Creating self-healing service compositions with feature models and microrebooting

Jules White*, Harrison D. Strowd and Douglas C. Schmidt

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

Abstract: Service-oriented architectures (SOAs) provide loose coupling and software reuse in enterprise applications. SOAs enable 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 and planning a recovery strategy make it hard to develop applications that can heal by swapping services.

Model-driven engineering (MDE) offers a potential solution to handling the complexity of building applications that can heal by swapping services. This paper presents an MDE technique called Refresh that is based on microrebooting and uses

1 feature models to derive a new and correct service composition when a failure occurs
2 an application's component container to shutdown the reference to the failed service
3 the application container to reboot the subsystem.

We also present the results from a case study that shows Refresh significantly reduces both modelling and healing implementation effort.

Keywords: feature modelling; service healing; reconfiguration; constraint satisfaction; service oriented architectures; SOAs; autonomic systems; enterprise Java Beans; model-driven development; microrebooting; self-healing; service composition.

Reference to this paper should be made as follows: White, J., Strowd, H.D. and Schmidt, D.C. (xxxx), 'Creating self-healing service compositions with feature models and microrebooting', *Int. J. Business Process Integration and Management*, Vol. X, No. Y, pp.000–000.

Biographical notes: Jules White is a Research Assistant Professor in the Electrical Engineering and Computer Science Department at Vanderbilt University. He received his BA in Computer Science from Brown University, his MS and PhD from Vanderbilt University. His research focuses on applying search-based optimisation techniques to the configuration of distributed, real-time and embedded systems. In conjunction with Siemens AG, Lockheed Martin, IBM and others, he has developed scalable constraint and heuristic techniques for software deployment and configuration. He is the Project-Lead of the Eclipse Foundation's Generic Eclipse Modelling System (GEMS http://www.eclipse.org/gmt/gems).

Harrison Strowd is a Graduate Student at Carnegie Mellon University, pursuing Master of Science in Information Technology – Software Engineering. He received his BS in Computer Science and Math from Vanderbilt University. His prior research focused on feature selection algorithms in the context of large-scale product lines. Currently, he is researching the application of probabilistic model checkers to data-intensive distributed computing systems.

Douglas C. Schmidt is a Full Professor in the Electrical Engineering and Computer Science Department and Associate Chair of the Computer Science and Engineering program at Vanderbilt University, Nashville, TN. During the past two decades he has led pioneering research on patterns, optimisation techniques and empirical analyses of object-oriented and component-based frameworks and model-driven development tools that facilitate the development of middleware and applications for distributed real-time and embedded (DRE) systems. He is an expert on DRE computing patterns and middleware frameworks and has published over 400 technical papers and nine books that cover a range of topics including high-performance communication software systems, parallel processing for high-speed networking protocols, quality-of-service (QoS)-enabled distributed object computing, object-oriented patterns for concurrent and distributed systems model-driven development tools. He received his PhD in Computer Science from the University of California, Irvine in 1994. (URL: www.dre.vanderbilt.edu/~schmidt).

1 Introduction

Organisations are rapidly deploying service-oriented architectures (SOAs) that create loosely coupled and highly reusable application components through the use of standardised message-oriented protocols, such as the Simple Object Access Protocol (SOAP). Often, within a single organisation or group of collaborating organisations, 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 complex. Since, software development projects already have low success rates and high cost, building a service capable of healing is hard (Barki et al., 1993). Moreover, building adaptive mechanisms greatly increases application complexity and can be hard to decouple 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 MDE tools to generate much of the complex healing code and in many cases, remove the healing code if it does not function properly. Although, numerous approaches (Joshi et al., 2005; Bhat et al., 2006; Calinescu, 2007; Denaro et al., 2007) have been devised to build MDE models and platforms for enterprise applications, these approaches tend to suffer from one or more of the following problems:

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

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

This paper presents an MDE approach and toolset called *Refresh*, for designing and implementing self-healing service compositions that addresses the limitations outlined above. Refresh is specifically designed for healing a service composition when

1 the application is implemented with a component-based technology, such as enterprise Java Beans or the CORBA Component Model

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.

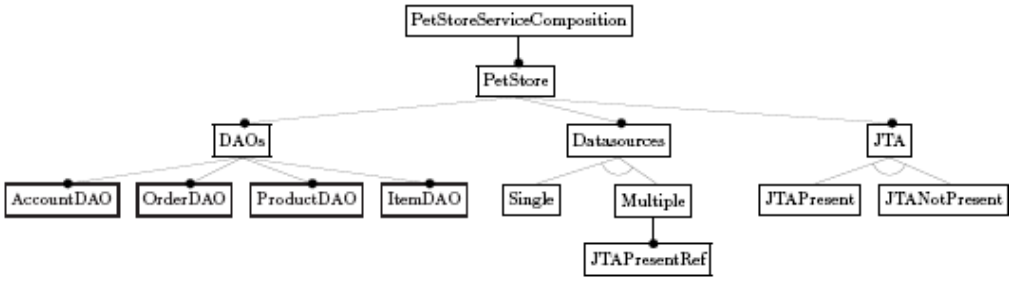
For each potential error state that an application's service composition could enter, conventional MDE adaptation techniques (Joshi et al., 2005; Bhat et al., 2006; Calinescu, 2007, Denaro et al., 2007) require explicitly modelling both the error state and the numerous actions to transition from the error state to a correct state. For large enterprise applications, moreover, there are usually a significant number of potential error states and complex nuanced considerations, such as availability of other services, database locks held and transaction states. These considerations make it hard to create a model for service composition healing. Rather than explicitly modelling error states and recovery actions, Refresh uses feature models (Kang et al., 1998) 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 effect 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.

This paper provides the following contributions to the study and development of self-healing service compositions:

- It shows how when a failure occurs (such as the inability to communicate with a dependent service) Refresh uses the application's feature models to derive a new and valid service composition from the currently available services and components, which eliminates the need to model every potential error state and recovery action.
- It describes Refresh's use of an approach based on *microrebooting* (Candea et al., 2004), which is a technique for rebooting a small set of failed components rather than an entire application server, to shutdown the failed service composition and launch the newly derived composition, eliminating the need for developers to implement recovery actions.
- It presents empirical results from a case study applying Refresh to an e-commerce application that shows Refresh provides a ~55% decrease in modelling complexity and ~60% decrease in implementation cost versus other MDE approaches for building self-healing service compositions.

Figure 1 Pet Store service composition feature model



The remainder of this paper is organised as follows: Section 2 presents the e-commerce application that we will use as a case study throughout the paper; Section 3 enumerates 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 microrebooting to reduce the complexity of modelling and implementing an application that can heal; Section 5 analyses 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 show the complexity of applying conventional MDE techniques to creating healing applications, we present a case study based on Sun's Java Pet Store e-commerce application (Sun Microsystems n.d.). The Pet Store provides a web-based storefront for selling pets. The store includes multiple categories of pets, products (e.g., bulldog and iguana) and individual product items (e.g., female bulldog puppy). Customers browse for pets and purchase different items.

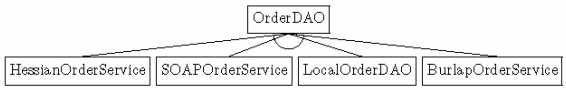
Sun and other parties use the Pet Store as a reference application to showcase various enterprise Java technologies. Since the Pet Store application is widely known and can serve as a reference for comparing different technologies, the Pet Store has been re-implemented in different programming languages and with different frameworks. For example, the Java Spring Framework (Johnson and Hoeller, 2004) 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 Pet Store 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 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*.

The Spring Framework allows the swapping of individual components in the Pet Store with proxies to remote services. Figure 1 lists the various DAOs that are available in the Pet Store. Each DAO 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.

Figure 2 Feature model of the J2EE Pet Store's order-DAO



3 Challenges of creating self-healing services

A common approach (Joshi et al., 2005; Lapouchnian et al., 2005; Barbier, 2006; Bhat et al., 2006; Elkorobarrutia et al., 2006; Calinescu. 2007; Denaro et al., 2007) to modelling 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 (Lapouchnian et al., 2005; Barbier, 2006; Elkorobarrutia et al., 2006) use s*tatecharts* (Harel et al., 1987) 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 modelling complexity. This section shows how, 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

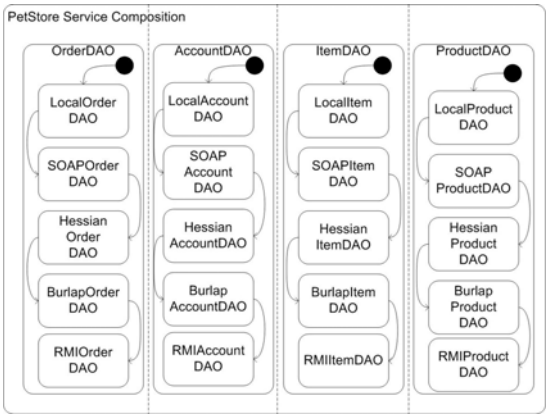
A healing model must use different error states for each implementation of a service type or component type

The failure of the OrderDAO seems like a fairly simple error condition to model and specify a recovery path for, but it is not. The problem with modelling each potential error state and recovery path is that the series of recovery actions that must be invoked is different for the local OrderDAO and remote service implementation.

For example, if the local OrderDAO fails, it may be swapped for another implementation. If a remote service fails, it may be necessary to free resources, such as memory used by caches or network ports, that were used by a connection to it. Services connected through different protocols also need separate error states to associate their unique recovery actions with.

If the Pet Store's service composition healing is modelled using statecharts, as shown in Figure 3, there are four different states for each DAO. To increase readability, Figure 3 does not include events and guards on transitions, which further complicate the model. There are 20 different states needed to represent the potential services and components that can serve as the Pet Store's DAOs.

Figure 3 Pet Store service composition statechart

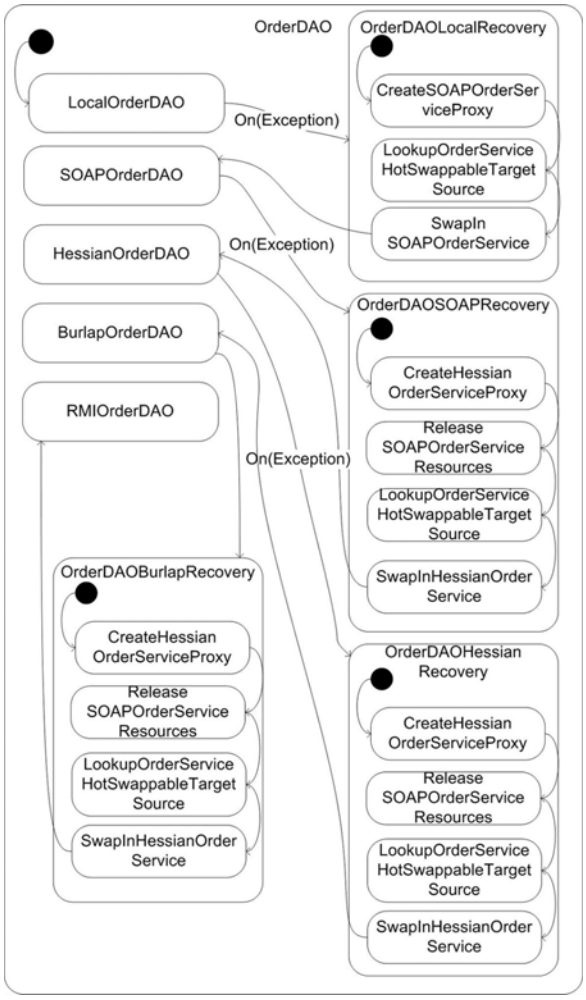


For every error state that the system needs to recover from, the model must explicitly specify a recovery path

For example not only should the failure of a Hessian and SOAP-based order service be modelled separately, but the series of recovery actions attached to each also should be modelled 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 to swap the service used for a DAO. The various actions for swapping the OrderDAO to one of the remote services is modelled 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 four to 25.

Figure 4 OrderDAO recovery paths statechart



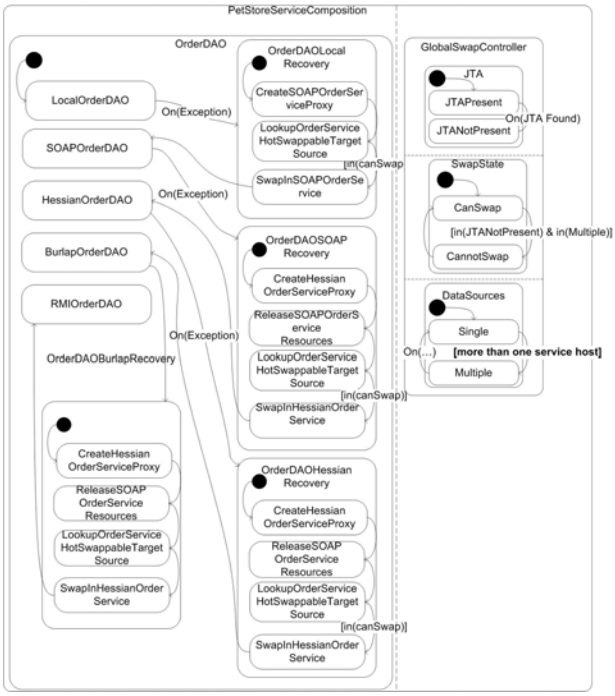
Healing a local error may require evaluating the global application state

For example, if the JTA is being used to manage transactions, the Pet Store can fail over to any remote service and still provide proper transaction behaviour. If JTA is not being used to manage transactions, however, the system can only provide transactions across a single data source, meaning that all 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 statechart to include the JTA consideration is shown 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. Section 4.2 shows how Refresh uses feature modelling and other techniques to eliminate the need to model every potential error state and recovery action.

Figure 5 OrderDAO recovery paths statechart 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 to build 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 specialised MDE tools may provide pre-built recovery actions for 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 hard to test, which can become problematic 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 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. Section 4.3 shows how microrebooting can significantly reduce this substantial development burden by loading a new service composition derived by a constraint solver.

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? 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 unforeseen problems hard. Section 4.3 shows how using microrebooting as the basis for recovery eliminates many of these hard to predict recovery side-effects and also provides a more well understood state transition mechanism.

4 Developing healing adaptations with Refresh

The challenges in Sections 3.1–3.3 stem from two causes:

1 the requirement that every error state and recovery path must be modelled explicitly

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.

4.1 Overview of Refresh

Refresh is based on the concept of microrebooting (Candea et al., 2004). 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 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.

Refresh interacts directly with the application container, as shown 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.

Refresh uses event stream processing (Luckham, 2001) to run queries against the application's event data and identify feature failures. The initial implementation of Refresh, based on the Spring Framework's IoC container, uses the Esper event stream processor (*Event Stream Intelligence with Esper and NEsper, http://esper.codehaus.org* n.d.) 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 (*Esper FAQ, http://esper. codehaus. org/tutorials /faq esper/ faq.html#performance* n.d.).

Figure 6 Refresh structure (see online version for colours)

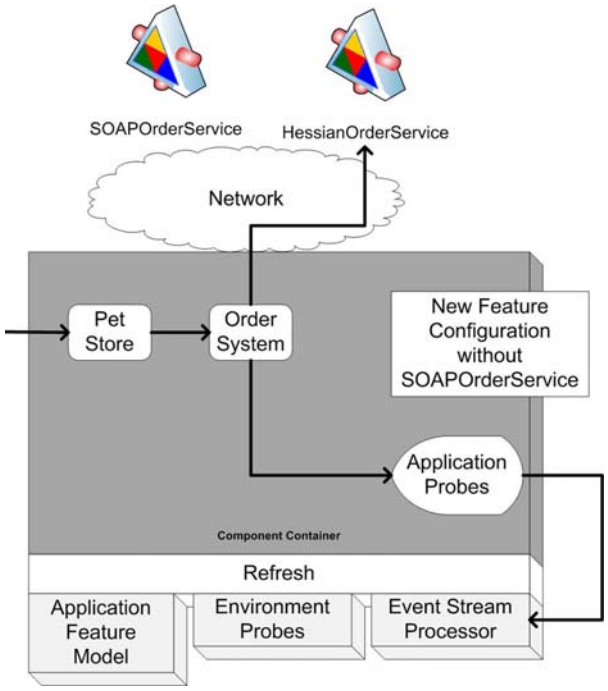
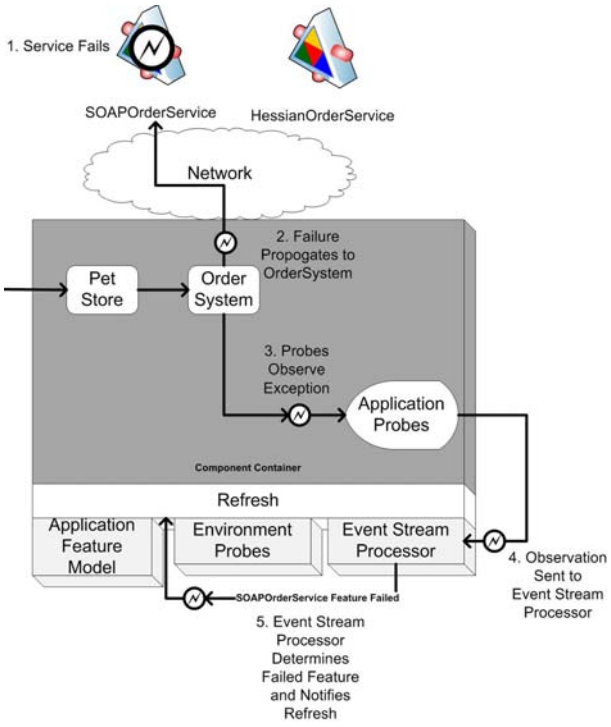


Figure 7 Error propagation to Refresh (see online version for colours)



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 analysed by Refresh are determined by the query specifications.

Once Refresh is notified of a feature failure, its three main tasks are to use

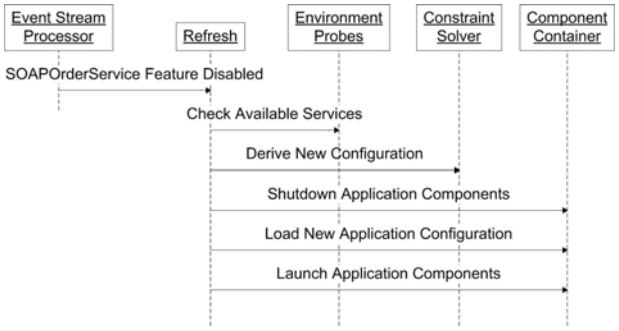
1 the container to shutdown the application's components

2 the application's feature model to derive a new application configuration and service composition

3 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.

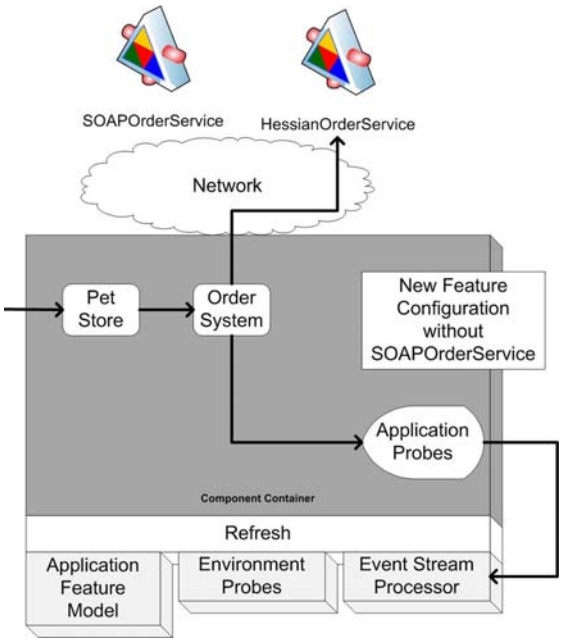
Figure 8 Refresh reconfiguration, shutdown and launch recovery sequence



To derive a new configuration of the application that 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 (Benavides et al., 2005, White, et al., 2007a, 2007b). Once the feature selection problem is transformed into a CSP, a high-performance general purpose constraint solver, such as the Java Choco (Benavides et al., 2007) solver, is used to derive a new set of features/configuration for the application.

After 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 (White et al., 2007a). 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 (see online version for colours)



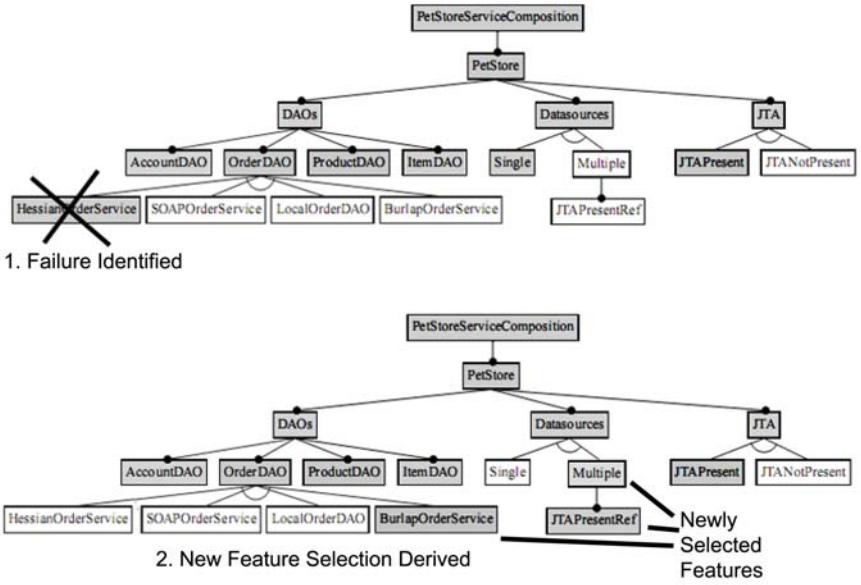
4.2 Use feature modeling to capture the rules for deriving what is considered a correct state

As discussed in Section 3.1, modelling each individual error state and recovery path is complex. Refresh uses feature modelling to avoid requiring developers to model each individual error state and recovery path. Feature modelling 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 modelling 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 *BurlapOrderService* rather than the failed *SOAPOrderService*.

Figure 10 Deriving a new service composition from the Pet Store feature model



Automated feature selection using a constraint solver

The key to Refresh's healing capabilities is its ability to use a constraint solver (Cohen, 1990) to derive a new feature selection for the application automatically. Prior work (Benavides et al., 2005, 2007; White et al., 2007a) provides extensive details on the process for transforming a feature selection problem into a CSP (Cohen, 1990), which is the input format of a constraint solver and deriving a feature selection. We briefly cover this mapping below.

A CSP is a series of variables and a set of constraints over the variables. For example, '$A + B < C$' is a CSP over the integer variables A, B and C. A constraint solver automatically derives a correct labelling (values for the variables). The labelling '$A = 1$, $B = 2$, $C = 4$' is a correct labelling 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 labelling of a feature selection CSP means that the feature is selected in the solution. If the labelling 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 (White et al., 2007a), 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).

4.3 Reusing the component container's shutdown/configuration/launch mechanisms for state transitions

Sections 3.2–3.3 show 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. In particular, it 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.

Moreover, 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 microrebooting 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 developers to:

1 determine where to transition to

2 decide how to accomplish the transition

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 guarantees help to resolve the problems outlined in Section 3.3 of dealing 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 relaunches 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 unforeseen 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. Moreover, 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 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 the following three versions of the Spring Pet Store with self-healing service compositions.

- The first 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.
- The third implementation was produced using Refresh, which captures the rules for configuring the application and its service composition in feature models and uses microrebooting to eliminate the need to implement recovery actions.

The self-healing for all three implementations was built around 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 (White, 2007).

Manual implementation

The top table in Table 1 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

3 swapping in the new service for the failed component/service.

As shown in Table 1, 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 modelling tool and framework that did not provide Spring-specific recovery action implementations. The models were built using statecharts, since it is arguably the most widely used and mature state modelling language. statecharts also have a number of powerful concepts, such as parallel states, which reduce the total modelling complexity.

Table 1 Comparing implementation effort for the Healing Pet Store

Initial implementation	*Manual*	*MDE/error state/ recovery path*	*Refresh*
Modelling			
Modelled states or features	0	111	33
Modelled connections/transitions	0	104	29
Model error identification	0	0	23
Modelling totals	0	215	85
Implementation			
Implement recovery actions	77	77	0
Implement recovery path chooser	31	0	0
Configuration modifications	96	44	67
Implementation tools	204	121	67

For the MDE implementation effort 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 assumed 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. Our experiments thus only measured the cost of modelling error states and recovery actions and implementing them.

The MDE error state/recovery action approach used the Statecharts presented in Section 3.1. The full Statechart healing specification requires 111 states and 102 transitions between states. As seen in Table 1, 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.

Refresh implementation

Finally, we implemented the swapping capabilities in the Pet Store using Refresh. Refresh's use of Feature models required a total of 33 model elements (features) and 29 connections versus the MDE approach's 111 model elements (states) and 102 connections (transitions). 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 Pet Store 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 55% less implementation effort than the other MDE approach and 60% less modelling effort.

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 151 ms 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 two gigabytes of RAM. The average time required by the constraint solver to derive a new feature selection was 12 ms. These times indicate that Refresh can provide high-performance application healing while reducing modelling and implementation effort.

6 Related work

Microrebooting (Candea et al., 2004) is a technique used to restart only the component, or collection of components in which the failure occurred. Refresh uses microrebooting to eliminate the need to model and implement recovery actions, as described in Section 4. The problem with applying microrebooting alone to service composition healing is that remote services usually cannot be rebooted and thus failures will persist across reboots. Refresh, however, dynamically derives a new service composition and application configuration before rebooting that eliminates the reference to the failed service. Reconfiguration of the service composition allows Refresh

to eliminate references to failed services and prevent an error from persisting across reboots.

Lapouchnian et al. (2005) propose using goal modelling to help develop autonomic applications. Moreover, Lapouchnian's technique uses feature models to help understand the variability in system objectives. Lapouchnian's techniques are focused on developing a design for an autonomic system and also rely on statecharts. Refresh, in contrast, does not require a specific application design – only that the application has different potential services or components that it can be composed of. Furthermore, as Section 3, Lapouchnian's use of statecharts adds a substantial development burden. Refresh does not use error state/recovery action based modelling and implementation and thus avoids this development burden.

Crawford and Dan (2002) developed a framework, known as eModel to assist in monitoring and adapting a system, based on its environment. One of their primary design goals was ease of use for model providers and model users interacting with the framework. This framework requires a model, in the form of an XML file, to specify the states to be identified and the actions to be taken in such situations. The model provider is thus required to identify all potentials states of the system and provide a specific set of actions to take for each state. Section 3 showed the problems associated with specifying error states and recovery actions. Unlike eModel, Refresh does not require explicit specification of recovery actions and avoids these difficulties.

There are a large number of other healing or adaptation approaches (Joshi et al., 2005; Lapouchnian et al., 2005; Bhat et al., 2006; Barbier, 2006; Elkorobarrutia et al., 2006; Calinescu, 2007; Denaro et al., 2007) that rely on identifying error states and then planning and executing some number of recovery actions. As shown in Section 3, modelling and implementing recovery actions is complex and costly. Moreover, as the empirical results from Section 5 showed, by eliminating the need to model and implement recovery actions, Refresh produced a 55% reduction in implementation effort and a 60% reduction in modelling effort compared to techniques that require error state and recovery action modelling.

7 Concluding remarks

Numerous MDE approaches for building self-healing service compositions (Joshi et al., 2005; Lapouchnian et al., 2005; Bhat et al., 2006; Barbier, 2006; Elkorobarrutia et al., 2006; Calinescu, 2007; Denaro et al., 2007) rely on developers modelling each potential error state and the recovery paths from each state. Regardless of the technique used, developers are always responsible for implementing the complex application-specific recovery actions. Moreover, since these approaches use recovery actions to transition an application between two arbitrary states, recovery actions can have unintended side effects on the application, such as producing deadlock or data corruption, that are hard to identify and avoid.

This paper describes how our Refresh technique uses feature modelling to capture the rules for deriving a correct service composition state. Our experience using Refresh showed that leveraging feature models to automatically derive new service compositions when a dependent service fails eliminates the complexity of needing to model each individual error state and recovery action.

Moreover, by using microrebooting to transition the application from its failed service composition to the new service composition, we found that developers need not implement complex recovery actions. Finally, through results obtained from applying Refresh to case studies, we observed that eliminating the modelling and implementation of recovery actions greatly reduced the cost of creating self-healing service compositions.

Refresh is available in open-source form as part of the *GEMS Model Intelligence* project at www.eclipse.org/gmt/gems.

References

Barbier, F. (2006) 'MDE-based design and implementation of autonomic software components', *Cognitive Informatics, 2006. ICCI 2006, 5th IEEE International Conference on 1*.

Barki, H., Rivard, S. and Talbot, J. (1993) 'Toward an assessment of software development risk', *Journal of Management Information Systems*, Vol. 10, No. 2, pp.203–225.

Benavides, D., Segura, S., Trinidad, P. and Ruiz-Cortés, A. (2007) '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*).

Benavides, D., Trinidad, P. and Ruiz-Cortes, A. (2005) 'Automated reasoning on feature models', *17th Conference on Advanced Information Systems Engineering* (*CAiSE05, Proceedings*), LNCS 3520, pp.491–503.

Bhat, V., Parashar, M., Liu, H., Khandekar, M., Kandasamy, N. and Abdelwahed, S. (2006) 'Enabling self-managing applications using model-based online control strategies', *Proceedings of the 3rd IEEE International Conference on Autonomic Computing*, Dublin, Ireland.

Calinescu, R. (2007) 'Model-driven autonomic architecture', *Proceedings of the 4th IEEE International Conference on Autonomic Computing*, Jacksonville, Florida, USA, June.

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

Cohen, J. (1990) *Constraint Logic Programming Languages*, Vol. 33, ACM Press, New York, NY, USA.

Crawford, C. and Dan, A. (2002) 'E-model: addressing the need for a flexible modeling framework in autonomic computing', *Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, *MASCOTS 2002*, *Proceedings on 10th IEEE International Symposium*, pp.203–208.

Denaro, G., Pezze, M. and Tosi, D. (2007) 'Designing self-adaptive service-oriented applications', in *Proceedings of the 4th IEEE International Conference on Autonomic Computing*, Jacksonville, Florida, USA, June.

Elkorobarrutia, X., Izagirre, A. and Sagardui, G. (2006) 'A self-healing mechanism for state machine based components', *Proceedings of the 1st International Conference on Ubiquitous Computing: Applications, Technology and Social Issues*, Alcal de Henares, Madrid, Spain, June.

Esper FAQ, available at http://esper.codehaus.org/tutorials/faqesper/faq.html#performance (n.d.).

Event Stream Intelligence with Esper and NEsper, available at http://esper.codehaus.org (n.d.).

Harel, D., et al. (1987) 'Statecharts: a visual formalism for complex systems', *Science of Computer Programming*, Vol. 8, No. 3, pp.231–274.

Johnson, R. and Hoeller, J. (2004) *Expert one-on-one J2EE development without EJB*, Wrox.

Joshi, K., Sanders, W., Hiltunen, M. and Schlichting, R. (2005) 'Automatic model-driven recovery in distributed systems', at the *24th IEEE Symposium on Reliable Distributed Systems* (*SRDS'05*) pp.25–38.

Kang, K., Kim, S., Lee, J., Kim, K., Shin, E. and Huh, M. (1998) 'FORM: a feature-oriented reuse method with domain-specific reference architectures', *Annals of Software Engineering*, Vol. 5, pp.143–168.

Lapouchnian, A., Liaskos, S., Mylopoulos, J. and Yu, Y. (2005) 'Towards requirements-driven autonomic systems design', *Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software*, pp.1–7.

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

Sun Microsystems (n.d.) *Java Pet Store Sample Application*.

White, J. (2007) 'Healing Pet Store case study implementation. http://www.dre.vanderbilt.edu/jules/petstore-casestudy-code.zip'.

White, J., Czarnecki, K., Schmidt, D. C., Lenz, G., Wienands, C., Wuchner, E. and Fiege, L. (2007a) 'Automated model-based configuration of enterprise Java applications', in *EDOC 2007*.

White, J., Nechypurenko, A., Wuchner, E. and Schmidt, D.C. (2007b) 'Optimizing and automating product-line variant selection for mobile devices, in *11th International Software Product Line Conference*.