Using Filtered Cartesian Flattening and Microrebooting to Build Enterprise Applications with Self-adaptive Healing

J. White[1], B. Dougherty[1], H.D. Strowd[2], and D.C. Schmidt[1]

[1] Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, TN, USA
E-mail: {jules,briand,schmidt}@dre.vanderbilt.edu
[2] Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
hstrowd@andrew.cmu.edu

Abstract. Building enterprise applications that can self-adapt to eliminate component failures is hard. Existing approaches for building adaptive applications exhibit significant limitations, such as requiring developers to manually handle healing side-effects, such as lock release, thread synchronization, and transaction cancellation. Moreover, these techniques require developers to write the complex recovery logic needed to self-adapt without exceeding resource constraints.
This paper provides two contributions to R&D on self-adaptive applications. First, it describes a microrebooting technique called Refresh that uses (1) feature models and a heuristic algorithm to derive a new and correct application configuration that meets resource constraints and (2) an application's component container to shutdown the failed subsystems and reboot the subsystem with the new component configuration. Second, we present results from experiments that evaluate how fast Refresh can adapt an enterprise application to eliminate failed components. These results show that Refresh can reconfigure and reboot failed application subsystems in approximately 150ms. This level of performance enables Refresh to significantly improve enterprise application recovery time compared to standard system or application container rebooting.

1 Introduction

Current trends and challenges. Enterprise applications are large-scale software systems that execute complex business processes, such as order placement and inventory management. Since many enterprise applications receive considerable client traffic, they are often hosted on multiple *application servers* distributed across a local network. Most enterprise applications utilize component middleware, such as Enterprise Java Beans (EJB), to reduce the effort of developing the distributed communication infrastructure by managing the complex distributed interactions between application components and ensuring data integrity through distributed transaction controls.

The failure of an enterprise application can have considerable negative impact (*e.g.*, lost orders, customer irritation, etc.) on an organization. As a consequence, high availability is important for most enterprise applications. Regardless of how much testing and system validation is done, systems can and often do fail [10]. In these situations, speedy recovery of system functionality is critical.

Many organizations use manual processes to recover from failures of enterprise applications [10]. For example, when an EJB application fails, system administrators may restart a group of application servers to attempt to remedy the error. If the error is not fixed by the restart, the administrators may begin collecting logs from the application servers and scanning them for errors. These manual processes are time consuming and error-prone and can leave an application offline for an extended period while the root cause of the failure is identified and remedied.

To address the limitations of human-based recovery of application failure, self-adaptive capabilities are needed that can identify failed components and perform self-adaptive healing to quickly recover. Rather than being off-line for minutes or hours, self-adaptive systems should be able to heal in milliseconds or seconds. Despite the potential payoff associated with self-adaptive healing capabilities, enterprise applications are rarely developed using these techniques since (1) developing the complex logic to determine how to fix a failure cleanly is hard and (2) implementing healing actions requires handling a plethora of challenging side-effects, such as the need to roll-back distributed transactions.

Rather than focusing on fine-grained self-adaptive healing systems, most organizations today leverage clustering and other redundancy mechanisms to ensure availability. Although these macro-level approaches can improve availability, they require additional hardware and complex system administration. Moreover, there are many types of failures that macro-level approaches cannot fix. For example, if a database or remote service that an enterprise application relies on becomes inaccessible due to a network failure, an entire cluster of redundant application instances will be brought down. In this situation, however, if the application could self-heal by loading additional components to communicate with an alternative but not identically accessed database, it could continue to function.

Since software development projects already have low success rates and high costs, building an application capable of healing is hard [3]. 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. In addition, most self-adaptive healing approaches are not suitable for enterprise applications because they do not take into account transaction state, clean release of resources, and other critical actions that must be coordinated with an enterprise application server.

Solution approach → Microrebooting and Feature-based Reconfiguration Our approach to reducing the complexity of developing self-adaptive healing enterprise applications is called *Refresh*. Refresh uses a combination of *feature models* [15] (which describe an application in terms of points of variability and their affect on each other) and *microrebooting* [8] (which is a technique for rebooting a small set of failed components rather than an entire application server) to significantly reduce the complexity of implementing an application with self-adaptive healing capabilities. When an application component fails, Refresh (1) uses the application's feature model to derive a new application configuration, (2) uses the application server's component container to shutdown the failed component, and (3) reboots the component in the newly derived configuration. Refresh relies on the ability to transform a feature model into a constraint satisfaction problem (CSP) and use a constraint solver to autonomously derive a new configuration.

Our previous work [24, 22] showed how Refresh's CSP-based healing could be used to reduce the complexity of implementing self-adaptive healing applications. When the self-adaptive healing mechanism needs to respect resource constraints, such as bandwidth or memory limits, a CSP-based approach for deriving application configurations from feature models becomes too slow for enterprise applications. Selecting a feature configuration that adheres to resource constraints is an NP-Hard problem that is time-consuming to solve with a CSP-solver.

This paper extends our previous work by showing how *Filtered Cartesian Flattening* and multidimensional multiple-choice knapsack heuristic algorithms can be used as the feature selection mechanism to drastically reduce feature selection and consequently, self-adaptive healing time. We show how these algorithms can be combined with microrebooting, component middleware container hotswap capabilities, and feature models to create self-adaptive enterprise applications. We also present empirical results that show the increase in scalability and speed provided by Filtered Cartesian Flattening versus a CSP-based reconfiguration approach.

Paper organization. The remainder of this paper is organized as follows: Section 2 presents the e-commerce application we use as a case study throughout the paper; Section 3 enumerates current challenges in applying existing MDE techniques for building self-adaptive healing applications that must adhere to resource constraints; Section 4 describes Refresh's approach to using feature models, microrebooting, and Filtered Cartesian Flattening to reduce the complexity of modeling and implementing an application that can heal; Section 5 analyzes empirical results obtained from applying Refresh to our case study; Section 6 compares Refresh with related work; and Section 7 presents concluding remarks.

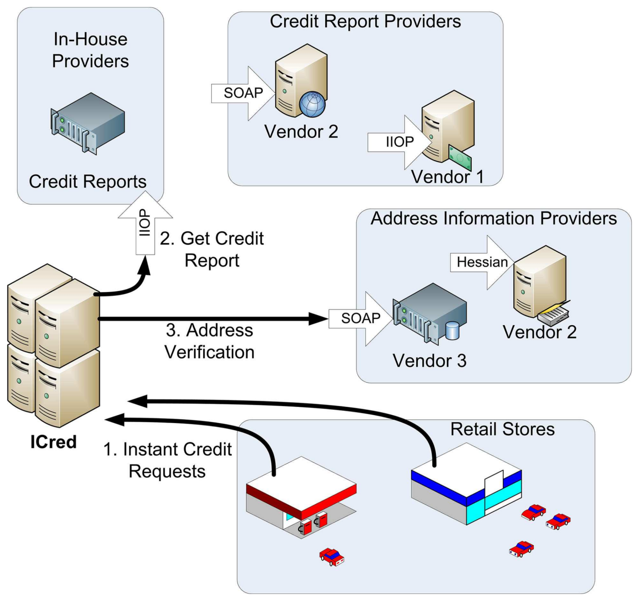

Fig. 1: The ICred Instant Credit Enterprise Application

2 Case Study: ICred

Enterprise applications have a number of complex considerations that make it hard to build an application capable of self-adaptive healing. To showcase these challenging aspects of enterprise applications, we present a case study based on an enterprise application that provides instant credit decisions for in-store purchases. Throughout the paper, we refer to our case study application as *ICred*. The high-level architecture of ICred is shown in Figure 1.

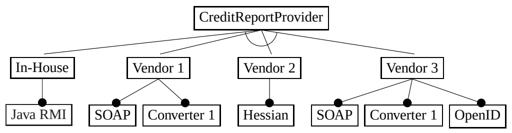

Fig. 2: Feature Model of the Available Credit Report Providers

When a customer in a retail store wishes to purchase an expensive item, such as a computer projector, the store clerk can offer the customer an instant line of credit to make the purchase and pay later. If the customer is interested in obtaining the line of credit, the store clerk keys in the customer's information and a request for credit is sent to the remote ICred server for approval. ICred must pull the customer's credit report and other needed information to make the credit decision.

ICred is used for a number of different retailers and each retailer has a specific set of requirements for validating a credit application and issuing an approval. Stores that sell less expensive and less durable items, such as computer equipment, may require a simple validation of the customer's residence information and bank accounts. Vendors of more expensive items, such as car dealerships, require more extensive sets of information, such as a full credit report and verification of a previous address. Each customer is supported by a custom configuration of ICred that is not shared.

Instances of ICred are run and managed by an information supplier on behalf of retail chains. Each piece of information needed for the credit decision can either be obtained in-house or from another information supplier. Whenever ICred requests a piece of information on a customer from another supplier, a small fee is paid to the information vendor that services the request. Information can be purchased from multiple vendors at varying prices based on volume.

An ICred configuration receives instant credit requests from thousands of retail locations and must be continuously available. A failure to make a credit decision could result in a customer not making a large purchase. When one of ICred's information suppliers becomes unavailable, ICred can fail over to another supplier. For example, Figure 3, shows the different sources of information that can be used to obtain credit reports.

Figure 3 shows a feature model for an e-commerce application called `CreditReport-Provider` that represents a service for obtaining credit reports. The `CreditReportProvider` feature has different sub-features, such as different potential vendors that can serve as the credit report provider service. If the `Vendor 1` feature is chosen, it excludes the other potential providers' services from being used (it constrains the other features). If

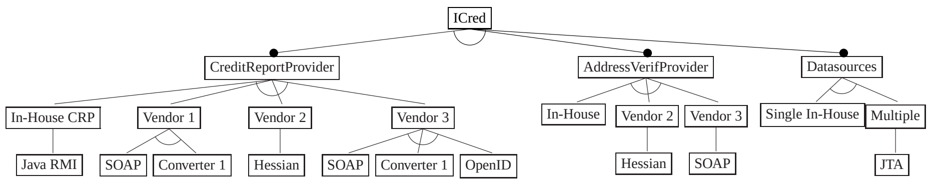

Fig. 3: Feature Model of the Available Credit Report Providers

`Vendor 1` service fails, a new feature selection can be derived that does not include the failed service's feature. When a component failure occurs, Refresh uses an application's feature model and a constraint solver to derive an alternate but legal configuration of the application's component that eliminates the failed component implementation.

Failing over to another supplier involves a number of complex activities. Information vendors represent the same information using slightly different formats and leverage different request protocols. Depending on the vendor chosen, it may be necessary to load various special converter and protocol handlers into the application. Moreover, since ICred receives a high request volume, it must try to ensure that the combination of protocols used by its current configuration of information vendors will not saturate the network. Finally, since per request prices vary across information vendors, ICred must also try to minimize the cost incurred by the configuration of external information vendors.

To showcase the complexity of performing self-adaptive healing in an enterprise application, we explore the difficulty of failing over between local and external information services in ICred. Section 3 presents the complexities of developing healing logic and adaptation actions. Section 4 shows how Filtered Cartesian Flattening can be used to derive a new application configuration to eliminate a failure and boot the configuration using the application's component container.

3 Self-adaptive Healing Challenges for Enterprise Applications

This section describes the challenges associated with implementing a self-adaptive healing enterprise application. First, we show that the need to adhere to resource constraints, such as total available network bandwidth, makes finding a way of healing an enterprise application an NP-Hard problem. Second, we discuss how even if a way of healing the application can be found, numerous accidental complexities, such as the need to properly handle in-process transactions, make it hard to implement healing actions.

3.1 Challenge 1: Resource constraints make adaptation actions extremely complex

When an application component fails and requires healing, adaptation actions must be run to reach a new and valid state. We term the sequence of adaptation actions that are run to fix a failed application subsystem as a *recovery path*. A chief complexity of implementing an application capable of self-adaptive healing is building the logic to select a recovery path for a given application failure.

Recovery actions are used to perform two key types of activities: (1) performing resource cleanup and release from failed application components and (2) determining what new application components can be loaded to heal a failure. The difficulty in building recovery logic is that the second critical activity, selecting the new components to load, requires finding a series of application components that fit into the resource limits of the application. Selecting a series of components that adheres to a resource limit is an instance of the NP-Hard knapsack problem.

For example, consider the failure of the `In-House CRP`. ICred's `In-House CRP` can be swapped out to one of three remote services. When the local `In-House CRP` fails, the recovery logic must determine the optimal subset of these remote services to fail-over to in order to fix the error. Furthermore, the recovery logic must attempt to minimize the cost of the information provider services that are used in the new configuration.

Network bandwidth consumption must be accounted for in the healing process. Each remote service uses a different protocol for communication and consumes varying amounts of network bandwidth. The Java RMI service uses the efficient binary IIOP protocol. The SOAP service, however, sends comparatively large XML messages over HTTP and consumes significantly more bandwidth. Depending on what combination of services are currently being used by the application, the network may or may not have sufficient bandwidth to fail over to the SOAP-based service. Even if the `Vendor 1` SOAP-based service is the cheapest to fail-over to, it may not be possible due to network bandwidth limitations.

If the SOAP-based service is the only of the three alternate remote services that is reachable after the failure, the healing logic may need to shutdown and swap other parts of the application (*e*.g., AddressVerifProvider, etc.) to less bandwidth consumptive remote services so that the SOAP service can be used. For example, if the `CreditReport-Provider` is using a SOAP-based remote service, it may need to be swapped to `Vendor 2`'s Hessian-based service to allow the SOAP-based product service to be used. Finding the right set of services to swap in and out of the application is NP-Hard and difficult to do quickly at runtime. Performing simultaneous cost optimization is even harder.

Designing this type of complex adaptive logic to choose a recovery path is hard. For most enterprise application development projects, this type of complex adaptation logic is not feasible to develop from scratch. Moreover, with nearly 53% of software development projects being completed over-budget and 18% of projects canceled [25, 17] adding this type of complex adaptive logic adds significant risk to a project. In Section 4.2, we show how we use feature models and the Filtered Cartesian Flattening algorithm to eliminate the need to write complex recovery path selection logic.

3.2 Challenge 2: Accidental complexity makes adaptation actions hard to develop

Enterprise applications are typically built on top of component middleware, such as Enterprise Java Beans. Component middleware provides an *application container*, which manages the intricate details of thread synchronization, distributed/local transaction control, and object pooling. One key challenge of developing self-adaptive healing mechanisms for enterprise applications is properly and cleanly handling the nuanced considerations related to these aspects of the application. For example, if a credit report provider fails, the application must ensure that any distributed transactions associated with the provider are rolled back and cleanly terminated before a new provider is swapped in. Figuring out the right way to terminate transactions, release locks, terminate network connections, and release other resources when healing occurs is hard.

When healing takes place, a further challenge of properly handling transactions and other container managed services is that the application does not have direct control over them. For example, EJBs are not allowed to perform thread synchronization or manually obtain locks. If a failure occurs in a multi-threaded application, therefore, it is hard for an EJB to ensure that data corruption does not occur if it reconfigures the application's internal structure.

An issue further complicating the healing process is that healing may require changing the policies the container uses to manage these services. In ICred, for example, if ICred is using all local data sources, it can use standard local transaction management through the container. If ICred fails over to a remote datasource, however, it must also force the container to reconfigure itself to use the Java Transaction API (JTA) to manage distributed transactions across both the local and remote datasources. It is hard to perform these numerous complex reconfiguration processes manually. Section 4 describes how we use the application component container's standard lifecycle mechanisms to perform healing and eliminate the need to write custom recovery actions.

4 Solution Approach→Combining Refresh and Filtered Cartesian Flattening

The challenges in Sections 3.1-3.2 stem from two primary causes: (1) the need for developers to implement complex recovery path selection logic that accounts for resource constraints and (2) the need for developers to implement complex recovery actions that correctly coordinate and handle the side-effects of healing, such as graceful transaction failure. This section presents an overview of *Refresh* [24] and shows how we extend it with the Filtered Cartesian Flattening algorithm to address these challenges.

4.1 Overview of Refresh

Refresh uses feature models to capture the rules for what is a correct system state, which when combined with the Filtered Cartesian Flattening feature selection algorithm, can be used to automate the selection of a new configuration to reboot into. After a new and valid configuration is found, Refresh uses the application's container to swap out the failed components and boot the new alternate configuration. Automating the reconfiguration process eliminates the need for developers to design and implement the recovery path selection logic, which addresses Challenge 2 from Section 3.1.

Using the container's normal lifecycle facilities to perform healing (*e.g.*, rebooting and hotswapping), eliminates the need for developers to manage the side-effects of healing since they are automatically managed by the container when lifecycle management activities are performed. As shown in Section 5, using Filtered Cartesian Flattening and container rebooting to perform resource constrained healing provides fast recovery at a significantly reduced development cost compared to recovery action oriented techniques.

Refresh is based on the concept of microrebooting [8]. 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 from the application's feature models that does not contain the failed features (*e.g.*, remote services).

The application configuration 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 Refresh healing process is shown in Figure 4.

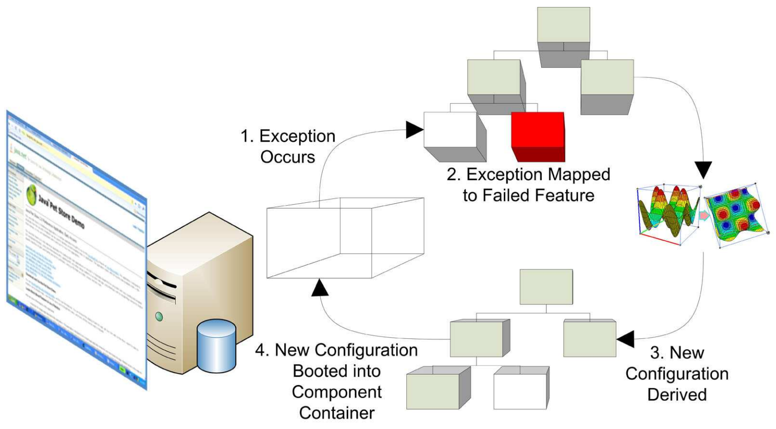
1. Exception Occurs
2. Exception Mapped to Failed Feature
3. New Configuration Derived
4. New Configuration Booted into Component Container

Fig. 4: Refresh Healing Process

Throughout the healing process, Refresh does not use any custom recovery actions. All error states are transitioned out of through a single recovery path, shutting down the application components via the container, automatically deriving a new and valid configuration/service composition, and restarting the application components. No application-specific recovery action modeling or recovery application implementations are required.

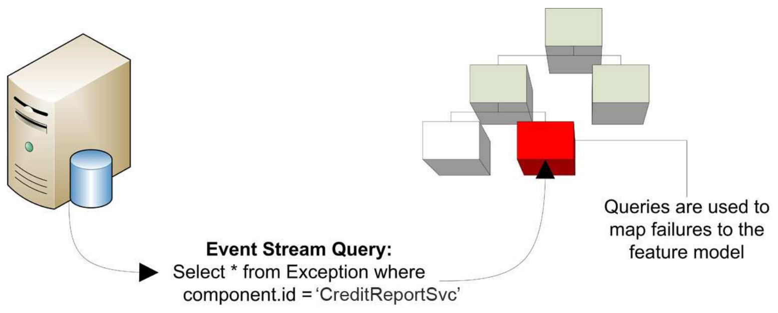
Event Stream Query:
Select * from Exception where component.id = 'CreditReportSvc'
Queries are used to map failures to the feature model

Fig. 5: Mapping Failures to Features

Refresh interacts directly with the application container, as shown in Figure 4. 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. An example event stream query and mapping to the feature model is shown in Figure 5. Whenever an application's configuration requires healing, *environment probes* are used to determine available remote services and global application constraints, such as whether or not JTA is present.

4.2 Feature Model Configuration Healing

At the core of the Refresh approach is its ability to derive a new configuration for the application that both eliminates any failed components and adheres to resource limitations. Refresh uses a feature model of the application to capture the rules for reconfiguration. When a failure occurs, the configuration space defined by the feature model is searched for a new and valid configuration.

A feature model is used to define the configuration space of an enterprise application by defining configuration rules, such as:

- What alternate implementations of components are available
- What dependencies (such as libraries, configuration files, etc.) must be used with each component
- What combinations of components form a valid and complete application composition
- Annotations describing how much RAM, Bandwidth, etc. is consumed by each feature

Searching a feature model's solution space for a valid configuration is an instance of the NP-complete circuit satisfiability problem. The feature model can define an arbitrary boolean formula. Each boolean term represents the presence of a specific feature. The constraints in the feature model are the AND, OR, and NOT constraints used to form the circuit satisfiability clauses. Numerous research approaches have applied techniques such as SAT solvers [4, 18], Binary Decision Diagrams (BDDs) [9], and Constrant Satisfaction Problem (CSP) solvers [20, 5], to find valid feature model configurations.

Our initial implementation of Refresh used the CSP-based approach proposed by Benavides [5] and extended by us to include resource constraints [23, 21]. CSP-based feature selection techniques work well when resource constraints are not included. Through experiments that we performed [22], however, we observed significant scalability problems for CSP-based feature derivation with resource constraints, as shown in the results in Section 5.3. Other exponential exact derivation techniques, such as SAT solvers and BDDs, suffer from these same scalability problems [22].

A number of heuristic techniques can be applied to improve the performance of these exact solving techniques. For example, by choosing the correct variable ordering, many BDD-based problems can be simplified significantly. Choosing the best variable ordering, however, is an NP-Hard problem and must be performed on a per-problem basis. Similar techniques can be applied to CSP-based configuration derivation, but must also be performed on a per problem basis.

Since the goal of Refresh is to simplify the implementation process of applications capable of self-adaptive healing, it would not be reasonable to expect these heuristic techniques to be learned and applied by normal developers. Moreover, the application of these techniques requires significant skill. Just as good application design is an art form, knowing which of these heuristics to apply and how to apply them is also an art. We do not think is reasonable to expect developers are willing and/or able to become experts in these techniques. We have therefore not considered these techniques for Refresh.

4.3 Filtered Cartesian Flattening

To overcome the scalability issues associated with finding a new and valid feature configuration, we incorporated the Filtered Cartesian Flattening feature selection algorithm into Refresh. Filtered Cartesian Flattening is an polynomial-time algorithmic technique that approximates a feature configuration problem with resource constraints as a multidimensional multiple-choice knapsack problem (MMKP) [22]. A standard knapsack problem attempts to find a subset of a series of items that fits into a knapsack of limited size and maximizes the value of the items inside the knapsack. An MMKP problem is a variant of a knapsack problem where the items are subdivided into disjoint sets and exactly one item must be chosen from each set to put into the knapsack. Both variants of the problem are NP-Hard [19].

The reason that Filtered Cartesian Flattening approximates the feature configuration problem as a MMKP problem is that there are a number of excellent polynomial-time heuristic algorithms that have been developed for MMKPs. For example, the M-HEU and C-HEU heuristic MMKP algorithms can solve large MMKPs in milliseconds with an average of over 95% optimality [19]. Once a feature configuration problem is represented as a MMKP, these heuristic algorithms can be used to derive a feature selection. When a failure occurs, the speed of Filtered Cartesian Flattening, which uses MMKP heuristic algorithms, is far more important than its minor tradeoff in healing solution optimality.

Filtered Cartesian Flattening approximates a feature model as an MMKP problem by finding a series of independent subtrees in the feature model that can be configured independently. Each of these subtrees is represented as an MMKP set. The items within the MMKP sets represent the valid configurations of their respective subtrees. Because each MMKP set represents a subtree of the feature model, by choosing a configuration from each MMKP set and composing them, a complete feature model configuration will always be reached.

Since there may be an exponential number of possible configurations of each subtree, Filtered Cartesian Flattening employs an approximation technique. As Filtered Cartesian Flattening enumerates the possible configurations of each feature model subtree, it bounds the MMKP set size and selectively filters which configurations are propagated into the sets. Typically, a heuristic that selects configurations with the best ratio of value/resource consumption is used as the selection criteria.

To derive a configuration that omits the failed feature while still adhering to resource constraints, refresh utilizes Filtered Cartesian Flattening. During the enumeration process, Filtered Cartesian Flattening disallows the inclusion of the failed feature to any of the MMKP sets. Due to this exclusion, the feature can not belong to any configuration that can be derived from the resulting MMKP problem, thus disallowing the failed feature to be present in the new feature set. After deriving the new feature configuration, the application container is used to shutdown the old configuration and boot the new configuration.

5 Refresh and Filtered Cartesian Flattening Performance

This section presents results from experiments we performed to empirically evaluate the performance of Refresh's feature reconfiguration and container-based healing. We used a reference implementation of an enterprise request processing application, implemented on top of the Java Spring Framework [13], that could fail over between a number of different remote and local data sources. The implementation was comprised of roughly 15,000 lines of code using a combination of Java, Java Server Pages, XML, and SQL.

Our prior work [24] conducted experiments to measure the reduction in implementation complexity provided by Refresh. This paper extends our prior work by evaluating the performance of feature model and container-based healing. Moreover, we analyze how automated feature selection techniques can be made more scalable to handle resource constraints and optimization goals.

5.1 Hardware and Software Testbed Configuration

The experiments with the application were performed on a Pentium Core DUO 2.4ghz processor, with 3 gigabytes of RAM, running Windows XP. A Java Virtual Machine, version 1.6, was run in client mode for all tests. We used Apache Tomcat 6 as the web container for the application.

To test the performance of Refresh, we implemented a self-adaptive healing version of the application and compared its performance to the conventional (non-adaptive) implementation. The first set of experiments compared the performance of the Refresh-based application to the conventional unmodified application to measure the overhead of using a container-based healing approach. The second set of experiments extended the Refresh application to adhere to a bandwidth constraint. We measured the configuration derivation times of both the Filtered Cartesian Flattening configuration derivation technique and the CSP-based technique to compare scalability.

5.2 Refresh Performance

To create an initial performance baseline to compare against, we used Apache JMeter to simulate the concurrent access of 30 different customers to the application and the time required to complete 1,000 requests. Figure 6 shows the average time required to complete various parts of the request process throughout the experiment. We also used Apache JMeter to simulate the concurrent access of 30 different customers to the Refresh-enabled application and the time required to complete 1,000 requests. To measure Refresh's worst case performance overhead, we used the CSP-based configuration derivation technique for this experiment since it was slower than the Filtered Cartesian Flattening technique. The performance results were identical to the conventional application implementation. This result was expected since the time-consuming healing process is only invoked during component failures. Moreover, our Refresh application implementation used very lightweight Spring interceptors to monitor components for

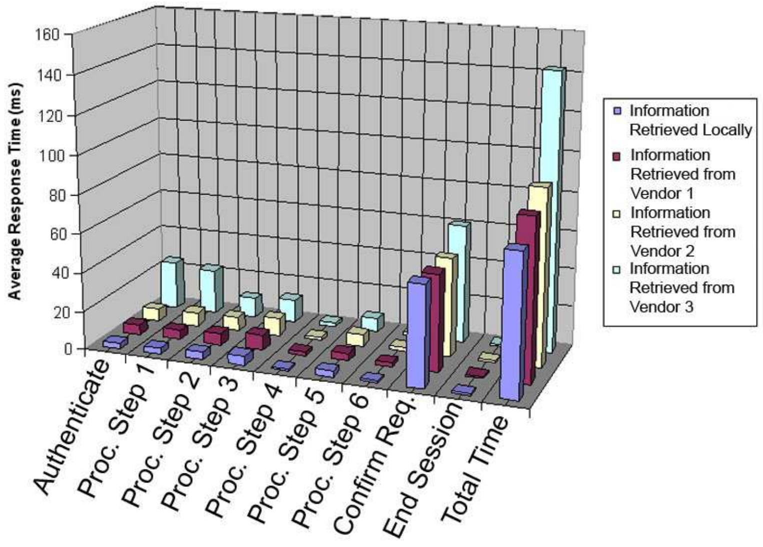

Fig. 6: Average Response Time for the Application

exceptions. We saw no measurable performance penalty for the use of these interceptors.

To determine how quickly the Refresh application could self-heal, we ran a further trial of Apache JMeter tests to simulate an additional 1,000 requests. During the experiment, we used fault injection to randomly simulate the failure of different services. The faults were injected by adding code to the local services to throw Java runtime exceptions that would force Refresh to heal the application by swapping remote services for the failed local services. After the local services were swapped to remote services, we randomly shutdown the remote services used by the application to force the failover to alternate remote services or back to a local service that had become available.

Over the tests, shutting down a failed subsystem and rebooting the container into a new configuration averaged roughly 140ms. The CSP technique required an average of an additional 10ms to find the new configuration to reboot into. When this result is compared to Figures 6, it can be seen that the healing time is slightly more than the average time to complete an order.

Figure 7 overlays the application's worst case response time using a local information provider, a remote information provider, and a remote information provider that is swapped back to a local provider because of a failure. The failure of the remote service is easily discernible on request 7. Before the failure occurs, the application has the same average performance as the conventional application using a remote service. Once the failed service is healed, the application again has the same average performance as the conventional application with the local service. This result indicates that container-based healing incurs little or no pre- or post- healing performance penalties.

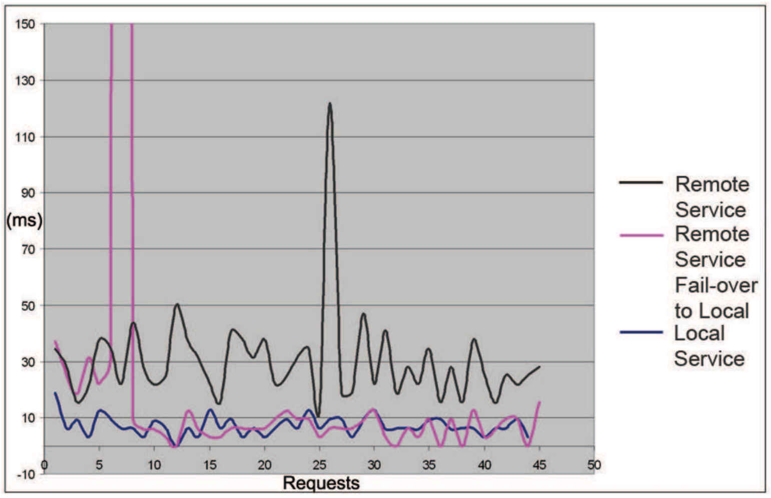

Fig. 7: Application Performance Before and After Healing

5.3 Filtered Cartesian Flattening vs. CSP-based Configuration Derivation

The next set of experiments compared the scalability and speed of Filtered Cartesian Flattening versus a CSP-based configuration derivation technique. We extended the Refresh application's healing configuration to attempt to respect a bandwidth constraint while healing. Moreover, we directed the healing mechanism to also attempt to minimize the total cost consumed by the new configuration's services. Our CSP-based configuration solver was based on the Java Choco open source constraint solver [1].

First, we compared the time for Filtered Cartesian Flattening and the CSP-based techniques to derive a new configuration for the standard points of variability in the application. We then iteratively added 32 additional information providers to consider in the configuration derivation process. Both techniques found solutions for each size configuration problem. The results from this experiment are shown in Figure 8.

Initially, the CSP technique requires 234ms to configure the conventional application implementation with the additional resource constraints and bandwidth minimization goal. In the experiments presented in Section 5.2, the CSP-based technique required an average of 10ms to find a new and valid configuration without resources constraints. The new constraints and optimization goal cause a significant increase in the solving time to 234ms. Furthermore, by the time the 32 additional information providers were added into the configuration, the CSP-based technique required over 30 minutes (1,835,406ms) to derive a new configuration.

The time for Filtered Cartesian Flattening to derive a new configuration across the different configuration sizes is shown by the red line in the lower part of Figure 8. Initially, Filtered Cartesian Flattening requires 15ms to derive a configuration, which is substantially less than the CSP-based technique's 234ms. Moreover, when the 32 additional providers are added, Filtered Cartesian Flattening is able to derive a configuration in 31ms. Filtered Cartesian Flattening's 31ms is many orders of magnitude less than the $\sim$30mins for the CSP-based technique. This result shows that Filtered Cartesian Flat-

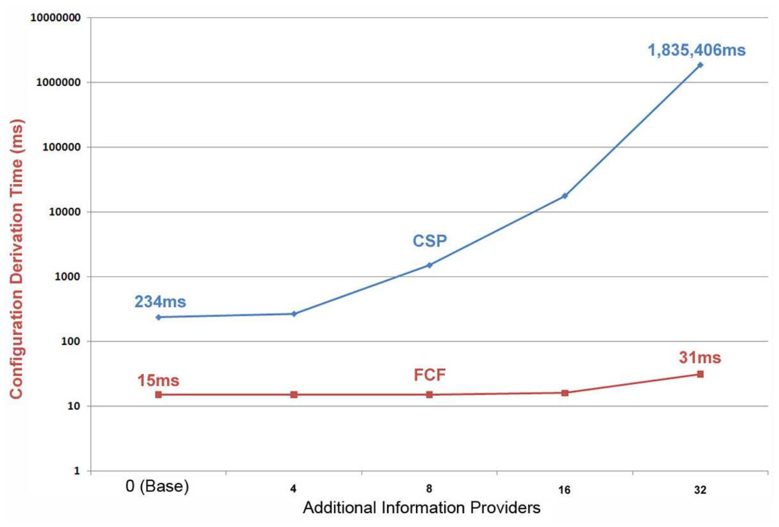

Fig. 8: Filtered Cartesian Flattening vs. CSP-based Configuration Derivation Time for the Application

tening is significantly more scalable than the CSP-based technique for our case study application.

6 Related Work

This section compares our work on Refresh and Filtered Cartesian Flattening with other research work. First, we compare Refresh to the original technique of microrebooting. Next, we compare and contrast Refresh with other feature-based self-adaptive healing techniques. Finally, we compare Refresh to other non-feature based self-adaptive healing techniques.

Microrebooting related work. Refresh is based on the idea of Microrebooting [8]. Microrebooting restarts individual components or collections of components to fix an error. The number of components that are rebooted continues to grow if successive reboots do not eliminate the error. Microrebooting can help to eliminate some types of problems but may not fix all issues.

For example, if one of the information provider services fails, restarting the application will not fix the error since it is in a remote component. Instead, the local application must be rebooted in an alternate configuration to eliminate the error. As we showed in Section 3.1 determining how to eliminate failed components is a challenging problem. Refresh uses the Filtered Cartesian Flattening algorithm to eliminate this problem by dynamically deriving a new application configuration to reboot the failed subsystem into. This type of reconfiguration and rebooting can eliminate both local errors and references to failed remote services, which microrebooting alone cannot fix.

Feature-based healing related work. Other approaches to application healing have been developed that leverage a combination of goal modeling and feature models [16]. In the approach by Lapouchnian et al. feature models are used to find points of variation in the application. The application adaptation is driven by Statecharts. As we showed in Section 3.1, specifying the logic to solve the NP-Hard problem of reconfiguring the application subject to resource constraints is hard to implement in either Java, C++, or Statecharts.

The approach of using Statecharts to drive the adaptive healing of the application burdens enterprise application developers with a extremely complex problem. Moreover, there can be an exponential number of states that may need to be modeled to properly adapt in all resource availability scenarios. In contrast, Refresh does not require an explicit adaptation plan but instead a model of how the application can be reconfigured. Refresh then automates the complex problem of deriving a new application configuration that fits the current available resources.

Self-adaptive healing related work. Other approaches also use the idea of identifying error conditions and then planning adaptation actions that should be triggered [7, 14, 11, 6, 2, 16, 12]. These approaches also require developers to handle the complex problem of determining how to best adapt the application's configuration while adhering to a resource constraint. Determining how to reconfigure in the face of a resource constraint is an NP-Hard problem. In contrast, Refresh automates this recovery logic by using the Filtered Cartesian Flattening approximation algorithm to derive a new application feature set that can be used to continue functioning.

7 Concluding Remarks

A common approach to simplifying the development of self-adaptive healing applications is to use a model of an application's adaptation logic to generate self-adaptive healing code or guide self-adaptive healing at runtime [7, 14, 11, 6, 2, 16, 12]. This approach to simplifying the development of self-adaptive healing applications does not, however, eliminate the key complexity, which is the logic needed to deduce how to heal the application. Moreover, when resource constraints must be considered in the adaptation process, determining how to adapt the application without exceeding the resource limitations is an NP-Hard problem.

This paper showed how our Refresh technique—based on a combination of microrebooting and dynamic reconfiguration using feature models—can simplify the development of self-adaptive healing applications. Rather than simply rebooting in the same configuration (which could cause errors involving remote services to persist), Refresh dynamically derives a new application configuration to reboot into using the application's feature model. Moreover, we showed that by using the FCF algorithm to perform the derivation of the new feature selection, Refresh could respect resource constraints and still find alternate feature configurations fast.

The following list presents the lessons we have learned from our experiences building self-adaptive healing enterprise applications using Refresh:

– **CSP-based reconfiguration techniques are sufficient if no resource constraints are present.** If resource constraints are not considered in the reconfiguration process, CSP and other exact techniques, such as SAT solvers, provide sufficient performance to derive new configurations. Only when resource constraints are added is FCF needed.

– **Container lifefcycle methods can managing accidental healing complexities.** Containers must be able to release resources, roll back transactions, and perform other cleanup whenever an application container is shutdown. By reusing this lifecycle mechanism to perform healing, significant accidental complexity is managed by the container on the developer's behalf.
– **Optimization goals may not be easy to formalize.** In many domains, resource constraints and optimization goals can be hard to formalize since it is not clear how choosing one service over another affects cost and resource consumption. Interactions between organizations, however, often do have a known resource consumption and cost associated with them.
– **Service-oriented architectures fit well into the Refresh healing model.** Many standard enterprise applications that do not use remote services do not have variability built into the components that can be used to process requests. Enterprise applications that use service-oriented architectures typically do have the potential to be swapped to fail over to alternate services at runtime.

Refresh is available in open-source form as part of the *GEMS Model Intelligence* project at `www.sf.net/projects/gems`.

References

1. Choco constraint programming system. http://choco.sourceforge.net/.
2. F. Barbier. MDE-based Design and Implementation of Autonomic Software Components. *Cognitive Informatics, 2006. ICCI 2006. 5th IEEE International Conference on*, 1, 2006.
3. H. Barki, S. Rivard, and J. Talbot. Toward an assessment of software development risk. *Journal of Management Information Systems*, 10(2):203–225, 1993.
4. D. Batory. Feature Models, Grammars, and Prepositional Formulas. *Software Product Lines: 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005: Proceedings*, 2005.
5. 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.
6. V. Bhat, M. Parashar, H. Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed. Enabling Self-Managing Applications using Model-based Online Control Strategies. *Proceedings of the 3rd IEEE International Conference on Autonomic Computing, Dublin, Ireland*, June 2006.
7. R. Calinescu. Model-Driven Autonomic Architecture. *Proceedings of the 4th IEEE International Conference on Autonomic Computing, Jacksonville, Florida, USA, June*, 2007.
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. K. Czarnecki, M. Antkiewicz, C. Kim, S. Lau, and K. Pietroszek. FMP and FMP2RSM: Eclipse Plug-ins for Modeling Features Using Model Templates. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 200–201, October 2005.
10. D. P. D. Oppenheimer, A. Ganapathi. Why do Internet Services Fail, and What can be Done about It? *Proceedings of the USENIX Symposium on Internet Techňnologies and Systems*, March 2003.
11. Denaro, Giovanni and Pezze, Mauro and Tosi, Davide. Designing Self-Adaptive Service-Oriented Applications. 2007.
12. X. Elkorobarrutia, A. Izagirre, and G. Sagardui. A Self-Healing Mechanism for State Machine Based Components. *Proceedings of the 1st International Conference on Ubiquitous Computing: Applications, Technology and Social Issues, Alcalá de Henares, Madrid, Spain, June*, 2006.
13. R. Johnson and J. Hoeller. *Expert one-on-one J2EE development without EJB*. Wrox, 2004.

14. K. Joshi, W. Sanders, M. Hiltunen, and R. Schlichting. Automatic Model-Driven Recovery in Distributed Systems. *At the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 25–38, 2005.
15. K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-; oriented reuse method with domain-; specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.
16. A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu. Towards Requirements-driven Autonomic Systems Design. *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, 2005.
17. K. Linberg. Software developer perceptions about software project failure: a case study. *The Journal of Systems & Software*, 49(2-3):177–192, 1999.
18. M. Mannion. Using First-order Logic for Product Line Model Validation. *Proceedings of the Second International Conference on Software Product Lines*, 2379:176–187, 2002.
19. M. Mostofa Akbar, M. Sohel Rahman, M. Kaykobad, E. Manning, and G. Shoja. Solving the Multidimensional Multiple-choice Knapsack Problem by constructing convex hulls. *Computers and Operations Research*, 33(5):1259–1273, 2006.
20. P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software, in press*, 2007.
21. 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.
22. J. White, B. Dougherty, and D. Schmidt. Filtered Cartesian Flattening. *Workshop on Analysis of Software Product-Lines at the International Conference on Software Product-lines*, October 2008.
23. 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.
24. J. White, H. Strowd, and D. C. Schmidt. Creating Self-healing Service Compositions with Feature Modeling and Microrebooting. *The International Journal of Business Process Integration and Management (IJBPIM), Special issue on Model-Driven Service-Oriented Architectures*, 2008.
25. B. Whittaker. What went wrong? Unsuccessful information technology projects. *INFORMATION MANAGEMENT AND COMPUTER SECURITY*, 7:23–29, 1999.