Creating Self-healing Service Compositions with Feature Models and Micro-rebooting

J. White*
Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, TN, USA
E-mail: jules@dre.vanderbilt.edu
*Corresponding author

H.D. Strowd
Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, TN, USA
E-mail: harrison.strowd@vanderbilt.edu

D.C. Schmidt
Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, TN, USA
E-mail: schmidt@dre.vanderbilt.edu

Abstract:
Service-oriented architectures (SOAs) are emerging as a powerful mechanism to provide loose coupling and software reuse in enterprise applications. SOAs expose individual reusable software applications or components as remotely accesible services that communicate using standardized message-oriented protocols, such as the Simple Object Access Protocol (SOAP). One possibility that SOAs provide is the ability for applications to heal themselves by failing over to alternate services when a critical application component or service reference fails. The numerous intricate details of identifying errors, releasing resources used to access services, and plan a recovery strategy makes developing applications that can heal by swapping services hard.

Model-driven engineering (MDE) offers a potential solution to handling the complexity of building applications that can heal by swapping services. Existing MDE solutions for building adaptive applications require developers to explicitly model each potential error state and recovery action, which can be extremely complex. Furthermore, developers must then implement the complex recovery actions modeled, which adds significant development complexity. This paper presents a technique based on micro-rebooting that 1) uses feature models to derive a new and correct service composition when a failure occurs; 2) uses an application's component container to shutdown the reference to the failed service; and 3) uses the application container to reboot the subsytem with the new service composition. The paper presents a case study that shows this approach significantly reduces both modeling and healing implementation effort.

1 Introduction

Service-oriented architectures (SOAs) are emerging as a powerful mechanism to provide loose coupling and software reuse in enterprise applications. SOAs expose individual reusable software applications or components as remotely accessible services that communicate using standardized message-oriented protocols, such as the Simple Object Access Protocol (SOAP). The loose coupling provided by message-oriented communication and standardized protocols allows applications to be rapidly composed from both newly developed custom components and from existing services.

Often, within a single organization or group of collaborating organizations, multiple services are available that can accomplish a particular task. The redundancy in services provides the potential to create applications that can heal themselves by failing over to leverage similar services when a service in their service composition (*i.e.* the services used by the application) fails. Failing over to another equivalent but not necessarily identical service can create robust applications that can adapt to service failures and remain functional.

Designing and implementing a mechanism to build self-healing service compositions is a complex endeavor. Since software development projects already have low success rates and high costs, building a service capable of healing is typically not feasible. Furthermore, building adaptive mechanisms greatly increases the complexity of an application and can be difficult to divorce from application code if the development of the adaptive mechanism is not successful.

Model-driven engineering (MDE) provides a potential solution to managing the complexity of developing adaptive services. In an MDE approach, high-level adaptive models are used to generate the complex adaptive code required to heal the application when services fail. This approach allows much of the complex healing code to be generated by the MDE tool and in many cases, removed in needed. Numerous approaches have been presented for building MDE models and platforms for enterprise applications but these approaches tend to suffer from one or more of the following problems:

1. they require tight-coupling between application code and adaptation logic or frameworks

2. they require significant development effort to explicitly model the numerous potential error states and recovery paths from an error state to a correct state

3. they require extensive effort to develop the adaptation action implementations for a realistic application

In this paper we present an MDE approach and toolset, called *Refresh*, for designing and implementing self-healing service compositions. In Section 4, we show that Refresh does not suffer from the above limitations. Refresh is specifically designed for healing a service composition when:

1. the application is implemented with a component-based technology

2. catastrophic failure is imminent

3. the application and any redundant instances in an application cluster cannot continue functioning correctly in their current configuration

4. the application has alternate composable services, that could potentially be exploited to avoid failure

For each potential error state that an application's service composition could enter, most existing MDE adaptation techniques require explicitly modeling both the error state and the numerous actions to transition from the error state to a correct state. For large enterprise applications, there are usually a significant number of potential error states and complex nuanced considerations (*e.g.* availability of other services, database locks held, transaction states, etc.) that make it very difficult to create a model for service composition healing. Rather than explicitly modeling error states and recovery actions, Refresh uses *Feature Models* to capture the rules for determining what is or is not a correct configuration/error state.

Feature models describe an application in terms of points of variability and their affect on each other. For example, in an e-commerce application, a feature might be a service for accessing an order database. The order feature can have different subfeatures, such as different potential services that can serve as the order database access service. If one particular order database access service is chosen, it excludes the other potential order services from being used (it constrains the other features). If the chosen service fails, a new feature selection can be derived that does not include the failed service's feature.

To avoid the challenges and accidental complexities of both modeling all possible error states and paths to correct states, Refresh uses an approach based on *micro-rebooting* [8]. When a failure, such as the inability to communicate with a dependent service, occurs, Refresh 1) uses the application's feature models to derive a new and valid service composition from the currently available services and components; 2) uses the application's component container to shutdown the failing application subsystem (*e.g.* remote reference to a failed service); 3) and restarts the application subsystem in the newly derived configuration (that points to a different service and includes any local components needed to communicate with it).

The remainder of this paper is organized as follows: Section 2 presents the e-commerce application that we will use as a case study throughout the paper; Section 3 illustrates current challenges in applying existing MDE techniques for building adaptive applications to our case study; Section 4 describes Refresh's approach to using feature models and micro-rebooting to reduce the complexity of modeling and implementing an application that can heal; Section 5 presents empirical results obtained from applying Refresh to our case study; Section 6 compares Refresh with related work; and Section 7 presents concluding remarks.

2 Case Study: The Java Pet Store

To illustrate the complexity of applying existing MDE techniques to creating healing applications, we present a case study based on Sun's Java Pet Store e-commerce application [12]. The Pet Store provides a web-based storefront for selling pets. The store includes multiple catetories of pets, products (*e.g.* Bulldog, Iguana), and individual product items (*e.g.* Female Bulldog Puppy). Customers browse for pets and purchase different items.

Sun and other parties use the Pet Store as a reference application to showcase various frameworks, such as the Java 2 Enterprise Edition frameworks [14]. Because the Pet Store is very widely known and can serve as a reference for comparing different technologies, the Pet Store has been re-implemented in different programming languages and with different frameworks. For example, Microsoft has created the .NET Pet Store [3] and the Java Spring Framework [10, 4] has created the Spring Pet Store. The Spring Framework's version of the Pet Store includes support for integrating web services and is the implementation we have chosen for the case study.

Figure 1, presents a high-level feature model of the features related to the Pet Store's data tier. Features are denoted by the various boxes in the diagram. The levels of hierarchy represent subfeatures. For example, all PetStore instances have *DAOs*, Datasources, and *JTA* as subfeatures (the filled circles at the top of the child features denote required features). The Pet Store Java Transaction API (JTA) feature can either be present, denoted when the child *JTAPresent* feature is selected, or not

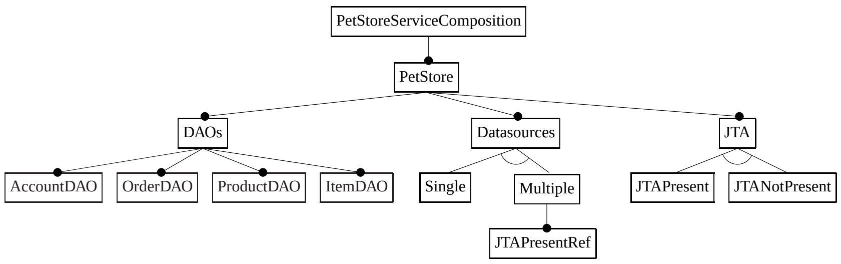

Figure 1: Pet Store Service Composition Feature Model

present. A Feature can also specify rules restricting the selection of other features if the feature is selected. For example, the selection of the *Datasources*/*Multiple* features requires that *JTAPresent* also be selected. This requirement is denoted by the *JTAPresentRef* required feature reference under *Multiple*.

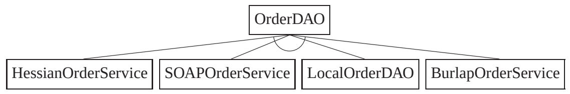

Figure 2: Feature Model of the J2EE Pet Store's OrderDAO

The SpringFramework allows individual components in the Pet Store to be swapped with proxies to remote services. Figure 1 lists the various DAOs that are available in the PetStore. Each of these DAOs can potentially be swapped for a remote service. Figure 2 shows the various options for the OrderDAO. Either the OrderDAO can be implemented by a local component or it can be implemented as a dynamically created Java proxy to a SOAP, Burlap, Hessian, or RMI order service. The case study focuses on failing over from the middle-tier DAOs to different remote services to demonstrate the complexities of applying existing MDE techniques.

3 Challenges of Creating Self-healing Service Compositions

A very common approach to modeling application healing is to model the individual error states that the application can enter and a recovery path (a sequence of recovery actions) to return the application to a correct state. For example multiple MDE approaches use *State Charts* to capture the various error states of an application and the sequences of recovery actions to return to a correct state. Enumerating each potential error state and each recovery path can require significant modeling complexity. As we will show through the rest of this section, even when an MDE tool can generate the majority of the self-healing code for a service composition, the requirement to model and implement recovery actions places a heavy burden on developers.

3.1 Challenge 1: Significant Modeling Complexity to Specify a Recovery Path from an Arbitrary Error State to a Correct State

A healing model must use different error states for each implementation of a service type or component type. The failure of the OrderDAO appears to be a fairly simple error condition to model and specify a recovery path for, but it is not. The problem with modeling each potential error state and recovery path is that the series of recovery actions that need to be invoked is different for the local OrderDAO and remote service implementation. If the local OrderDAO fails, it may simply need to be swapped for another implementation. If a remote service fails, it may be necessary to free resources that were used by a connection to it, such as memory used by caches or network ports.

The type of remote service that is being communicated with can also be important to the recovery action. For example, different recovery paths will be needed to release resources that were used by a connection to a SOAP-based web service as opposed to a Hessian-based web service proxy. Thus, for each type of service or implementation of the OrderDAO, separate error states and recovery paths are needed. Requiring separate error states for each service implementation can cause the number of error states to explode when a real enterprise application is modeled.

If the Pet Store's service composition is modeled using State Charts, as shown in Figure 3, there are 4 different states for each DAO. Futhermore, there are 20 different states needed to represent the potential services and components that can serve as the Pet Store's DAOs. Another property of this model worth noting is that it does not yet include any recovery logic. Instead, the model just includes some placeholder transitions from one potential service to the next.

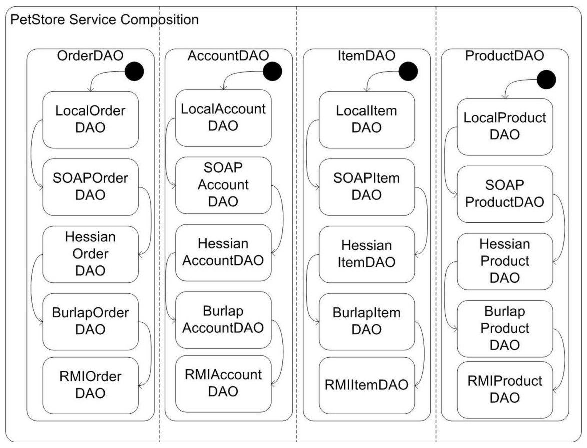

Figure 3: Pet Store Service Composition State Chart

For every error state that the system needs to recover from, the model must explicitly specify a recovery path. For each of the numerous error states that can be produced, as described above, an individual recovery path must be defined to heal the service composition. For example not only do the failure of a Hessian and SOAP-based order service need to be modeled separately, but the series of recovery actions attached to each also needs to be modeled separately. As with error states, the number of recovery path specifications produced for healing each component of an enterprise application can be large.

The Pet Store requires a number of recovery actions to take place in order to swap the service used for a DAO. The various actions for swapping the OrderDAO to one of the remote services is modeled in Figure 4. First, to swap a DAO, a Spring `HotSwappableTargetSource` (an object capable of swapping an active component in the application) must be obtained. Next, any resources held by the old DAO implementation or DAO proxy to a remote service must be released. After releasing resources, a new proxy to another remote service can be created. Finally, the newly created proxy can be swapped into the application using the `HotSwappableTargetSource`. Including the recovery paths in the model ups the total number of states per DAO from 4 to 25.

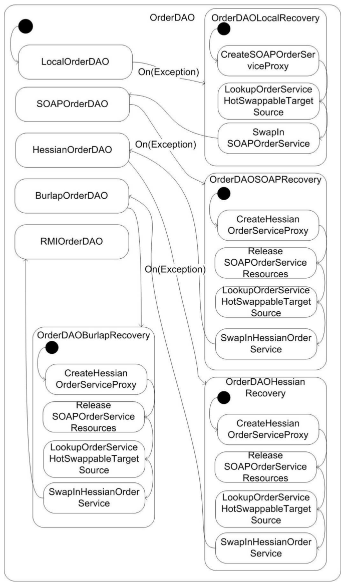
OrderDAO
OrderDAOLocalRecovery
LocalOrderDAO
On(Exception)
CreateSOAPOrderServiceProxy
LookupOrderService HotSwappableTarget Source
SOAPOrderDAO
SwapIn SOAPOrderService
HessianOrderDAO
On(Exception)
OrderDAOSOAPRecovery
BurlapOrderDAO
CreateHessian OrderServiceProxy
RMIOrderDAO
Release SOAPOrderService Resources
On(Exception)
LookupOrderService HotSwappableTarget Source
SwapInHessianOrder Service
OrderDAOBurlapRecovery
CreateHessian OrderServiceProxy
OrderDAOHessian Recovery
Release SOAPOrderService Resources
CreateHessian OrderServiceProxy
LookupOrderService HotSwappableTarget Source
Release SOAPOrderService Resources
SwapInHessianOrder Service
LookupOrderService HotSwappableTarget Source
SwapInHessianOrder Service

Figure 4: OrderDAO Recovery Paths State Chart

Healing a local error may require evaluating the global application state. In the models thus far, if the OrderDAO fails, it can be replaced with any of the potential viable order services. If the Java Transaction API (JTA) is being used to manage transactions, the Pet Store can fail over to any remote service and still provide proper transaction behavior. If, however, JTA is not being used to manage transactions, the system can only provide transactions across a single datasource, meaning that all of the DAOs must be accessing the same database instance. Requiring the use of a single database instance prevents an arbitrary service from being chosen. In the non-JTA situation, the service may only fail over to a remote service if the service is accessing the same database instance as all other referenced remote services.

An extension of the OrderDAO recovery State Chart to include the JTA consideration is show in Figure 5. Each transition to the swap states now includes a guard to ensure that swapping is allowed. A new *GlobalSwapController* has been added to the model to only allow swapping when either JTA is present or a single data source is being referenced by the application's service composition.

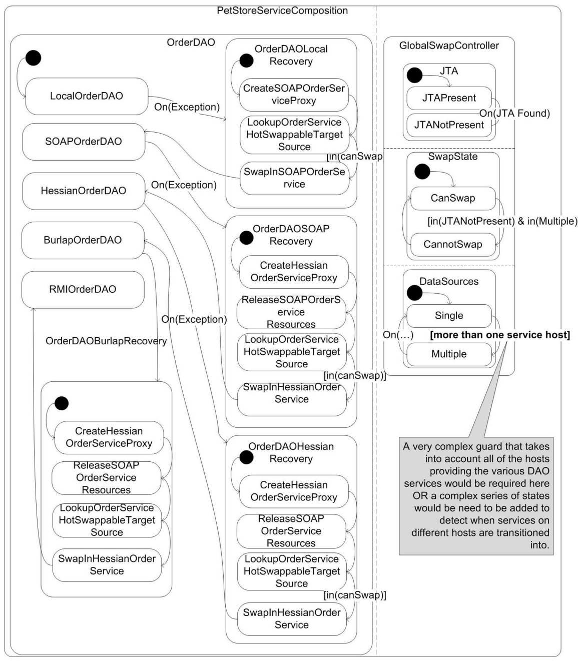
PetStoreServiceComposition
OrderDAO
OrderDAOLocal Recovery
GlobalSwapController
LocalOrderDAO
On(Exception)
CreateSOAPOrderServiceProxy
JTA
JTAPresent
On(JTA Found)
JTANotPresent
SOAPOrderDAO
LookupOrderService HotSwappableTarget Source
[in(canSwap
SwapState
SwapInSOAPOrderService
HessianOrderDAO
On(Exception)
CanSwap
[in(JTANotPresent) & in(Multiple)]
OrderDAOSOAP Recovery
CannotSwap
BurlapOrderDAO
CreateHessian OrderServiceProxy
DataSources
RMIOrderDAO
ReleaseSOAPOrderService Resources
Single
On(Exception)
On(...)
[more than one service host]
OrderDAOBurlapRecovery
LookupOrderService HotSwappableTarget Source
Multiple
[in(canSwap)]
SwapInHessianOrder Service
CreateHessian OrderServiceProxy
ReleaseSOAP OrderService Resources
OrderDAOHessian Recovery
A very complex guard that takes into account all of the hosts providing the various DAO services would be required here OR a complex series of states would be need to be added to detect when services on different hosts are transitioned into.
LookupOrderService HotSwappableTarget Source
CreateHessian OrderServiceProxy
SwapInHessianOrder Service
ReleaseSOAP OrderService Resources
LookupOrderService HotSwappableTarget Source
[in(canSwap)]
SwapInHessianOrder Service

Figure 5: OrderDAO Recovery Paths State Chart when Accounting for JTA

3.2 Challenge 2: Significant Complexity to Write Reconfiguration Code that Can Bring the System from an Arbitrary Error State to a Correct State.

Regardless of the MDE approach used for building the application healing mechanism, developers must always implement the application-specific recovery actions. This requirement parallels the development of enterprise applications and services, where despite the frameworks used, developers are always required to implement the core business logic. Some specialized MDE tools may provide pre-built recovery actions for very specific domains, but in general, nearly every MDE approach requires developers to write the recovery actions.

For each path from an error state to a recovery state, complex recovery logic must be written. The more error states that are possible in the application, the more recovery actions must be written by developers. These numerous recovery actions can be both expensive to develop and difficult to test - a potential problem when projects are already prone to failure and cost overruns.

In the Pet Store application, there are four separate DAOs that can each be swapped to one of four remote services to avoid failures. To implement a simple swapping mechanism in the Pet Store, the Spring framework provides numerous complex utility classes for hotswapping components and connecting to remote services, such as Apache Axis web services. Despite these numerous utility classes, as is shown in Section 5, to create an action to swap just the OrderDAO to one of the four remote services requires 77 lines of Java code to implement the swapping logic and 11 lines of XML code to enable and configure the swapping action in the Pet Store. Although some level of refactoring and object-oriented design can be used to share common logic across actions, implementing each action still requires significant effort.

3.3 Challenge 3: Executing Arbitrary Recovery Actions in Arbitrary Error States can have Numerous Unforeseen Side-effects.

Error states are often specified in such a way that the system as a whole can be in numerous different states that all fall under the definition of the same error state. For example, when the OrderDAO fails, the Pet Store can have orders in progress, category listings in progress, and numerous other nuanced conditions. Building a robust and correct recovery action requires taking into account the side effects of the recovery action on the complex overall state of the application.

For example, what will happen if the local OrderDAO is swapped with a remote service during the submission of one or more customer orders? Can the orders potentially be left in an inconsistent state in the database? Does the safety of the swap depend on whether or not a local or JTA-based transaction mechanism is used? These complex nuanced questions are not easy to answer and must be considered for each recovery action implementation. These intricacies make developing a recovery action that will not lead to unforseen problems hard.

4 Modeling and Building Healing Adaptations with Refresh

By evaluating the challenges in Sections 3.1-3.3, it is apparent that they stem from two causes: 1) the requirement that every error state and recovery path must be explicitly modeled and 2) that developers must implement every complex recovery action. This section describes our MDE toolset, called *Refresh*, that eliminates these two sources of substantial complexity.

Refresh uses feature models to capture the rules for what is a correct system state, which as we will show in Section 4.2, eliminates the need to explicitly model every error state (since each state can be checked for correctness on-demand). Second, rather than requiring complex recovery actions to be implemented, Refresh uses the application's component container to shutdown the application, reconfigure its service composition, and restart the application in the new and correct state. As is shown in Section 5, this reuse of standard container mechanisms for adaptation significantly reduces healing development effort without sacrificing performance.

4.1 Overview of Refresh

Refresh is built around the concept of micro-rebooting. When an error is observed in the application, Refresh uses the application's component container to shutdown and reboot the application's components. Using the application container to shutdown the failed subsystem takes milliseconds as opposed to the seconds required for a full application server reboot. Since it is very likely that rebooting in the same configuration (*e.g.* referencing the same failed remote service) will not fix the error, Refresh derives a new application configuration and service composition from the application's feature models that does not contain the failed features (*e.g.* remote services).

The service composition dictates the remote services used by the application. The application configuration determines any local component implementations, such a SOAP messaging classes, needed to communicate and interact properly with the remote services. After deriving the new application configuration and service composition, Refresh uses the application container to reboot the application into the desired configuration. The overall structure of Refresh is shown in Figure 6.

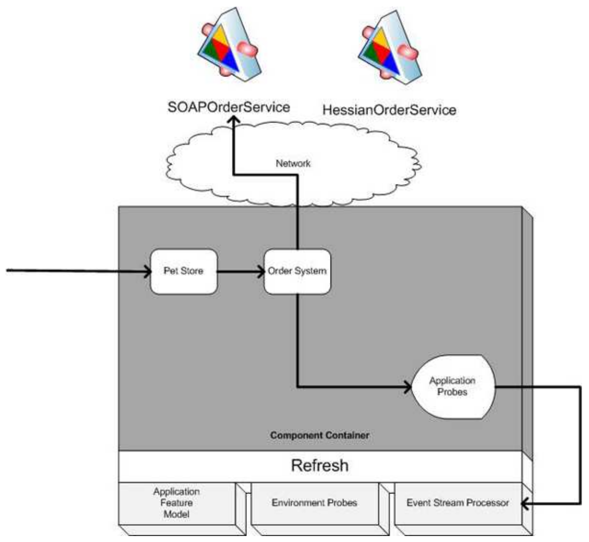

Figure 6: Refresh Structure

Refresh interacts directly with the application container, as can be seen in Figure 6. During the initial and subsequent container booting processes, Refresh transparently inserts *application probes* into the application to observe the application components. Observations from the application components are sent back to an *event stream processor* that runs queries against the application event data, such as exception events, to identify errors. Whenever an application's service composition needs to be healed, *Environment probes* are used to determine available remote services and global application constraints, such as whether or not JTA is present. Finally, Refresh includes a *feature model* of the application that dictates the rules for deriving a new application configuration and service composition when the application needs to be healed and rebooted.

Refresh uses event stream processing [11], to run queries against the application's event data and identify feature failures. The initial implementation of Refresh, based on the Spring Frameworks IoC container, uses the Esper event stream processor [2] for Java. Esper is a high-performance event stream processor that is capable of handling 100,000 events a second with

2,000 queries on a single dual-core CPU [1].

Each feature in the feature model that could potentially fail is associated with a group of event stream queries. At runtime, when a query associated with a feature returns a result, Refresh is notified that the associated feature has failed, as shown in Figure 7. The data and objects observed and analyzed by Refresh are determined by the query specifications.

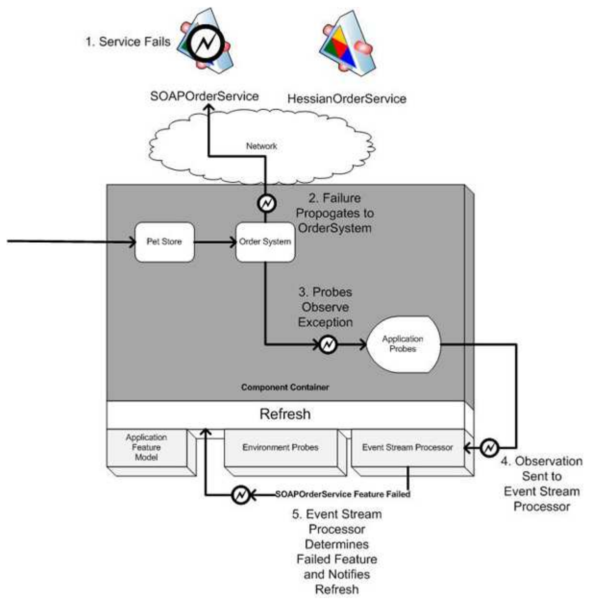
1. Service Fails
SOAPOrderService
HessianOrderService
Network
2. Failure Propogates to OrderSystem
Pet Store
Order System
3. Probes Observe Exception
Application Probes
Component Container
Refresh
Application Feature Model
Environment Probes
Event Stream Processor
4. Observation Sent to Event Stream Processor
SOAPOrderService Feature Failed
5. Event Stream Processor Determines Failed Feature and Notifies Refresh

Figure 7: Error Propogation to Refresh

Once Refresh is notified of a feature failure, it has three main tasks: 1) to use the container to shutdown the application's components; 2) to use the application's feature model to derive a new application configuration and service composition; and 3) to use the container to reboot the application in the new configuration. The sequence of events from a feature failure notification to the rebooting of the container are shown in Figure 8.

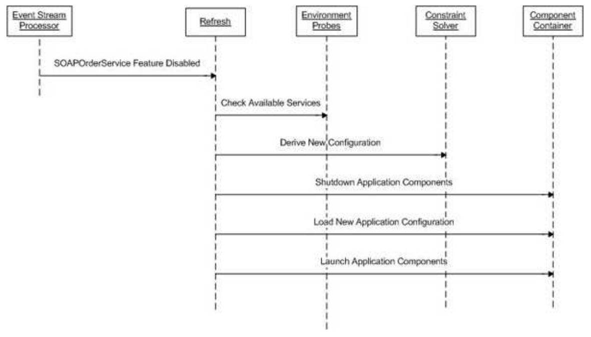

Figure 8: Refresh Reconfiguration, Shutdown, and Launch Recovery Sequence

To derive a new configuration of the application does not include the failed feature, Refresh transforms the feature selection problem into a constraint satisfaction problem (CSP) using techniques that have been developed by us an others in prior work [16, 6, 17]. Once the feature selection problem is transformed into a CSP, a high-performance general purpose constraint solver, such as ILog's JSolver [9], Geocode [13], or Choco [5], is used to derive a new set of features/configuration for the application.

Once the new application configuration and service composition is derived, Refresh invokes the container's shutdown sequence to properly release resources, abort transactions, and perform other critical activities. The new configuration is injected into the container through programmatic calls or by regenerating the application's configuration files [16]. After the configuration is injected into the container, the application is launched in the new configuration without the failed service, as shown in Figure 9.

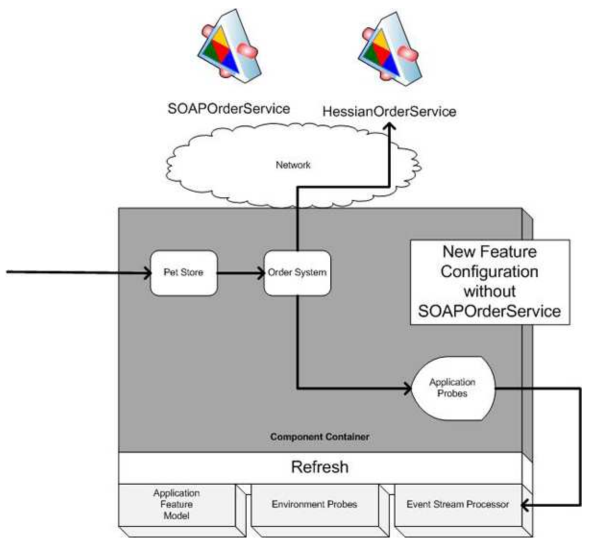

Figure 9: Refresh Launches the Application in the New Configuration

4.2 Solution 1: Use Feature Modeling to Capture the Rules for Deriving what is Considered a Correct State

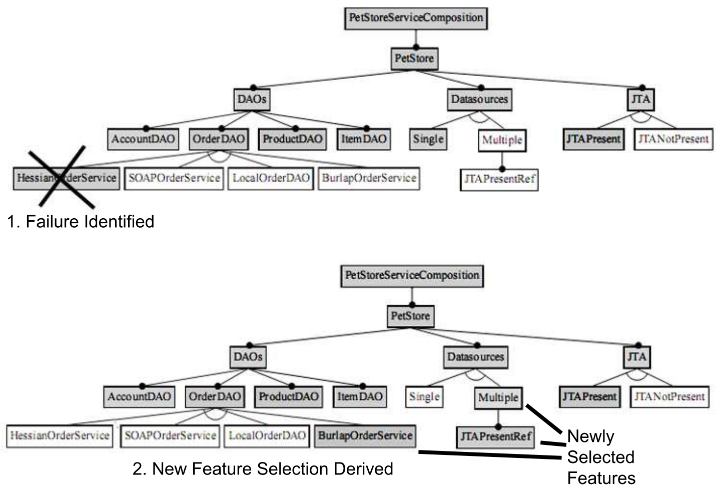

Figure 10: Deriving a new Service Composition from the Pet Store Feature Model

As was pointed out in Section 3.1, modeling each individual error state and recovery path is complex. Refresh uses feature modeling to avoid requiring developers to model each individual error state and recovery path. Feature modeling captures the rules–rather than individual error states and recovery paths–for deriving what constitutes a correct application configuration and service composition. In terms of healing, feature modeling describes:

- the component or service types that are needed to compose the application
- the sets of components or services that can serve as the implementation of a service type in the application's composition
- the rules dictating the requirements, such as dependent libraries, required by each component or service implementation
- the rules constraining how the choice of one service implementation restricts the choices of other component or service implementations in the same application composition

When the failure of a feature is observed, Refresh uses the feature model of the application to derive an alternate set of features for the application that does not include the failed feature. For example, in the Pet Store, when the *LocalOrderDAO* feature fails, Refresh uses the feature model to derive an alternate feature selection for the Pet Store. In the example shown in Figure 10, Refresh chooses a new feature selection that uses the *HessianOrderDAO* rather than the failed *LocalOrderDAO*.

Automated Feature Selection Using a Constraint Solver: The key to Refresh's healing capabilities is its ability to use a constraint solver to automatically derive a new feature selection for the application. Prior work provides extensive details on the process for transforming a feature selection problem into a constraint satisfaction problem (CSP), which is the input format of a constraint solver, and deriving a feature selection. In this section, we briefly cover this mapping.

A constraint satisfaction problem is a series of variables and a set of constraints over the variables. For example, "$A+B<C$" is a constraint satisfaction problem over the integer variables A, B, and C. A constraint solver automatically derives a correct labeling (values for the variables). The labeling "$A=1, B=2, C=4$" is a correct labeling of the example CSP.

A selection of features from a feature model can be represented by a set of integer variables with domain 0 or 1. Each variable represents a unique feature from the feature model. If the variable representing the *HessianOrderService* is represented by the variable V_1, then $V_1 = 1$ in a labeling of a feature selection CSP means that the feature is selected in the solution. If the labeling contains $V_1 = 0$, it implies that the feature is not selected in the solution. The configuration of an application and its service composition is represented as a set of these variables that denote which services and application components are enabled in a configuration.

Rules dictating the proper composition of the services are specified as constraints over the V_i variables. For example, since only one of *HessianOrderService* and *SOAPOrderService* can be used at a time by the Pet Store, a constraint can be used to capture this rule. Let, V_2 be the variable representing the *SOAPOrderService*. This rule is specified as the constraint $V_1 = 1 \rightarrow V_2 = 0$. As described in [16], complex rules, such as memory constraints, can be described using a CSP.

When a feature is flagged as failed, Refresh adds a new constraint to the feature selection process preventing the failed feature from being selected (*e.g.*, $V_i = 0$). Refresh then uses a the constraint solver to derive a new feature selection that can be used by the application based on the environmental constraints (*e.g.* JTA vs. No JTA) and feature model composition constraints (*e.g.*, only one of the order services may be selected at a time). When only standard feature modeling rules, like excludes, requires, cardinality, and attribute values are used to describe constraints, the solver can very quickly produce a correct solution [17]. More complex constraints, such as memory resource constraints, can be added to augment standard feature modeling rules but require more care in the feature model specification process to allow the solver to quickly derive a solution [17].

Eliminating Error State and Recovery Path Modeling Complexity: Because the new feature selection is introduced into the application by shutting down the old references to remote services and launching the new component configuration and service composition, separate recovery actions are not needed. Furthermore, since feature models specify the rules for deriving a correct/incorrect configuration and do not enumerate all possible error configurations, they require significantly fewer modeling elements. As we will show in Section 5, the equivalent healing behavior to the 111 state State Chart described in Section 3.1 can be produced in Refresh using a feature model with 33 features –a roughly 70% reduction in total model elements. The feature models also have 33 connections versus the 102 connections for the State Chart.

4.3 Reusing the Component Container's Shutdown/Configuration/Launch Mechanisms for State Transitions

Sections 3.2-3.3 illustrated the complexity and large development burden of writing recovery actions to heal an application by failing over to alternate services. Refresh attacks the problem with a combination of code reuse and automation. Refresh reuses an application container's ability to shutdown an application's components, reconfigure the components (*i.e.* create the newly desired service composition), and launch the application in the new state (*i.e.* transition the application into the new service composition state). By reusing existing mechanisms that are well-tested and trusted by developers, the need to write custom recovery actions is eliminated.

Second, since rebooting in the same application configuration with the same service composition is unlikely to fix a failed reference to a service, Refresh automatically derives a new and valid application configuration and service composition. This automated approach to deriving a new service composition from an application's feature model allows micro-rebooting to be applied to service composition healing. Normally, with a manual recovery action implementation process, developers would deduce the correct states to transition the application into and implement the transition logic. Refresh's automated derivation process eliminates the need for a developer to: 1) determine where to transition to, 2) decide how to accomplish the transition, and 3) implement the transition.

Container Rebooting-based Healing Reduces Potential Unintended Side-effects: A key benefit of using the container's built in component management mechanisms for state transitions is that they are guaranteed to bring the non-persistent application state to the desired correct state. This guarantee helps to resolve the problems outlined in Section 3.3 of having to deal with the potential of unintended side-effects from recovery actions.

With Refresh, the application container shuts down components, which releases resources and resets in-memory state, and then re-launches the application with a clean memory state. With recovery actions, there is the potential that one or more of the affects on the application will have unforeseen consequences to the non-persistent in-memory application state. These unforseen side-effects are not possible with a container rebooting approach that resets non-persistent state.

A container rebooting approach does not eliminate the possibility that persistent application state, such as database rows, will not be placed into an inconsistent state. The approach does, however, have a number of properties that make this scenario far less likely than a recovery action approach. First, all components typically *must* implement lifecycle methods that are called by the container to manage the component. If a component does not properly handle persistent state on shutdown, it is a flaw in the implementation of the component that could emerge–even if the application never uses healing mechanisms.

Second, most enterprise applications maintain the consistency of persistent application state through transactions. Furthermore, most enterprise applications use container-managed persistence APIs, such as JTA. Even the Non-JTA examples provided for the Pet Store still use an alternate container-managed persistence API that works across only a single datasource. When the container is used to as the healing transition mechanism, any transactions that are in process will be properly rolled back or committed by the container during the healing of the application's service composition.

5 Applying Refresh to the Java Pet Store

To compare the development effort of including recovery actions into the Pet Store, we implemented three versions of the Spring Pet Store that provided the ability to swap failed DAOs with remote services and to swap from failed remote services to other remote services (the modifications for the three implementations are available from [15]). One implementation was produced using a purely manual approach that used Spring's proxying and aspect infrastructure to implement the monitoring of the DAOs and Spring *HotSwappableTargetSources* to swap remote services on-the-fly. The second implementation was produced assuming an MDE tool was provided that could model the error states and recovery actions for the Pet Store and generate the required monitoring code and recovery path logic but not the implementations of the recovery actions. We refer to this MDE approach as the *MDE error state/recovery path* approach. Finally, a third implementation was produced using Refresh.

Manual Implementation: The top table in Figure 11 shows the results of the initial implementation efforts. The manual approach required implementing two key classes a *ServiceSwapper* capable of 1) looking up the Spring HotSwappableTargetSource for a DAO; 2) connecting to a Hessian, Burlap, SOAP, or RMI remote service; and 3) swapping in the new service for the failed component/service. As is shown in the results figure, the class required 77 lines of code. The second class implemented was a Spring MethodInterceptor that was used to monitor each invocation on a DAO or remote service for Exceptions and call the appropriate ServiceSwapper when an Exception occurred. This class required 20 lines of code. Finally, the components were included in the Pet Store by adding them to the XML configuration files for the Pet Store, which required adding 96 lines of XML code.

MDE Error State / Recovery Path Implementation: The analysis for the MDE error state/recovery path approach was based on a generic model of the minimum effort that would be required for any MDE adaptation modeling tool and framework that did not provide Spring-specific recovery action implementations. There are no MDE frameworks that we are aware of that have Spring-specific recovery action implementations that could accomplish the required swap. For the MDE analysis, we measured only the lines of code required to implement the ServiceSwapper and to integrate the needed ServiceSwappers into the configuration files of the Pet Store. We asssumed that all of the logic for choosing the correct ServiceSwapper to execute, the implementation of the MethodInterceptor, and all configuration code required to integrate the method interceptors and their dependent proxies into the configuration file would be generated by the tool.

The MDE error state/recovery action approach used the State Charts presented in Section 3.1. The full State Chart healing specification requires 111 states and 102 transitions between states. As can be seen in Figure 11, the MDE approach still requires 77 lines of code to implement the ServiceSwapper recovery action but eliminates the 31 lines of code needed to implement the recovery path execution logic and the 20 lines of code required for the monitoring implementation. Furthermore, an MDE approach reduces the lines of XML configuration code that must be added from 96 to 44 (a roughly 54% savings).

Refresh Implementation: Finally, we implemented the swapping capabilties in the Pet Store using Refresh. Refresh's use of Feature models required a total of 21 model elements (features) versus the MDE approache's 32 model elements (16 error states and 16 recovery actions associated with the error

	Manual	MDE State/Action	Refresh
Initial Implementation			
Modeled Error States or Features	0	16	21
Modeled Recovery Actions	0	16	0
Implement Introspection	20	0	0
Model Error Identification	0	16	16
Implement Recovery Actions	77	77	0
Implement Recovery Path Chooser	31	0	0
Configuration Modifications	96	44	67
Sub Totals	224	169	104
Extend Implementation for Bandwidth Consideration			
Modeled Error States or Features	0	0	17
Modeled Recovery Actions	0	0	0
Implement Introspection	0	0	0
Implement Error Identification	0	0	0
Implement Global Recovery Coordinatc	47	47	0
Implement Recovery Path Chooser	31	0	0
Configuration Modifications	6	6	0
Sub Totals	84	53	17
Totals	308	222	121

Figure 11: Comparing Implementation Effort for the Healing Pet Store

states). Refresh also required 16 lines of code to specify the Esper queries over the event stream of the Pet Store to map queries to the failure of one of the 16 features. Refresh's use of the container's built-in shutdown/configuration/launch mechanisms for healing, eliminated the need to implement the code for the ServiceSwapper. Refresh automatically generates the required monitoring code for the Pet Store (this was assumed for the other MDE approach as well). Refresh did require 23 more lines of code to be modified in the configuration file of the Pet Store versus the other MDE approach. These extra lines of configuration code are a result of adding the Refresh annotations dictating how to dynamically modify the application's configuration based on a feature selection.

Overall, the Refresh approach required 4̃8% less implementation effort than the other MDE approach and 5̃4% less than the manual approach. The error state/recovery action approach required roughly 2̃5% less effort than the manual approach. We would expect this percentage to be larger for scenarios requiring more complex monitoring code. The manual approach also benefited from the extensive proxying and hotswapping capabilities of Spring, which substantially reduced its total development cost.

Refresh Performance: We used Apache JMeter to simulate the concurrent access of 40 different customers to the Pet Store and the time required to complete 4,000 orders. We simulated the failure of different DAOs to force Refresh to heal the Pet Store by swapping remote services for the failed DAOs. After the DAOs were swapped to remote services, we iteratively shutdown the services used by the Pet Store to force the failover to alternate remote services. Over the tests, Refresh averaged 151ms from the time an exception indicating a failure was observed until the Pet Store was reconfigured and rebooted with a new service composition. These times were obtained by running the Pet Store on a 2.0ghz Intel Core DUO on Windows XP with 2 gigabytes of RAM. The average time required by the constraint solver to derive a new feature selection was 12ms. These times indicate that Refresh can provide high-performance application healing while reducing modeling and implementation effort.

6 Related Work

Other autonomic web services approaches, such as the system proposed by Birman et al. [7], require predefined invalid states to be identified. Birman et al. developed a tool known as Astrolabe that stores the state of a set of resources in a hierarchy of tables. Astrolabe uses SQL queries to determine if the system has reached an invalid state. Once an invalid state has been identified, the application can be healed. This approach suffers from the five challenges outlined in Section 3. First, separate queries must be specified to identify the failure of individual service and component type implementations, complicating the healing model. Furthermore, even once an error has been identified, complex recovery actions must be devised to rectify the error. As we outlined in Section 4, the reliance on feature models and application container rebooting eliminates the need to model error states for each implementation and write complex recovery actions.

Microrebooting, a process developed by Candea et al., is a technique used to restart only the component, or collection of components in which the failure occurred. In this way they minimize the number of requests lost and the performance cost of each reboot. The problem with this technique is that the system may potentially be left in an inconsistent state following a reboot, because while the newly rebooted components are in their initial state, the rest of the system is midway through its services. Reboots in refresh are based on the same set of rules used to initialize the system. In this way refresh is ensured to reboot to a consistent state each time.

Garlan and Schmerl use a model based approach to self healing that requires the system to monitor the run time behavior of each component and handle each component failure individually. This requires the system to have a specific path from a set of invalid states to valid states. Refresh uses its initialization procedure to reboot the system in a valid state. In this way refresh greatly decreases the overhead required to define the parameters of the system.

Kephart and Chess identify the challenge of testing self healing applications due to the large number of diverse states that the application may find itself in. Refresh is easily tested by verifying the set of rules used to determine valid states. This set of rules is used each time the system is configured or reconfigured. In this way the system is guaranteed to be configured to satisfy these rules. Thus the accuracy of the system depends completely on the validity of the rules used to specify the set of valid states.

7 Concluding Remarks

In Section 3, we showed that modeling each potential error state and recovery action for an application's service composition is complex. Using State Charts, the Java Pet Store requires 111 states with 102 transitions to capture a very simple failover process from the Pet Store's middle-tier data access objects to remote services. Furthermore, modeling recovery actions requires developers to implement the recovery actions. A simple failover action to swap a remote service for one DAO in the Pet Store requires roughly 77 lines of code.

Section 4 illustrated that by using feature models and application container rebooting, our MDE approach, called Refresh, eliminated the need to model each potential error state and recovery path. Furthermore, Section 5 presented empirical results that demonstrated a 70% reduction in total modeling elements required for Refresh's feature model approach versus MDE approaches that require modeling each individual error state and recovery path. Furthermore, Refresh also provided a 61% reduction in service composition healing implementation effort by eliminating the need to create recovery actions. Refresh is an opensource project available as part of the *GEMS Model Intelligence* project at http://www.eclipse.org/gmt/gems.

REFERENCES

[1] Esper faq, http://esper.codehaus.org/tutorials/faq_esper/faq.html#performance.

[2] Event stream intelligence with esper and nesper. http://esper.codehaus.org.

[3] .NET Pet Store, http://msdn2.microsoft.com/en-us/library/ms978487.aspx.

[4] The Spring Framework, http://www.springframework.org/about.

[5] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using Java CSP solvers in the automated analyses of feature models. *Post-Proceedings of The Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*.

[6] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated Reasoning on Feature Models. *17th Conference on Advanced Information Systems Engineering (CAiSEŠ05, Proceedings), LNCS*, 3520:491–503, 2005.

[7] K. Birman, R. van Renesse, and W. Vogels. Adding high availability and autonomic behavior to Web services. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 17–26, 2004.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot-a technique for cheap recovery. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, 2004.

[9] A. Chun. Constraint programming in Java with JSolver. *Proc. Practical Applications of Constraint Logic Programming, PACLP99*, 1999.

[10] R. Johnson and J. Hoeller. *Expert one-on-one J2EE development without EJB*. Wrox, 2004.

[11] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001.

[12] S. Microsystems. Java Pet Store Sample Application.

[13] C. Schulte and P. Stuckey. Efficient constraint propagation engines. *Arxiv preprint cs.AI/0611009*, 2006.

[14] T. Valesky. *Enterprise JavaBeans*. Addison-Wesley Reading, MA, 1999.

[15] J. White. Healing pet store case study implementation. http://www.dre.vanderbilt.edu/ jules/petstore-casestudy-code.zip,
2007.

[16] J. White, K. Czarnecki, D. C. Schmidt, G. Lenz, C. Wienands, E. Wuchner, and L. Fiege. Automated Model-based Configuration of Enterprise Java Applications. In *EDOC 2007*, October 2007.

[17] J. White, A. Nechypurenko, E. Wuchner, and D. C. Schmidt. Optimizing and Automating Product-Line Variant Selection for Mobile Devices. In *11th International Software Product Line Conference*, September 2007.