Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Engineering and Simulation

JULES WHITE, DOUGLAS C. SCHMIDT, AND ANIRUDDHA GOKHALE

Vanderbilt University, Department of Electrical Engineering and Computer Science, Box 1679 Station B, Nashville, TN, 37235

{jules, schmidt, gokhale}@dre.vanderbilt.edu
(251)-533-9432

The goal of autonomic computing is to reduce the configuration, operational, and maintenance costs of distributed applications by enabling them to self-manage, self-heal, and self-optimize. This paper provides two contributions to the Model-Driven Engineering (MDE) of autonomic computing systems using Enterprise Java Beans (EJBs). First, we describe the structure and functionality of an MDE tool that visually captures the design of EJB applications, their quality of service (QoS) requirements, and the autonomic properties applied to their EJBs. Second, the paper describes how MDE tools can be used to generate code to simulate autonomic systems for verification and plug EJBs into a Java component framework that provides autonomic capabilities.

Keywords: Autonomic Computing, Model-Driven Engineering, Enterprise Java Beans

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 [28], which have been shown to be a significant contributor to distributed system repair and down time [19]. 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 [13], thereby reducing human interaction with the system to minimize down-time from operator error. Although the benefits of autonomic computing are significant [13], 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, such as Enterprise Java Beans (EJB) [17], offer limited autonomic features, such as application server 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 today is the lack of design tools and frameworks that can (1) alleviate the complexities stemming from the use of manual development methods, (2) generate code that mirrors the specifications of the model, and (3) provide mechanisms for validating adaptive systems early in the design cycle. Some infrastructure does exist, such as IBM’s Autonomic Computing Toolkit [10], which focuses on system-level logging and management. System-level autonomic toolkits are inadequate, however, for fine-grained autonomic capabilities, such as adjusting algorithms to handle different request demands, which are intended 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 more 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 components selectively as they fail, 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 [4], which can incur excessive overhead, particularly for Java-based systems due to JVM initialization latency. In contrast, a component-level autonomic framework provides mechanisms to restart only the point of failure [3].

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. Manually analyzing application autonomic aspects such as checking whether the right state is being monitored by the right components, is a complex process. Moreover, creating a system that provides component-level adaptation greatly increases the difficulty of assuring that it operates properly throughout all of its adaptive modes [27].

Simplifying autonomic system development via MDE techniques. *Model-Driven engineering* (MDE) [21] is a generative software paradigm that combines

- *Domain-Specific Modeling Languages* (DSMLs) whose type systems formalize the application structure, behavior, and requirements within particular domains, such as software defined radios, avionics mission computing, online financial services, warehouse and freight management, or even the domain of middleware platforms. DSMLs are described using *metamodels*, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Developers use DSMLs to build applications using elements of the type system captured by metamodels and express design intent declaratively, rather than imperatively.
- *Transformation engines and generators* that analyze certain aspects of models and then synthesize various types of artifacts, such as source code, simulation inputs, XML deployment descriptions, or alternative model representations. The ability to synthesize artifacts from models helps ensure the consistency between application implementations and analysis information associated with functional and quality of service (QoS) requirements captured by models.

MDE tools are a promising means of reducing the cost associated with creating and validating autonomic computing systems. Models of autonomic systems developed with MDE tools can be constructed and checked for correctness (semi-)automatically to ensure that application designs meet autonomic requirements. These tools can also 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 manually imbue autonomic qualities into distributed applications – we have created the *J3 Toolsuite,* which is an open-source[1] MDE environment that supports the design and implementation of EJB autonomic applications. J3 consists of several MDE tools and their supporting autonomic computing frameworks including (1) *J2EEML*, which is an MDE tool that captures the design of EJB applications, their QoS requirements, and the autonomic adaptation strategies of their EJBs via a DSML [14], (2) *Jadapt*, which is an MDE tool that analyzes the QoS and autonomic properties of J2EEML models, and (3) *JFense*, which is an autonomic computing framework that enables the monitoring, configuring, and resetting of individual EJBs [6].

This paper builds on our previous work on autonomic systems presented in [26] and provides new results on our work that simplifies the design and validation of systems with component-level autonomic behavior. The paper describes the structure and functionality of J2EEML and shows how it simplifies autonomic system development by providing notations and abstractions that are aligned with autonomic comput-

---

[1] The J3 Toolsuite DSMLs, tools, and frameworks are available in open-source form at www.sourceforge.net/projects/j2eeml.

ing, QoS, and EJB terminology, rather than low-level features of operating systems, infrastructure middleware platforms, and third-generation programming languages. We also describe how

- Jadapt generates Prolog simulation and Java implementations from J2EEML models to ensure that autonomic applications meet their specifications with minimal manual coding,
- Prolog Qualitative Differential Equation (QDE) simulations generated from Jadapt can greatly reduce the complexity of system validation, and
- JFense provides a set of reusable autonomic components that allow developers to plug-in EJB applications and focus on crafting their autonomic logic, rather than writing the glue code for constructing autonomic systems.

Finally, we present a case study that qualitatively and quantitatively evaluates how the J3 Toolsuite reduces the complexity of developing an autonomic EJB application.

Our case study centers on an EJB-based *Constraint Optimization aNd Scheduling sysTem* (CONST) 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. Since a heavy load is placed on the *RTM*, it must be designed to maintain its *QoS assertions*, such as ensuring that the RTM does not exceed a maximum response time 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.

Figure 1

Paper organization. The remainder of this paper is organized as follows: Section 2 describes the MDE J3 Toolsuite we created to simplify the development of autonomic EJB applications; Section 3 gives an overview of the J3 Toolsuite and describes key challenges we faced when developing it; Section 4 presents the Prolog QDE simulation and validation environments generated by Jadapt; Section 5 quantifies the reduction in manual effort achieved by using the J3 Toolsuite on our CONST case study shown in Figure 1; Section 6 compares our work with related research; and Section 7 presents concluding remarks and summarizes lessons learned from our work.

2 The J3 Toolsuite for Autonomic System Development

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

- **J2EEML**, which is a DSML-based MDE tool tailored for designing autonomic EJB applications. J2EEML uses visual representations to model domain-specific abstractions, such as beans, QoS properties, and adaptations. J2EEML also specifies the mapping from QoS requirements to application components.
- **Jadapt**, which is an MDE 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 computing framework that provides components for monitoring, analysis, planning, and execution. Developers can use these components to avoid writing custom autonomic software. JFense can be configured to meet the autonomic requirements for a range 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.

The J2EEML DSML enables EJB developers to construct models that incorporate autonomic and QoS concepts as first-class entities.

J2EEML itself was developed for both the *Generic Modeling Environment* (GME) [15] and the *Generic Eclipse Modeling System* (GEMS) [22], which are general-purpose MDE environments that were created to simplify the creation of *metamodels* and *model interpreters*. The J2EEML metamodel characterizes the roles and relationships in the autonomic computing domain, and model interpreters 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 comprising 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.

Figure 2

2.1 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 six steps:

1. Each session bean is represented by dragging and dropping session bean atoms into the J2EEML model. Developers then provide the Java Naming and Directory Interface (JNDI) name of each 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. Figure 2 shows a model of the remote interface composition of the *TruckStatusModule* from CONST.
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 of 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

Figure 3 shows a J2EEML structural model of our CONST case study application described in Section 1. In this figure, each bean within CONST has been modeled 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 enterprise architectures into smaller pieces, J2EEML allows EJB structural models to contain child EJB structural models or subsystems. Beans within these children show up as ports that can receive connections from the parent solution. This hierarchical design allows developers to decompose models into manageable pieces and enables different developers to encapsulate their designs.

For CONST, 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. Finally, we set the various transactional, remote visibility, and state type properties of the various bean model entities.

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 [1], which hides Entity beans from clients through Session beans, and (3) model validation, including checks for proper JNDI naming.

3 Designing J2EEML to Address Key Concerns of Autonomic Computing

Autonomic applications require four elements to achieve their goals: *monitoring, analysis, planning, and execution* [13]. These elements form a *controller* that observes and adapts the application to maintain its functional and QoS goals, such as maintaining a minimum response time of 100ms for requests. This section describes how the monitoring, analysis, and planning aspects of autonomic systems presented unique challenges when designing and building J2EEML and shows how we addressed each challenge. To focus the discussion, we use the CONST *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 impacting 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 the issue of how to self-monitor key data, e.g., by capturing CPU and memory utilization, exceptions thrown by the application, 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 of J2EEML is that autonomic EJB applications can measure properties of their current state introspectively and determine if the property values indicate the application is in a safe or optimal state. J2EEML models the properties it measures via *QoS assertions*, which determine which properties an autonomic system can introspectively measure 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.

QoS assertions can be based on a traditional measurement of system health, such as response time, or a non-traditional measurement, such as the most recent sequence of thrown exceptions. J2EEML allows assertions based on both types of variables. For traditional measurements based on continuous variables, such as response time, a QoS assertion specifies a list of conditions under which it is active. For example, the QoS assertion "*Location Management System too Slow*" might become active when the response time of the Geo Database is greater than 300ms and the response time of the RTM is greater than 2 seconds. Developers create QoS conditions by placing *Condition* model entities as the children of QoS assertions. Each Condition entity specifies a continuous variable, a comparator, such as less than, and a value to compare to. For non-traditional measurements, the appropriately named QoS entity is added to the model but no continuous variable conditions are specified. In Sections 4 and 5, we provide further discussion on how non-traditional assertions are utilized for simulation and code-generation.

The continuous variables utilized by J2EEML may be directly measured from the system or derived from other known values of continuous variables. Continuous variables are described by dragging and dropping *ContinuousVariable* entities into the model. Each ContinuousVariable may contain multiple child Landmark entities that indicate the distinguished values of the variable that are relied upon by the QoS assertions. For our “Location Management System too Slow” assertion, we are interested when the response time of the Geo Database exceeds 300ms. The developer would first create a ContinuousVariable named “*Geo Database Response Time*”. Instead of directly specifying the 300ms landmark, the developer adds a Landmark abstraction to the “Geo Database Response Time” called “*Response Time Too High for Location Management.*” Further Landmark abstractions may be added as well, such as “*Response Time Allows Increased Accuracy*,” to indicate that the Geo Database's response time is below the level allowed to use more accurate positioning algorithms. Abstracting away the exact values of the distinguished landmarks of the variables allows for the behavior to be modeled before real values are known. This type of abstraction also allows the behavior to be modeled independently of the underlying hardware. If the autonomic system runs on a different platform, the description of the behavior is still correct although the Landmark abstractions may be bound to different real values. The “Location Management Too Slow” assertion would then have a Condition child added specifying that it is active when “Geo Database Response Time” is greater than “Response Time Too High for Location Management.”

To provide for relationships between continuous values in the system, J2EEML provides mechanisms for specifying how a continuous variable is derived from another continuous variable. Two or more ContinuousVariables can be connected through *Derivation* connections specifying how they evolve with respect to each other. Continuous-

Variables may be monotonically increasing, monotonically decreasing, sum, difference, product, or quotient functions of each other. ContinuousVariable derivations can be used to specify queuing relationships, such as deriving mean queue length from arrival rate and service time, or other valuable properties. In Section 4, we discuss how ContinuousVariables, Landmarks, and Derivations are used by Jadapt to facilitate validation and simulation of an autonomic application and reduce the cost of verifying its correctness.

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 trade-offs 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 and reliability. 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 [16] 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 generate the appropriate monitoring code in the *RTM*'s implementing class.

Figure 4

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 [20].

In CONST, for example, we must ensure separation of concerns in the application design and find an efficient means of monitoring. A natural approach to collecting request statistics for the RTM might be to simply add the appropriate state collection code into the route time logic. The monitoring logic for the *RTM*, however, should not be entangled with the route time calculation logic, and reduce its readability and maintainability. Moreover, the time to monitor and analyze each request should be insignificant compared to the time to fulfill each route request.

After the structural and QoS 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 monitor its route time calculation requests.

J2EEML supports aspect-oriented modeling [8] 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 EJB, produces clearer models. It also 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.

Figure 5

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 *NoExceptionsThrown* and *NeverReturnsNull*. Combining *MinimumResponseTime* and *NoExceptionsThrown*, 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, which 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. To choose a hierarchical multi-layered (composed of a tree structure of analysis components) vs. monolithic single-component 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 hierarchy or number of layers of analysis logic. A key issue to consider is whether an application should have a single-layer vs. a hierarchical 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 must also 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 the RTM’s autonomic layer analyzes its 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 to monitor every request to the *RTM* 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 simplify complex QoS analyses via divide-and-conquer. A hierarchical QoS assertion is only met if all its child assertions are met, i.e., 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 Figure 6, where *GoodResponseTime* is an aggregation of several properties of the response time.

Figure 6

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

Modeling QoS assertions hierarchically also enhances developer understanding of how to decompose the analysis engine into layers. The hierarchical model of 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. Further decomposition analyses can be performed by taking into account the adaptive actions triggered by each QoS assertion. In Section 3.3.1, we discuss these other analyses.

3.3 Planning

Planning is the phase in autonomic systems where applications examine the results of their analyses 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 planning 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. For applications that need more flexible and fine-grained control of their behavior this simple one-layer architecture is less suitable. For example, if the RTM needs to switch algorithms in response to a degradation in response time, a small controller located close to the RTM would be able to react more quickly and with less overhead than a larger controller located farther away. If however, the RTM needed to switch algorithms due to a period of high demand predicted from historical data, a small controller located close to the RTM is infeasible since it is unlikely to have access to the appropriate data for the prediction. Moreover, a predicted period of high demand may necessitate changes to components other than just the RTM and thus require a large monolithic controller with access to multiple components. 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 CONST, one layer ensures that the *RTM* is always available and the next layer down ensures 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.

Modelers can specify each action and its affect on system state. Intention entities can be dragged to an action to specify which continuous variable will be changed by the action and the direction of its influence, either increase or decrease, thereby enabling modelers to capture the affects of actions and identify actions that have incompatible intentions. As discussed in Section 4.2, these intentions can automatically identify possible unsafe evolutions of autonomic system state.

Once an assertion has failed to hold for a specific component, the application must determine how to fix the problem. To model the appropriate actions, J2EEML uses *adaptation plans*, which are groups of actions that 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. Figure 7 shows a J2EEML model that associates the *ResponseTime* QoS assertion with the *ChangeAlgorithms* single-layered adaptation plan.

Figure 7

Adaptation plans indicate the responsibilities of an autonomic layer, i.e., the adaptation plan specifies the actions that the autonomic layer can perform in the event of a QoS failure. This association also guides the selection of 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 assertions 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.

The key to determining the right granularity of the analysis and planning is determining the relationship between QoS assertions and the adaptation plans they trigger. A multi-layered QoS assertion with an adaptation plan containing a single action cannot be broken into several independent analysis and planning layers since it requires a specific combination of values of several continuous variables to become active. If an adaptation plan contains multiple actions that each are relevant if one of the continuous variable conditions is true, the plan can be subdivided into a hierarchical set of QoS assertions where each assertion is based on a single continuous variable and action.

For example, consider a QoS assertion, *Improve_ Response_Time,* to reduce the response time of the RTM that contains two continuous variable conditions, *RTM_Response_Time* > *Medium* and *RTM_Request_Arrival_Rate* > Medium, and two adaptive actions *Use_Less_Accurate_Algorithms* and *Use_Batch_Processing*. In this case, Use_Less_Accurate_Algorithms is relevant when the response time is too high and Use_Batch_Processing is relevant when the request arrival rate is high enough to make it efficient. In this case, Improve_Response_Time could be composed hierarchically of two QoS assertions, one that switches to less accurate algorithms when the response time is too high and one that turns on batch processing when the request arrival rate is sufficiently high.

QoS assertions that are based on a single continuous variable that is derived from one or more other continuous variables may also be subdivided if the adaptive actions that are triggered by it are designed to alter the variables from which the conditional variable is derived. For example, consider a QoS assertion that when the RTM's mean queue length exceeds a specific landmark switches the RTM to less accurate algorithms and causes the RTM to begin rejecting some portion of its requests (possibly based on priorities). In this case, the continuous condition is based on mean queue length which is derived from the arrival rate and service time of the RTM. Clearly, using less accurate algorithms will decrease service time and rejecting requests will cap or decrease the request arrival rate. Thus, the QoS assertion can be subdivided into two QoS assertions, one that attempts to decrease the arrival rate when it exceeds a threshold and one that decreases the service time when the mean queue length exceeds a specified value. Having a model of the continuous variables, their derivations, and the adaptive conditions for the system greatly informs decisions on whether to choose a monolithic or layered planning and analysis engine and how a layered engine should be hierarchically composed.

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

Figure 8

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. Figure 8, shows the high-level architecture of the J3 Toolsuite and how JFense fits into it. Jadapt is a J2EEML model interpreter that supports rapid development and verification of autonomic code by generating implementations of EJBs from a structural model.

Jadapt is a bridge between a J2EEML model and the JFense framework, i.e., it generates Java code for (1) the 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 the J2EEML implementation, which reduces problems stemming from misinterpreting specifications and inconsistencies between interfaces and their implementations.

Figure 9

To simplify the development of autonomic EJB applications, we created the JFense framework for constructing autonomic EJB systems. JFense provides a multi-layered architecture for monitoring, analyzing, planning, and executing in an autonomic system. The basic structure of JFense is defined as follows:

1. Each bean has a guardian class responsible for monitoring its state and running QoS analysis, as shown in Figure 9. The beans push state data out to the guardians using an event-based system. The guardians act as observers on the beans, i.e., they are the key elements for monitoring beans and routing state information to the proper QoS analysis objects.
2. An analysis class for each QoS goal is created by Jadapt. These QoS goals are used by the guardians to analyze the bean’s current state and determine if it is meeting its QoS requirements. Hierarchical QoS goals are created through aggregation.
3. Each guardian class has an associated action plan for determining the course of action if a QoS goal fails. The guardian also notifies any guardians at the level above when it cannot maintain its QoS goals.

When a bean's state changes, it notifies its guardian that a state change event has occurred. The guardian then uses each of its QoS analysis objects to analyze the bean's state and ensure that its objectives are still being met.

Bean requests are the default state information monitored by guardians. Jadapt generates proxies that monitor the input, output, time, and exceptions thrown for each method accessible through the beans local or remote interface and pass it to the Guardians.

Beans monitor requests on their accessible methods through generated proxies. When a request is issued to the bean, the generated proxy first receives the request and notes the starting time. The proxy then notifies the guardian that a request is starting so that any pre-conditions on the request can be analyzed. These pre-conditions can be used to identify QoS failures in other portions of the system, other systems, or clients. The proxy then passes the request to the actual method that contains the logic to fulfill it (we refer to this method as the *implementing method*). When the implementing method has returned, the bean again notifies its guardian, which enables the guardian to check post-conditions, such as output correctness or servicing time. Finally, the result is passed back to the caller.

After the state is routed to the analysis object, it determines if its QoS property is being met. JFense has several predefined analysis objects for common functions, such as monitoring request time. Other autonomic analyses can be added by extending the JFense analysis interfaces or implementing the class skeletons generated by Jadapt from the J2EEML model. If the QoS is not being maintained, the analysis object notifies the guardian, which will either directly execute an action plan or propagate the QoS failure event up the chain of guardians.

Guardians also use the Strategy pattern [7] to determine how to react to a QoS failure. Different planning strategies can be plugged into a guardian at design- or run-time to find the appropriate course of action for each QoS failure. Strategies can be plugged in at both design and run-time. The default strategy uses a hashing scheme to associate QoS analysis objects with Command pattern [7] actions, which encapsulate actions as objects, to allow adaptations to be queued, logged, or undone. In the event of a QoS failure, the appropriate action is looked up from the table and executed.

JFense alleviates developers of the need to build an autonomic framework from scratch. In the highway freight scheduling system, for example, JFense handles inter-layer communication so that developers can focus on the logic needed to analyze the state data, determine the correct course of action, and adapt the system. JFense also provides the communication, monitoring, and message bus infrastructure to glue the provided logic together, which significantly reduces the time and effort required to build autonomic applications that monitor their own state and adapt to achieve their goals.

4 Simulating and Validating Autonomic Systems with the J3 Toolsuite

To address the challenge of testing and validating a system with component-level autonomic properties, we developed a simulation environment for the J3 Toolsuite based on the Qualitative Differential Equation (QDE) simulation algorithm [24]. The J3 Qualitative Simulation (QSim) environment is an adaptation of the Prolog implementation described in [25]. QSim predicts the possible behaviors consistent with a qualitative differential equation model of a system. A QDE model is an abstraction of an ordinary differential equation model that specifies a set of real valued variables and the functional and algebraic constraints between them [24]. QDE models abstract away real values of variables in favor of a finite set of *Landmarks*, or values of interest. The values of variables can then be constrained to monotonically increasing or decreasing functions of each other to describe how their states evolve with respect to each other.

QDE simulation is well-suited for decreasing the complexity of validating the behavior of an autonomic system in a specific environment. The J3 simulator treats the *ContinuousVariables* described in J2EEML models as the environment that the autonomic application reacts to. The QoS assertions specify regions of values of the *ContinuousValues* in which *AdaptationPlans* become active. Using this information, the J3 simulator can predict the evolution of the *ContinuousVariables* (the autonomic system's environment) and discover the sequence of autonomic adaptations that will occur. This type of simulation allows developers to simulate and validate the behavior of an autonomic system early in the design cycle and catch design errors before the application is implemented and they are more costly to correct.

4.1 Validating Adaptive Assumptions via Simulation

One difficulty of designing an autonomic application manually is it is hard to understand what states the application will enter due to its adaptations. For example, it may be difficult to foresee that one adaptation may trigger a series of changes in the application that cause two other conflicting adaptations to occur. For example, assume that the Optimizer contains an adaptation that when the Optimizer is under light load causes it to switch to a more accurate mode of operation. Under the more accurate mode of operation, the Optimizer requests that the RTM also use a more accurate route time algorithm. Assume that the RTM also has an adaptation that when under heavy load causes it to switch to a less accurate algorithm. In this situation, it is possible for the Optimizer and RTM to enter a situation where the Optimizer is requesting that the RTM use more accuracy and the RTM is trying to decrease accuracy. These adaptations are not compatible, since both involve disruptive algorithmic changes, but can become active together. It is crucial that these types of conflicting adaptive plans be identified. Without a formal method of predicting how and when a set of adaptations will occur, an autonomic application requires enormous amounts of testing to ensure that it has been tested in all of its possible adaptive states.

The J3 simulation environment allows developers to query the Prolog autonomic simulator for the possible adaptive executions of the system given an initial state for the continuous variables of the environment. The J3 simulator uses the QSim algorithm to evolve the continuous variables according to the QDE model described in J2EEML and identify the AdaptationPlans that are active in each environmental state. In each simulation step, the J3 simulator identifies the currently active adaptations and their modeled affect on the environment. For example, if an action is triggered that decreases the response time of the RTM, the simulation will ensure that the RTM's response time either transitions from increasing to steady or steady to decreasing in the next simulation step.

Questions can be posed to the J3 simulator, such as "can a system state be reached in which both the Optimizer's increase accuracy adaptation and the RTM's decrease accuracy adaptation are active?" This querying capability allows developers to validate their assumptions about the autonomic behavior and ensure that error conditions do not occur. The simulator can also be posed questions, such as "can the system ever reach a state such that the RTM's response time is greater than the high landmark?" This querying capability allows developers to not only model an autonomic application but to check that their assumptions about its behavior are correct.

Another aspect of the simulation is the role of the QoS assertions that are not dependent on a continuous variable condition but instead on a non-simulatable property, such as the most recently thrown series of exceptions. For these assertions and the adaptation plans they trigger, the simulator assumes that they can become active in any system state. Simulating all valid adaptive states that could be produced by assuming that the non-simulatable assertions could produce would be extremely complex.

It is also beneficial that developers include as few adaptations that affect the continuous variables as possible since they make the behavior of the system non-deterministic. By default, the simulator does not factor in the intents of these assertions but can be enabled to do so. Developers can still, however, make useful deductions about the non-simulatable properties. For example, they can query the simulation for states that can be reached where a simulatable-assertion triggers an action that is incompatible with an action of a non-simulatable assertion. The non-simulatable assertions can still be used to identify possibly unsafe sets of adaptations.

4.2 Reducing QDE Simulation Complexity with J3

As discussed in [24], the QSim algorithm can experience a combinatorial explosion if a large number of continuous variables are modeled that each have multiple landmarks. Each state of the system environment is denoted by a set of tuples identifying each variable's current landmark and the direction of change of the variable. Landmarks may either be a specific value from the model, such as medium, or a value between two landmarks, such as between medium and high (denoted by medium..high). Since the variables are continuous, between any two system states, a variable may remain at the same landmark, or change to the next landmark with either the same direction of change or steady. For example, the RTM's response time may transition from medium/std to low..medium/dec, medium...high/inc, or medium/std. At each step in the system state, there are four possible next states for each variable. Thus there are 4^C possible next system states, where C is the number of continuous variables. Clearly, a large number of continuous variables can create a combinatorial explosion and make evolving the system computationally expensive.

Autonomic systems, modeled in J2EEML, exhibit a unique property that greatly reduces the number of possible next system states (by system, we are describing the continuous environment of the application). Each landmark of a continuous variable indicates a point at which an adaptive action is enabled. Each action, in turn, has a specific affect on the continuous variables of the system, which is specified through the Intention entities. In system states where adaptation actions are active, the number of next states is reduced since the actions fix the direction of change of the continuous variable it is affecting. If an adaptive action is active that affects a continuous variable, then that variable can have at most two possible next states. Either the variable remains between two landmarks (between the landmarks X and Y is denoted X ..Y) and switches its direction of change or it remains between two landmarks with the same direction of change, as seen in Figure 10. If the variable is exactly at a single landmark, it will have only one possible next state that is proscribed by the direction of change.

Figure 10

For example, if the RTM's response time is medium/inc and an adaptation becomes active that reduces the response time, the variable must transition to medium/std. If the action caused the response time to increase, the variable would transition to medium..high/inc. Each variable thus has exactly one or two next states when affected by an adaptation action. At any state, if *T* variables are proscribed to two next states by actions and *O* variables are proscribed to a single next state, the system has $(2^T)(4^O)$ fewer possible next states. The J3 Toolsuite is therefore most effective at simulating systems with a large number of adaptive actions relative to the number of continuous variables. For example, if we have ten different adaptive actions that are triggered by various states of two continuous variables, the continuous variables will be proscribed to a significantly smaller set of states than if fewer adaptive actions were present. Systems with large numbers of adaptive actions are precisely those that are most difficult to analyze manually.

It is also worth noting that a continuous variable will only be present in a model if it has a QoS assertion that relies on it. Every continuous variable will therefore have at least one adaptive action that can influence it. Each continuous variable will also have at most one landmark, corresponding to no active adaptations, that isn’t used by a QoS assertion. Any other landmark of a continuous variable must correspond to a value of interest to a QoS assertion. More than one unused landmark is superfluous information that merely increases the simulation complexity.

4.3 Analyzing Architectural Properties to Inform Design Decisions

The J3 simulator can also be asked static questions about the autonomic system, such as “are there any QoS assertions that are based on a single continuous variable and trigger adaptive actions that modify the constituents from which the variable is derived?” This question directly corresponds to the derivation decomposition analysis described in Section 3.3.1. The query is resolved by asking Prolog if there exists a QoS assertion whose QoS condition relies on a variable Z such that Z is derived from variables X,Y and the adaptation plan of the assertion contains two actions with intentions that affect X and Y. We can therefore use the Prolog simulation model to identify QoS assertions that can be decomposed and guide architectural design decisions.

The other decomposition analysis described in 3.3.1 can also be aided by the Prolog simulation. The user can query Prolog for a QoS assertion with conditions based on a set of continuous variables *C*, that triggers and adaptation plan containing actions *A*, such that each action in A affects exactly one variable in C. This query corresponds to finding QoS assertions that contain multiple conditions that could possibly be split into several QoS assertions each containing a single condition and triggering a single adaptation on their continuous variable.

4.4 Generating the Simulation from J2EEML

A straightforward transformation is used to generate a Prolog knowledge base for the QSim algorithm. For each model entity, a predicate/argument statement is generated specifying the type of the entity and its unique id. A QoS assertion with id 23, for example, will be transformed into the Prolog statement “qosassertion(23).” Properties of the entities are transformed into predicate relations on the id of the entity and the value of the property. A QoS assertion with id 23 and named “ResponseTime” would generate the predicate self_name(23,’ResponseTime’). Relationships between entities denoted by connections or containment, such as the Adaptation plan which is used by a QoS assertion, become predicate statements relating the two ids of the related entities. If ResponseTime uses the *ChangeAlgorigthms* Adaptation plan with id 24, a predicate self_adaptationplan(23,24) would be generated.

After the Prolog knowledge base is generated by Jadapt, a predefined set of rules are generated to connect the knowledge base to the QSim algorithm. A sample rule to find the landmarks of a continuous variable is:

```
landmarks(VarName,LandmarkNames) :-
   self_name(VarId,VarName),
   self_landmarks(VarId,Landmarks),
   findall(LName,
          (member(V,Landmarks),
           self_name(V,LName), Landmarks).
```

Similar rules are generated to determine the active actions given a system state, the set of continuous variables, and other rules needed by the Prolog implementation of the QSim algorithm described in [24].

5 Evaluating Development Effort Savings of the J3 Toolsuite

We developed the CONST highway freight scheduling system case study described in Section 1 to show the advantages of using the J3 Toolsuite to develop autonomic EJB applications. The initial implementation of this case study required ~1,200 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, all of this code would have been developed manually. With the J3 Toolsuite, in contrast, ~883 lines of code were generated by Jadapt from our J2EEML specification.

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. This refactoring can be seen in Figure 11.

Figure 11

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 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 MDE 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. 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.

As with the autonomic modeling and generation capabilities of the J3 Toolsuite, significant reductions in development complexity were yielded by applying MDE 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.

6 Related Work

Kandasamy et. Al [10] present an online optimization framework for autonomic applications using a hierarchical architecture similar to J2EEML. The paper also presents forecasting and control strategies for managing an autonomic application. This work is complementary to the J3 Toolsuite in that it provides methods for designing how an autonomic system reacts and behaves. Moreover, the hierarchical control structure used by JFense could be adapted to incorporate the methods presented in [10]. One significant difference of the J3 Toolsuite and the work presented in [10] is that J3 is an MDE approach to building autonomic applications. The optimization framework presented in [10], conversely, does not provide an integrated modeling tool or simulation environment as J3 does.

Many of the decomposition analyses that the J3 simulator enables are similar to the design rules that are proposed in [30,31], i.e., the simulator promotes the decision of how to decompose the QoS entities into appropriate independent modules so that they can be developed and function independently of each other. The static Prolog representation of the QoS entities, continuous variable conditions, and specified adaptive action intentions, allow automated analysis of whether the entities can be further decomposed into independent modules. The simulation work is complimentary to these types of design rules and can be used to automate some of types of design analysis for large scale systems.

An increasing number of MDE tools exist for modeling component-based systems. Cadena [9] is an MDE 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, additional code must be written to enable an application to monitor itself and specify how it will react to QoS failures.

Other research initiatives present middleware approaches to managing the QoS of distributed applications similar to JFense. The Generic Object Platform Infrastructure (GOPI) [5] provides a pluggable and modular platform for the development of middleware. GOPI, in particular, includes support for annotating interface interaction points with QoS attributes. As with the J3 Toolsuite, there is no limitation on what can be considered a QoS attribute. These attributes are mapped to specific middleware configurations through code to tailor an application's performance. QoS groups can be created to partition the interaction points into sets that share QoS requirements. JFense also provides the ability to associate components that have similar QoS requirements. JFense, however, allows a single component to be associated with multiple QoS groups whereas GOPI does not. In GOPI, each communication protocol can have a *QoS manager* associated with it to ensure that a communication binding maintains its required QoS. This design is similar to the JFense approach of using *Guardian* classes to monitor EJBs and notify the appropriate adaptations when QoS degrades. GOPI requires that developers implement the planning logic that determines what response should be taken to a QoS degradation. By using the J3 Toolsuite, the planning logic is automatically generated from the J2EEML model. Furthermore, adaptations can be written once and incorporated into multiple aspects of an application by merely updating the J2EEML model and regenerating the JFense code. Using a model-driven middleware approach provides significant benefits to the implementation and refactoring of adaptation logic when compared to hand-coding with a platform such as GOPI.

QuO [23] is another middleware architecture for mapping QoS to objects. In QuO, the state of the operating environment can be partitioned into regions. Transitions between these regions trigger adaptive behavior. This architecture is similar to how JFense operates, i.e., JFense adaptations occur as assertions become true or false. A key difference between the J3 Toolsuite and Quo is that J3 is a complete model-driven process for developing adaptive applications and not just a QoS-aware middleware framework. With J3, most of the tedious configuration and implementation code is generated from the modeling tool. As discussed previously, this greatly reduces the cost of refactoring adaptations as the understanding of the target operating domain improves. Moreover, it decreases the initial entry cost of building an adaptive application.

IBM’s Autonomic Toolkit [18] 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 [10] for Autonomic Computing, which converts existing log files to the Common Base Event format [11]; 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 Toolsuite is specifically tailored to reducing design and implementation complexity, as well as providing a runtime framework.

Another related research area is *microrebooting* [3], which posits that entering unsafe states in large-scale systems is unavoidable and can be combated by recursively rebooting increasingly larger portions of the system until the unsafe state is cleared. This research is complementary to our J3 work, e.g., JFense provides a framework whereby rebooting logic can be inserted at the component level to enable microrebooting. Moreover, in J2EEML, application designers can specify exactly which components must support rebooting and use Jadapt to automatically weave the required code into those locations.

7 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 since developers must reason about complex sets of QoS assertions and ensure that applications meet them. The J3 Toolsuite described in this paper provides *Model-Driven Engineering* (MDE) tools and an autonomic computing framework, which enable EJB applications to self-manage and maintain their QoS assertions.

To facilitate self-management, the J3 Toolsuite allows developers to capture the structure of EJB applications and their QoS assertions in models 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 could not use introspection to determine whether their QoS assertions are being met. The J2EEML MDE tool helps link assertions and structure by allowing developers to specify this mapping via a DSML. J2EEML also includes mechanisms for modeling and simulating complex EJB structures, interactions, and architectures and using these models to generate code that mirrors the specifications from the model, which frees developers from reinventing complex autonomic software for each new application.

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 can be 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 computing 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 J3 Toolsuite also provides a simulation environment that greatly reduces the complexity of validating developer assumptions about autonomic behavior. This environment uses an automatically generated Prolog knowledge base to enable the simulation to identify QoS assertions that may be decomposable and provide a more flexible system design. As shown in the experiments described in Section 5, this autonomic system simulation and analysis capability greatly improves the ability of the developer to test, validate, and refine autonomic system designs.

The following summarizes our lessons learned thus far by developing and applying the J3 Toolsuite:

- Developing adaptations for EJB applications is hard. Most EJB developers do not think about designing components that can be adapted, swapped, restarted, or reconfigured to handle errors.
- A model of the introspectively measured continuous properties of an autonomic application and how it reacts to them can be simulated

with QSim, but modelers must only specify continuous variables and landmarks directly related to the adaptations. Polluting a model with extraneous information creates a combinatorial explosion for the simulation.

- Creating a model and simulation of an autonomic application greatly enhances the ability of developers to understand the complex behavior that would ordinarily be buried in hundreds of source files.
- Constraint checking and code generation can greatly reduce and/or eliminate hard-to-debug runtime errors, such as JNDI naming errors.
- Adaptive system behavior validation can be simplified by using a modified QDE simulator. A simulation, however, relies on intention entities to predict all the consequences of an adaptive action correctly. In many cases, all the side-effects of an adaptation may not be completely known until the system is built and tested.
- Non-simulatable properties, such as the list of the most recent exceptions thrown by the application, are problematic when predicting the behavior of an autonomic system. New techniques will need to be developed to fully understand how they may affect an autonomic application's behavior.

In future work, we are developing more sophisticated autonomic distributed applications in the domain of shipboard computing and earth science experiments using our J3 Toolsuite. These applications will serve as a testbed for investigating various autonomic architectures. We are also enhancing these tools to increase their simulation capabilities to include the ability to provide weighted intentions of actions. Using weighted intentions will allow the simulation to predict the result of the activation of conflicting actions. Finally, we plan to explore modeling and simulation of adaptive re-deployment of components in response to hardware and software failures.

References

1. Alur D, Crupi J, Malks D (2003) J2EE Core Patterns. Sun Microsystems Press
2. Asikainen T, Männistö T, Soininen T (2004) Representing Feature Models of Software Product Families Using a Configuration Ontology ECAI 2004. Workshop on Configuration
3. Candea G, Kawamoto S, Fujiki Y, Friedman G, Fox A (2004) Microreboot -- A Technique for Cheap Recovery. In: Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI), San Francisco, CA, December
4. Candea G, Fox A (2001) Designing for High Availability and Measurability. In: Proc. of the 1st Workshop on Evaluating and Architecting System Dependability
5. Coulson G, Baichoo S, Moonian O (2002) A Retrospective on the Design of the GOPI Middleware Platform. In: ACM Multimedia Journal
6. Eymann T, Reinicke M (2003) Self-Organizing Resource Allocation for Autonomic Networks. In: Proc. DEXA Workshops
7. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, New York
8. Gray J, Roychoudhury S (2004) A Technique for Constructing Aspect Weavers Using a Program Transformation Engine. In: Proc. of AOSD '04, Lancaster, UK
9. Hatcliff J, Deng W, Dwyer M, Jung G, Prasad V (2003) Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In: Proc. of the 25th International Conference on Software Engineering, Portland, OR
10. IBM, Autonomic Computing Toolkit, www106.ibm.com/developerworks/autonomic/overview.html.
11. IBM Developerworks, Specification: Common Base Event. (www106.ibm.com/developerworks/webservices/library/ws-cbe/).

12. Kang K, Cohen SG, Hess JA, Novak WE, Peterson.SA (1990) Feature Oriented Domain Analysis (FODA) - Feasibility Study. Technical report, Carnegie-Mellon University

13. Kephart JO, Chess DM (2003) The Vision of Autonomic Computing. IEEE Computer, January

14. Ledeczi A, Bakay A, Maroti M, Volgysei P, Nordstrom G, Sprinkle J, Karsai G (2001) Composing Domain-Specific Design Environments. IEEE Computer, November

15. Ledeczi A (2001) The Generic Modeling Environment. In: Proc. Workshop on Intelligent Signal Processing, Budapest, Hungary

16. Loyall J, Bakken D, Schantz R, Zinky J, Karr D, Vanegas R (1998) QoS Aspect Languages and Their Runtime Integration. In: Proc. of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Components

17. Matena V, Hapner M (1999) Enterprise Java Beans Specification, Version 1.1. Sun Microsystems, December

18. Melcher B, Mitchell B (2004) Towards an Autonomic Framework: Self-Configuring Network Services and Developing Autonomic Applications. Intel Technology Journal, November

19. Oppenheimer D, Ganapathi A, Patterson D (2003) Why do Internet services fail, and what can be done about it?. In: Proc. USENIX Symposium on Internet Technologies and Systems, March

20. Tarr P, Ossher H, Harrison W, Sutton SM (1999) N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. 21st International Conference on Software Engineering, May

21. Schmidt D (2006), Model-Driven Engineering, IEEE Computer, Vol. 39. No. 2, February, pp. 41-47

22. White J, Schmidt D (2005) Simplifying the Development of Product-Line Customization Tools via MDD. In: Proc. Workshop: MDD for Software Product Lines, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, October

23. Zinky J, Bakken D, Schantz. R (1997) Architectural Support for Quality of Service for CORBA Objects. In: Theory and Practice of Object Systems, Vol. 3, No. 1
24. Kuipers B, (1986) Qualitative Simulation, In: Artificial Intelligence No. 29, pg. 289-338
25. Bratko I, (2001) Prolog: Programming for Artificial Intelligence, Third Edition. Addison Wesley, New York
26. White J, Schmidt D, Gokhale A (2005) Simplifying the Development of Autonomic Enterprise Java Bean Applications via Model Driven Development. In: Proc. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, October
27. Huebscher M C, McCann J A (2004) Simulation Model for Self-Adaptive Applications in Pervasive Computing. In: Proc. 14th International Conference on Database and Expert Systems Applications, Prague, Czech Republic, September
28. Nagaraja K, Oliveira F, Bianchini R, Martin R, Nguyen T (2004) Understanding and Dealing with Operator Mistakes in Internet Services. In: Proc. 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA, December
29. Kandasamy N, Abdelwahed S, Khandekar M A (2006) Hierarchical Optimization Framework for Autonomic Performance Management of Distributed Computing Systems. In Proc. 26th IEEE International on Distributed Computing Systems, ICDCS 2006. Lisbon, Portugal, July
30. Parnas D L (1972) On the Criteria To Be Used in Decomposing Systems into Modules. In: Communications of the ACM, Vol. 15, No. 12, pp. 1053 – 1058, December
31 Baldwin C, Clark K (1999) Design Rules: The Power of Modularity. MIT Press, Cambridge, MA

Figure Legends

Fig. 1. An Autonomic Architecture for Scheduling Highway Freight Shipments

Fig. 2. J2EEML Remote Interface Composition Model for the *TruckStatusModule*

Fig. 3. J2EEML Structural Model Showing Bean-to-Bean Interactions

Fig. 4. J2EEML Model Associating the *ResponseTime* QoS Assertion with the *RouteTimeModule*

Fig. 5. J2EEML Mapping of QoS Assertions to EJBs

Fig. 6. J2EEML Hierarchical Composition of ResponseTime QoS Assertion J2EEML Hierarchical Composition of *ResponseTime* QoS Assertion

Figure 7: An Association between the *ResponseTime* QoS Assertion and the *ChangeAlgorithms* Adaptation Plan

Figure 8, Developing an Autonomic Application with the J3 Toolsuite.

Figure 9, The JFense Architecture

Figure 10, ContinuousVariable Evolution

Figure 11, Refactoring the RTM's QoS Assertions