex QoS assertions that cannot be broken into smaller pieces imply the need for a monolithic analysis engine. A large number of assertions – especially hierarchical QoS assertions – imply the need for a multi-layered analysis engine.

Modeling QoS assertions hierarchically also enhances developer understanding of how to decompose the analysis engine into layers. The hierarchical model of the QoS assertions corresponds directly to the decomposition of the analysis engine into layers. Developers can use J2EEML to first add complex QoS assertions to their models and then determine if the complex assertion can be accomplished by combining the results of several smaller analyses. If so, developers can add these smaller QoS assertions as children of the original QoS assertion to represent the smaller analyses and then apply this iterative process to the new children.

3.3 Planning

Planning is the phase in autonomic systems where applications examine the results of their analysis and decide what actions to take to reach their assertions. For our highway freight scheduling example, this could involve changing the *RTM* to use a less precise but faster algorithm that maintains the minimum response time as demand grows. A typical autonomic application may have hundreds of assertions and planning the correct actions in the face of QoS failures is critical to an autonomic application. The following are key challenges faced when developing an autonomic analysis engine:

Challenge 3.3.1 Designing a means to specify layered adaptation plans. As with monitoring and analysis, planning can be implemented with a layered architecture. A simple, one-layer architecture would monitor, reason, and react to all system events at one level, which works well for macro-level events and actions. This simple approach is less suitable for applications that need more flexible and fine-grained control of their behavior. To increase flexibility and fine-grained control, therefore, more layers can be integrated into the system. Layers distribute intelligence throughout the system and support a divide-and-conquer approach to planning.

After the planning is provisioned into layers, each layer must be assigned a responsibility to react to and recover from QoS failures. In our highway freight scheduling example, one layer might ensure that the *RTM* is always available and the next layer down might ensure that a minimum response time is maintained. Intelligent separation of responsibilities can produce hierarchical chains of command that reduce the complexity of accomplishing the overall assertion. Finding these well-proportioned divisions of labor is hard.

J2EEML models adaptation by specifying the actions the system should take when a QoS assertion fails. Each application component may have a group of assertions associated with it. If one assertion does not hold for the component, it indicates a QoS failure that must be fixed. Developers can use J2EEML to specify groups of actions that must be taken to correct these failures.

Once an assertion has failed to hold for a specific component, the application must determine how to fix the problem. To model the appropriate course of action, J2EEML uses the concept of *adaptation plans*, which are groups of actions that can be performed to fix a specific type of QoS assertion failure. For example, if the average response time assertion fails, the *RTM* must change its calculation algorithms to be less precise but run faster. Changing algorithms allows the *RTM* to improve its response time and fix the QoS assertion failure. The steps required to change the calculation algorithms for the *RTM* are modeled as an adaptation plan. The *RTM* implements the Strategy pattern [12] to allow its algorithms to change. The first action from the adaptation plan is to suspend any incoming requests, the second action is to change the algorithm, and the final action is to resume the suspended requests. These actions form the adaptation plan for handling failures in the average response time for the *RTM*.

Adaptation plans indicate the responsibilities of an autonomic layer, i.e., the adaptation plan specifies the actions that the autonomic layer can choose from in the event of a QoS failure. This association also aids in choosing a single-layer or multi-layered planning architecture. If a complex QoS assertion does not have adaptation plans associated with its children, the proper course of action to take when one of the child QoS as-

sertions fails cannot be determined by the data available to the child. If only top-level QoS assertions have associated adaptation plans, this implies the need for a single planning layer. If, however, the QoS children have adaptation plans associated with them, this implies that they can determine the corrective course of action and require a multi-layered planning solution.

Figure 7 shows a J2EEML model that associates the *ResponseTime* QoS assertion with the *ChangeAlgorithms* single-layered adaptation plan. This association indicates that when the *ResponseTime* QoS assertion fails for a component, the *ChangeAlgorithm* adaptation plan should be executed.

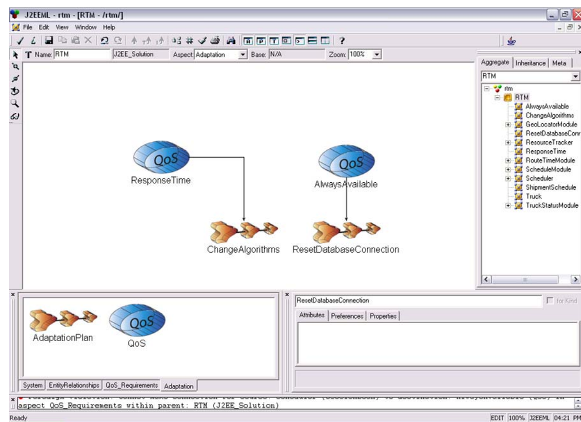

Figure 7: J2EEML Model Associating the ResponseTime QoS Assertion with the ChangeAlgorithms Adaptation Plan

3.4 Reducing the Complexity of Developing Autonomic Systems with JFense and Jadapt

JFense is a component-level framework that performs autonomic functions, such as monitoring the QoS of EJBs, analyzing system state, communicating between autonomic layers, determining how to adapt to QoS failures, and executing adaptation plans. It addresses many of the challenges related to developing autonomic EJB applications. For example, it is responsible for integrating the monitoring, analysis, planning, and execution logic of applications, which alleviates developers from reinventing an autonomic framework for their application. It also handles the communication between layers so that application developers need not write code to move data from the monitoring logic to the analysis logic. JFense provides a configurable event-based monitoring framework that allows developers to focus on writing autonomic logic, which saves developers from re-inventing and debugging a complex autonomic framework. Finally, JFense can coordinate adaptation actions between layers and execute the actions.

Jadapt is a J2EEML model interpreter that supports rapid development and verification of autonomic code by generating implementations of EJBs from a structural model. It serves as a bridge between a J2EEML model and the JFense framework, i.e., it generates Java code for (1) a J2EEML structural model and (2) plugging the generated EJBs into the JFense framework. Jadapt generates configurations for JFense to mirror the J2EEML model, stubs for the EJBs, EJB deployment descriptors, and monitoring, analysis, planning, and execution class stubs, which relieves developers from tedious and error-prone coding tasks. Moreover, Jadapt ensures that the code mirrors the system architecture in J2EEML implementation, which reduces problems stemming from misinterpretation of the specification and inconsistencies between interfaces and their implementations.

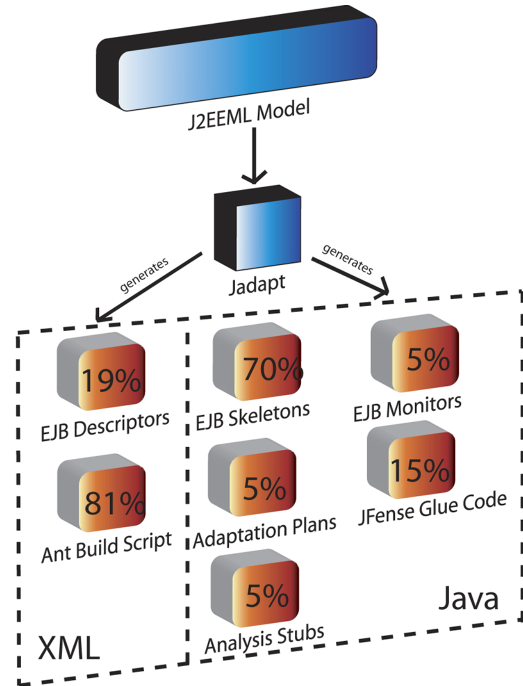

Figure 8: Jadapt Code Generation

Figure 8 illustrates the Jadapt generated code for the highway freight system. The generated code included the Session EJBs for the *RTM*, *Scheduler*, *ResourceTracker*, and *ShipmentSchedule* components. Entity beans were generated for the *ShipmentSchedule*, *Truck*, and *Driver* components. Each Session and Entity beans also had their remote and local interfaces generated, along with the appropriate methods exposed in the model. The generated beans with their associated QoS assertions had the appropriate JFense glue code generated into the implementations. This glue code constructs the appropriate guardian for the beans, captures and relays requests on the exposed methods to JFense, and provides hooks for adapting the component at runtime. Each bean also contained JNDI lookup code for the other beans that it had an interaction with in the model. A remote client test skeleton was generated to obtain a reference to each bean's home, create an instance of the bean, and allow the developer to run tests on the instance.

Jadapt assumes that developers will modify the generated beans outside of the J2EEML development environment. Bean classes are therefore annotated with XDoclet [15] attributes to automate the synchronization of the bean class, interfaces, and descriptors. XDoclet reads these attributes and generates the required interfaces and deployment descriptor XML. Developers only need to maintain the central bean class and synchronize the interfaces to it with XDoclet.

The bean descriptor XML generated by Jadapt includes the transactional, security, visibility, relationship, and container type properties declared in the model. For example, to ensure that the design, deployment, and configuration are in sync, if method `getCoordinates()` on class `RTM` can be accessed by a limited set of security roles, its security declaration will be included in the generated bean descriptor. Generating the bean descriptor eliminates errors from hand-crafting descriptor XML.

For our highway freight scheduler, Jadapt reduced the application development time by generating (1) all skeleton classes and interfaces required for EJBs, (2) XDoclet attributes to maintain class, interface, and deployment descriptor synchronization during the development cycle, (3) XML deployment descriptors for the EJBs, (4) an Ant build infrastructure, (5) project documentation from descriptions captured in the model, and (6) glue code to plug the generated EJBs into JFense. The generated code accounted for approximately one-third of the Java code required to implement the application. The generated XML artifacts, Ant build infrastructure, and documentation required no additional hand-crafting by developers. The generated EJBs required developers to supply the business logic for the exposed methods. The generated JFense analysis and adaptation classes required the appropriate analysis and adaptation logic to be filled in. Since developers can easily adjust the application design and regenerate the application skeleton, their efforts focus on designing the application and implementing the logic required for the business methods.

4 Evaluating Development Effort Savings of the J3 Process

We developed the highway freight scheduling system case study to illustrate the advantages of using the J3 Process to develop autonomic EJB applications. The initial implementation of this case study required several thousand lines of Java code. The generated EJB implementations accounted for nearly 75% of the complete code base, the test framework accounted for 20%, and the JFense glue code accounted for 5%. Using a traditional development approach, much of this code would have been developed manually. With the J3 Process, in contrast, all code except for the business logic and testing logic was generated initially by Jadapt from our J2EEML specification, which accounted for approximately one-third of the code required to implement the Java classes for the application. This section describes how we refactored our design and analyzed the resulting development effort required for changes using both the J3 Process and traditional development processes.

Using our highway freight scheduling case study, we evaluated the impact of adding new sources of information that required monitoring and where the logic would reside. In our initial design, only response times of the *Scheduling* component were monitored. We then refactored the design to monitor response times of the *RTM* component, as well. Adjusting the design using J2EEML and re-generating the implementation took approximately five mouse clicks and resulted in the generation of ~20 new lines of source code that correctly mirrored the specification and was correct-by-construction. Without an MDD tool, in contrast, each change that required the addition of one new connection in the model would have required a manual change to the implementation and more testing to attain the same level of confidence. In a large-scale development effort with many changes, the J3 Process would therefore reduce refactoring costs significantly.

To evaluate the impact of design refactoring on the analysis and planning layers of the highway freight system, we modified its initial design by changing its response time analysis and adaptation into a hierarchy of average and maximum response times. The response time assertion was intended to maintain both properties on the average and maximum response times. Both the average and maximum response times had new adaptation plans designed to counteract degradations in either QoS level. For both adaptation plans, the first action is to suspend the processing of incoming requests. The second action taken is to change the algorithm used by the *RTM* to one that is less precise but faster. The final action taken by the adaptation plans is to resume processing requests. The refactoring in J2EEML was straightforward and took ~12 mouse clicks. The change generated ~75 new lines of code, which minimized the complexity of the design change and implementation update. Again, for large development projects without MDD tool support, many such changes would occur and hence the manual redevelopment effort would be much higher.

To evaluate the development effort associated with sharing adaptation plans between QoS assertions, we refactored our highway freight system to share the improved response time adaptation plan between both the average response time QoS assertion and the maximum response time QoS assertion. The existing plan was then augmented with three new adaptations that were shared between all assertions. After this change was made to the model and Jadapt regenerated the model artifacts, 36 new lines of code were present that updated the existing adaptation plan to include the new adaptations and changed the adaptation plan of the maximum response time to use its modified adaptation plan. As with other refactorings we analyzed, adjusting the J2EEML model and regenerating the code required ~12 mouse clicks, while developing the equivalent functionality manually

required significantly more effort. Knowing what to adapt to meet the new specification is also more straightforward using J2EEML, whereas with a handcrafted approach developers would need to modify multiple source files to accomplish the same task.

As with the autonomic modeling and generation capabilities of the J3 Process, significant reductions in development complexity were yielded by applying MDD to the implementation of the structural model. For example, when a single `SessionBean` with one method was added to the J2EEML model, the resulting bean, interfaces, deployment descriptor, and helper classes generated 116 lines of Java code and 80 lines of XML. The model change in J2EEML required two drag and drop operations. As with the autonomic code generated by Jadapt, the code was correct-by-construction and the JNDI name of the bean was also correct. Adding two interactions from existing beans to the new bean generated another ~12 lines of error-prone JNDI lookup/narrowing code that was automatically generated by Jadapt, thereby simplifying developer effort and enhancing confidence in the results.

As illustrated in these experiments, the mapping between QoS assertions and components facilitates the generation of code artifacts to reduce application development time. For example, understanding which components need to be monitored and analyzed allows Jadapt to generate proxies and interceptors [16] for the monitored components, which can relay state information to the monitoring and analysis logic. Code can then be generated that moves the monitoring and analysis results to the appropriate planning layers.

5 Related Work

An increasing number of MDD tools exist for modeling component-based systems. Cadena [18] is an MDD tool for building and modeling component-based DRE systems, with the goal of applying static analysis, model-checking, and lightweight formal methods to enhance these systems. Other tools, such as Rational Rose, provide UML modeling capabilities for component-based systems. In contrast to J2EEML, these tools are not tailored to the domain of modeling autonomic functionality in component-based systems. For example, they lack the ability to establish the critical mapping between QoS properties, components, and adaptations, which forces developers to (1) resort to traditional textual descriptions for specifying QoS properties and (2) maintain separate models for understanding how the QoS, adaptation, and components in the system interrelate. As a result, it is hard to understand how an application will monitor itself and how it will react to QoS failures.

IBM's Autonomic Toolkit [4] addresses the issues of monitoring, analysis, planning, and executing autonomic applications. It includes the Autonomic Management Engine, which monitors events, analyzes them, then plans and executes corrective action on a computing resource; the Generic Log Adapter [13] for Autonomic Computing, which converts existing log files to the Common Base Event format [14]; and the Log and Trace Analyzer for Autonomic Computing, which reads logs in the Common Base Event format, correlates the logs based on different criteria, and displays the correlated log records. These tools do not, however, address the complexity of integrating autonomic functionality into applications, i.e., they do not help developers design their autonomic applications or implementing the logic required by them. In contrast, the J3 Process is specifically tailored to reducing design and implementation complexity, as well as providing a runtime framework.

6 Concluding Remarks

In theory, autonomic systems can minimize the impact of human error in development and management. In practice, however, it is hard to develop the monitoring, analysis, planning, and execution aspects required for autonomic systems reliably and productively. In particular, developers must reason about complex sets of QoS assertions and ensure that applications meet them. Autonomic capabilities provide a means for EJB applications to self-manage and attempt to maintain the QoS assertions. To facilitate self-management, the structure of EJB applications and their QoS assertions must be captured formally so applications can reason about themselves.

The bridge between the QoS assertions of autonomic systems and their structural designs involves mapping these assertions to specific system components. Without this mapping, applications cannot use introspection to determine whether their QoS assertions are being met. The J3 Process described in this paper provides MDD tools and an autonomic computing framework to support these capabilities to simplify the development of autonomic EJB applications.

The J2EEML MDD tool helps link assertions and structure by allowing developers to specify this mapping via a DSML. J2EEML also includes mechanisms for modeling complex EJB structures, interactions, and architectures and using these models to generate code that is correct-by-construction, which frees developer from reinventing complex autonomic frameworks.

After capturing structural properties, QoS assertions, and assertion to structure mapping in J2EEML, developers still must integrate autonomic features into their -distributed EJB applications. This integration is often complicated due to the lack of component-level frameworks for autonomic systems. To address these concerns, we have developed the Jadapt code generation tool and the JFense autonomic framework. Jadapt allows developers to generate the code needed to plug their application's EJBs into JFense. JFense provides a comprehensive and flexible framework for multi-layered autonomic monitoring, analysis, planning, and execution architectures, which allows developers to focus on the system's business logic and QoS analysis logic.

The following are our lessons learned thus far by developing and applying the J3 Process:

- Creating a flexible system to aid the development of autonomic EJB applications is hard, e.g., not all applications want to monitor the same types of data sets. A DSML must therefore be flexible to incorporate unanticipated data sets, yet also handle the most common cases intuitively. Striking this balance between flexibility and general case utility took patience and iteration.
- Developing adaptations for an application is hard. Most developers do not think about designing components that can be adapted, swapped, restarted, or reconfigured to handle errors. Providing a DSML to aid developers in seeing the crosscutting adaptive concerns was hard.
- Creating a model of the mapping from components to QoS properties and adaptive behavior greatly enhances the ability of developers to understand the complex behavior of autonomic systems that would ordinarily be buried in hundreds of source files.
- Constraint checking and code generation can greatly reduce and/or eliminate hard-to-debug JNDI naming errors. Constraint checking of JNDI allows these errors to be detected at design time rather than runtime.

In future work, we are developing increasingly sophisticated autonomic distributed applications using our J3 Process MDD tools to serve as a testbed for investigating various autonomic architectures, monitoring strategies, and planning strategies. We are also enhancing these tools to increase their expressive and code generation capabilities. We plan to integrate our MDD tools with CIAO [6], which is an open-source, QoS-enabled CORBA Component Model (CCM) implementation.

The J3 Process DSMLs, tools, and frameworks are available at www. sourceforge.net/projects/j2eeml.

References

[1] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. IEEE Computer, January 2003.

[2] D. Oppenheimer, A. Ganapathi, D. Patterson, Why do Internet services fail, and what can be done about it?, "USENIX Symposium on Internet Technologies and Systems," March 2003.

[3] V. Matena, M. Hapner, "Enterprise Java Beans Specification, Version 1.1," Sun Microsystems, Dec. 1999.

[4] Autonomic Computing Toolkit, IBM, www106.ibm. com/developerworks/autonomic/overview.html.

[5] G. Candea, A. Fox, "Designing for High Availability and Measurability," Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability (EASY), July 2001.

[6] N. Wang, D. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. Loyall, R. Schantz, and C. Gill, "QoS-enabled Middleware," in *Middleware for Communications*, edited by Q. Mahmoud, Wiley and Sons, New York, 2003.

[7] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," IEEE Computer, Nov. 2001.

[8] T. Eymann, M. Reinicke, et al., "Self-Organizing Resource Allocation for Autonomic Networks," DEXA Workshops, 2003.

[9] A. Ledeczi "The Generic Modeling Environment", Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001.

[10] D. Alur, J. Crupi, D. Malks, "J2EE Core Patterns," Sun Microsystems Press, 2003.

[11] J. Gray and S. Roychoudhury, "A Technique for Constructing Aspect Weavers Using a Program Transformation Engine", Proceedings of AOSD '04,, Lancaster, UK, March 22-26, 2004.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reus

[13] E. Giguere, "Create GLA components using Release 2 of the Autonomic Computing Toolkit," IBM Developerworks, (www106.ibm.com/ developerworks/edu/ac-dw-ac-glacomp2i.html? TACT=104AHW20&S_CMP=HP).

[14] " Specification: Common Base Event ," IBMDeveloperworks, (www106.ibm.com/developerworks/ webservices/library/ws-cbe/).

[15] XDoclet, (xdoclet.sourceforge.net/xdoclet/).
able Object-Oriented Software,"Addison-Wesley, 1995.

[16] R. Koster, T. Kramp, "Loadable Smart Proxies and Native-Code Shipping for CORBA," in *Proceedings of the Third IFIP/GI International Conference on Trends towards a Universal Service Market (USM), Munich*, September 2000.

[17] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, "QoS Aspect Languages and Their Runtime Integration.," in *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Components,* May 1998.

[18] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," Proceedings of the 25th International Conference on Software Engineering, Portland, OR, May, 2003.