Automated Model-based Configuration of Enterprise Java Applications

Jules White and Doulas C. Schmidt
Vanderbilt University,
{jules, schmidt}@dre.vanderbilt.edu

Krzysztof Czarnecki
University of Waterloo,
kczarnec@swen.uwaterloo.ca

Christoph Wienands, Gunther Lenz,
Egon Wuchner, and Ludger Fiege
Siemens AG,
{christoph.wienands, lenz.gunther,
egon.wuchner, ludger.fiege}@siemens.com

***Abstract*—The decentralized process of configuring enterprise applications is complex and error-prone, involving multiple participants/roles and numerous configuration changes across multiple files, application server settings, and database decisions. This paper describes an approach to automated enterprise application configuration that uses a feature model, executes a series of probes to verify configuration properties, formalizes feature selection as a constraint satisfaction problem, and applies constraint logic programming techniques to derive a correct application configuration. To validate the approach, we developed a configuration engine, called Fresh, for enterprise Java applications and conducted experiments to measure how effectively Fresh can configure the canonical Java Pet Store application. Our results show that Fresh reduces the number of lines of hand written XML code by up to 92% and the total number of configuration steps by up to 72%.**

I. Introduction

Enterprise applications are large-scale software programs, typically hosted on multiple application servers, that perform complex business processes. Enterprise applications commonly support thousands or more simultaneous users and are often written using component middleware, such as Enterprise Java Beans. Due to their large number of components, complicated XML-based configuration files, and complex interdependencies between components, enterprise applications are often hard to configure.

Enterprise application configuration is typically a decentralized process. Multiple development roles edit configuration files, install applications, and perform other configuration steps to deploy an enterprise application. Each role usually operates semi-independently from other roles and focuses on aspects of application configuration pertinent to requirements the role is responsible for. For example, database developers identify the best database vendor, database schema, and database configuration parameters to use; component developers determine what software components are needed to meet the functional requirements for the application; and IT administrators install and configure application servers on the appropriate nodes in data centers.

The diverse configuration decisions made by each role outlined above constrain the possible configuration decisions of other roles. For example, when database developers choose a database, component developers must use the appropriate database driver for that database. These configuration decisions are distributed across roles and configuration files and must ultimately be integrated to create a complete and valid configuration. When integration takes place, each role often performs other configuration steps (such as installing the correct database driver) necessitated by decisions made by other roles. This integration process may require adding new components to adapt the application to its target environment, loading extra libraries into the application server, or other types of configuration steps.

It is hard to keep track of and analyze an enterprise application's configuration decisions (configuration state) since these decisions are enacted by multiple roles, involve hundreds or more components, and are spread throughout numerous configuration files. Even after the configuration state is collected, the complex interdependencies and implications of the configuration decisions must be understood to check the validity of the configuration state and derive further configuration steps to perform. Finally, after a complete configuration for the application is derived, the configuration must be enacted by the multiple roles in numerous configuration files.

Configuration errors related to functional requirements have been shown to be a major contributor to enterprise application downtime and cost. In some studies, for example, misconfiguration from manual processes has been shown to cause over 50% of all application failures [10]. One approach to alleviating the complexity of configuring enterprise applications is to use model-driven development [25]. With a model-based approach, a model of the application's configuration rules and configuration state is first built. Configuration artifacts, such as XML configuration files, are generated from the model. By creating a model of application components and configuration requirements, algorithmic techniques (such as constraint solvers) can be used to check configuration correctness and derive valid configurations.

Feature modeling [17], [9] is a promising modeling technique for representing the configuration state of enterprise applications. This technique can capture the configuration dependencies between roles and non-functional requirements for enterprise applications. Feature modeling provides a set of modeling formalisms that decompose an application based on functional and non-functional variations and formalize the rules by which these variabilities may be composed into an application variant. In the context of enterprise applications, feature modeling can be used to capture (1) what configuration decisions must be made to install an enterprise application, (2)

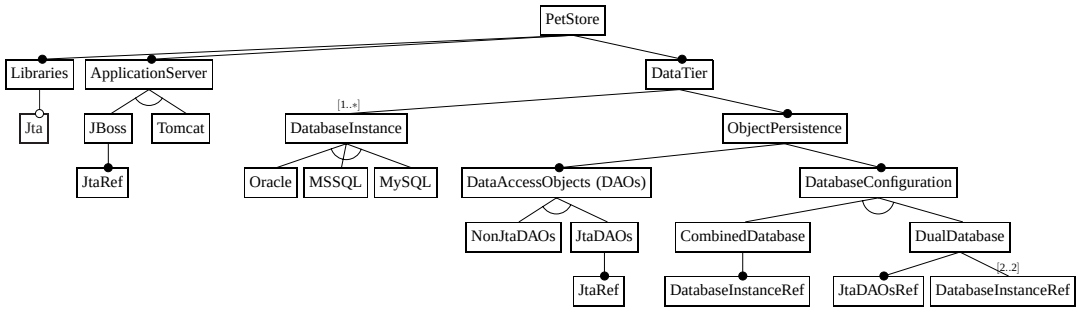

Fig. 1: Feature Model of the Features Related to the J2EE Pet Store's Data Tier

what roles are responsible for what configuration steps (by having a separate feature model per role), (3) how each role's configuration steps affect other roles, and (4) how the target infrastructure and requirements limit the valid configuration possibilities.

To configure an application with a feature model, development team members (such as component developers, database developers, etc.) first identify a *feature selection*, which is a group of desired functional capabilities that constitute a complete configuration of an application and adhere to the constraints specified in the feature model. These participants must then determine what configuration actions, such as adding component IDs to application XML descriptors or installing a specific database, are required to enable and/or implement the functionality specified in the feature set. What we term *feature selection* is also often called *product configuration* [19]. To avoid confusion, we use the term *application configuration* to denote editing XML files, installing application servers, and other configuration related actions. Likewise, we define *feature selection* as the process of determining a valid set of configuration parameters (*i.e.*, filling in variabilities) with respect to a feature model's constraints.

The challenge with using existing model-based approaches, including feature models, for enterprise application configuration is that they often require a single large monolithic model of the system [8], [14], [3], [22], [23], [11]. Enterprise configuration decisions are often spread across multiple files, developers, and hosts, however, so it is time consuming to build and maintain accurate feature models. Moreover, the decentralization of enterprise application configuration decisions makes it easy for monolithic models to drift out of sync with the actual configuration state.

Some approaches advocate the use of multiple models [7], [4] that contain references to each other. This multi-model organization better mirrors the decentralized structure of enterprise application configuration and improves developer concurrency. The multi-model approach, however, requires that each role manually specify how changes to other roles' models affect elements in its own model. Manually specifying these effects is thus tedious and error-prone, as shown in Section II-C.

This paper describes how we created and applied an automated application configuration tool called *Fresh* to configure enterprise Java applications. Our Fresh approach uses a novel probe-based synchronization technique to allow each role to use its own feature model, while also not requiring manual cross-model effect specification and synchronization. Each probe is executable Java code that tests a property of the target environment (such as what libraries have been installed) and updates a role's feature model according to the results of the test (such as disabling or enabling a corresponding feature). As each role changes its feature selection and enacts changes on the application or target environment, Fresh probes translate the changes into feature modifications in any affected models. Roles synchronize models by describing how they affect and are affected by code and configuration changes to the application and target environment.

Fresh combines its multi-model approach with a constraint solver to reduce the complexity of enterprise application configuration. The key contribution of this paper is showing how Fresh simplifies enterprise application configuration by:

1) Automatically collecting the application's distributed configuration state with probes, *e.g.* determine the database installed, etc.
2) Phrasing the completion of the application's feature selection as a constraint satisfaction problem.
3) Deriving any remaining required features by solving the constraint satisfaction problem with a constraint solver, *e.g.* if a database driver is not installed determine which one is needed.
4) Rewriting the application's configuration files to include any new required features *e.g.*, add the database driver to the application configuration.

The remainder of the paper is organized as follows: Section II describes key challenges of configuring enterprise Java applications; Section III describes how Fresh frames the identification of valid configurations of enterprise Java applications as a constraint satisfaction problem and applies a constraint logic programming solver to automate and simplify application configuration; Section IV analyzes the results from experiments that quantify the reduction in configuration complexity gained by applying Fresh to configuring the Java Pet Store application; Section V compares Fresh with related research; and Section VI presents concluding remarks.

II. Challenges of Enterprise Java Application Feature Selection

This section first explores the various roles involved in configuring an enterprise Java application. It next examines the complex constraints and dependencies exposed by these roles. It then summarizes the challenges of deriving a configuration that integrates the configuration decisions of all of the roles and adheres to the application's functional and non-functional configuration constraints.

A. Example Enterprise Java Application: Pet Store

As a reference architecture of an enterprise Java application, we use the J2EE Pet Store application [2], which provides an example e-commerce site that allows customers to search for and purchase pets over the Internet. Pet Store was developed originally to showcase the benefits of J2EE technologies. Since its original release, nearly every major J2EE application server has included a refactored version of Pet Store as an example application. Microsoft has also reimplemented Pet Store (called Pet Shop) in .NET to highlight the differences between J2EE and .NET.

Since Pet Store is widely known and demonstrates the features of enterprise Java, we use it in this paper to show the configuration challenges of enterprise Java applications. To show the application's numerous points of variability we built a feature model of the Pet Store bundled with the Java Spring framework [16], which allows developers to create highly-modular and configurable enterprise Java applications. In particular, Spring uses (1) the factory pattern [13] to instantiate and interconnect enterprise Java components (beans) and (2) Java reflection to shield application components from details of the configuration process. At launch, a factory is created and initialized using one or more XML configuration files, which determine what components it constructs and how they are wired together. In the process of constructing objects, the factory may associate crosscutting aspect advice with them, generate dynamic proxies to perform remote invocations, load objects into a naming service, or perform numerous other complex application configuration tasks.

We bounded the scope of the feature model presented in this paper to a group of features related to the data tier of Pet Store. For example, in the feature model shown in Figure 1, the Pet Store can use either a *CombinedDatabase* setup, where both order and product data is stored in the same database, or a *DualDatabase* setup where product and order data are stored in separate databases. Depending on which setup is chosen, the Pet Store's application configuration files must be changed to include the appropriate Data Access Objects (DAOs). If a DualDatabase setup is used, developers alter the Pet Store configuration files to instruct Spring to instantiate and use the *JtaDAOs* and wire them into the application.

B. Challenges Produced by a Decentralized Configuration Process

For Pet Store—as with the majority of enterprise Java applications—the participants involved in feature selection can be divided into six roles [21]: enterprise bean (component) developer, web developer, client application developer, database developer, application assembler, and IT administrator (*e.g.*, application deployer and administrator). The numerous roles involved—and the complexity of the configuration constraints—thus make Enterprise Java applications prone to common configuration problems. Ideally, these errors should be identified when an application fails to load into its container properly. Often, however, these errors reflect subtle inconsistencies, such as incorrect file permissions, that may be overlooked and could lead to problems, such as security breaches.

Below we describe four major types of configuration challenges produced by the complexity of configuring an enterprise Java application that motivated our work on Fresh, as described in Section III.

a) Challenge 1: Feature selection complexity: The component dependencies or non-functional requirements are not adhered to when a feature set is selected because the large number of constraints, features, and roles involved makes it hard to derive a correct configuration. For example, the Spring Pet Store offers the ability to use a single or dual database setup and either plain Data Access Objects (DAOs) or Java Transaction API (JTA) enabled DAOs. As shown in Figure 2, if database developers choose to use the dual database setup then component developers must support transactions across multiple databases. This decision requires the use of JTA-

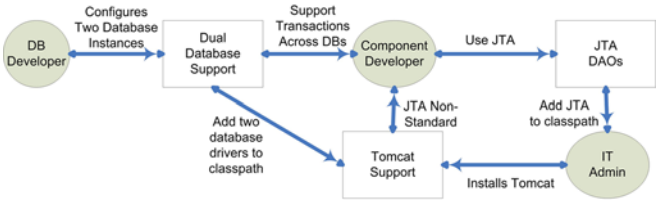

Fig. 2: Data Tier Feature Selection Forces and Their Effect on Various Roles

enabled DAOs, which prevents the Pet Store from running in a standard J2EE web container, such as Tomcat [5]. IT administrators must therefore either use a full-blown J2EE Application Server, such as JBoss [12], or configure the web container with additional components to support JTA. In this case, a decision made by database developers ripples through the functional composition decisions made by other roles. In general it is hard to take these complex dependencies and constraints across roles into account and derive a correct configuration.

b) Challenge 2: Incorrect feature selection implementation: After a feature set is selected, multiple configuration files must be edited and various actions (such as starting processes and adding message queues) taken by the roles to enable the desired features. For example, a non-functional variant can be produced if IT administrators do not edit application server XML configuration files properly to load the correct libraries or do not completely understand the requirements or implications of feature selection decisions. The non-functional

variant may fail to load properly into its container or load correctly but function incorrectly. As shown in Figure 3, to enable transaction support across databases with JTA, requires the coordination of multiple roles and configuration files.

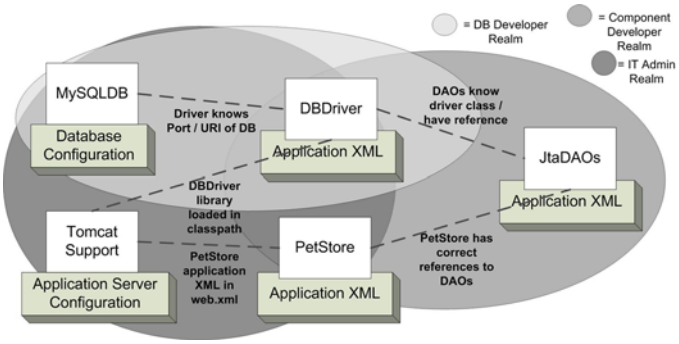

Fig. 3: Configuration Dependencies between Features and Roles for Data Tier Configuration

c) Challenge 3: Human mis-communication: Often, one or more roles misunderstand decisions made by other roles. Costly and hard to identify misunderstandings can involve environmental properties, such as application server platform and file permissions. For example, Pet Store provides both generic DAOs, which use only standard SQL mechanisms, and DAOs for Oracle and MSSQL, which use vendor-specific interfaces. The standard SQL DAOs will load properly into the Pet Store without errors regardless of the database vendor. The Oracle SequenceDAO, however, uses an Oracle-specific thread-safe sequence and may encounter runtime exceptions.

Failing to use the Oracle SequenceDAO with an Oracle database will not prevent the application from launching, but could potentially cause thread-safety problems. Thread safety problems are hard to diagnose Component developers may believe (incorrectly) that the application uses an MSSQL database instead of an Oracle database, thus causing a configuration problem that is dangerous *and* hard to identify. The mistake will therefore likely be identified only after incurring damage, such as data corruption.

Information may fail to flow across roles because participants do not understand which decisions impact other roles. In Figure 3, each role needs to understand where its realm of responsibility overlaps another role's realm of responsibility. In Pet Store, for example, IT administrators can enact decisions on the target infrastructure, such as selecting the component container that will be used. Component developers, however, may not have access to the target infrastructure or may be located in a separate office and thus may not be aware that IT administrators selected a specific container.

C. Limitations with Conventional Configuration Approaches

Various approaches [7], [4], [14], [3], [22], [23], [11] have been presented for configuring component applications using feature models and related mechanisms. Below we describe four types of limitations with current approaches that motivated our work on Fresh, as described in Section III.

Problem 1: Single model approaches. Conventional configuration approaches advocate the use of a single monolithic model [3], [22], [23], [11], where all configuration decisions are made in a single large model. Enterprise Java development involves multiple participants, which makes it hard to synchronize a single large model. The tight-coupling between roles also limits developer concurrency and does not integrate well with common development practices, such as eXtreme Programming, that focus on source code.

A further complication of tightly-coupled, single-model approaches are that they capture relevant information for each role's viewpoint in a single model. It is hard to capture all the information required for each viewpoint in an intuitive and usable manner. Moreover, a monolithic model potentially exposes participants from each role to irrelevant details from other roles. Even though different types of filtering mechanisms can be applied to limit what each viewpoint sees, developing these mechanisms is complicated since the complexity of the model may make it hard to predict which details are or are not relevant. Section III describes how Fresh uses a loosely coupled multi-model strategy to avoid the problems associated with a single model.

Problem 2: Manual synchronization and mapping of elements across models. Some approaches allow each role to use its own model, but require that any decisions made by a role that affect other roles be mapped manually to those roles' models [7], [4]. Most approaches, however, do not prescribe how the accuracy of these mappings is ensured or maintained. Mapping elements, such as features, across roles is problematic because it is hard for each role to anticipate which of its decisions will affect another role, what role it will affect, and how the effect will manifest in the other role's model. These dependencies between the decisions of different roles can only be enforced if they are expressed as mappings between models, which can be unwieldy in large organizations.

For example, it may be difficult to precisely map changes from one feature model to changes in another feature model. Even if each role can identify which decisions affect other roles, the effect of selecting a feature in one role's feature model must be evaluated from the viewpoint(s) of the other roles. Relating the effects of a role's feature selections to the features of other roles means that roles must relate features and decisions across viewpoints that they are not familiar with, which is tedious and error-prone. Section III-C describes how Fresh addresses this problem.

Problem 3: Not all necessary variabilities/decisions are captured in a model. Conventional approaches assume all decisions that are relevant to the configuration of an application are captured in the feature model [7], [4], [14], [3], [22], [23], [11], though they do not prescribe how this is accomplished. Documenting all decisions and variabilities is hard. In some cases, a role may not deem a variability important enough to its viewpoint to include it in the model. Another viewpoint, however, may be affected by this undocumented variability. The complexity of the model and the distinct separation of the roles' viewpoints makes it hard for each role to understand if

a variability should be documented for another role's sake. Section III-C, shows how Fresh addresses this limitation by using probes to gather feature configuration information directly from the target environment.

Problem 4: No runtime feedback. Conventional approaches [7], [4], [14], [3], [22], [23], [11] do not account for how dynamic application changes that affect the feature model can be identified and understood. If a container changes a runtime policy, it implicitly changes feature selections. Without some way to relate runtime changes back to the feature model, the model is only a design-time artifact. In this case, no feature decisions made by application containers or other runtime decisions can be constrained or understood.

In enterprise Java applications, it is undesirable to determine all application-related decisions at design-time. For example, the concept of *cloning* (*i.e.*, determining the number of instances of a feature) is a design-time decision in most approaches. In enterprise Java applications, containers normally manage object pools and dynamically change the number of instances (clones) of the objects at runtime. Other types of decisions, such as load-balancing policies, are also often best determined dynamically at runtime. In Section III-C, we present Fresh's approach to using probes to receive runtime feedback.

III. Solution Approach: An Automated Configuration Engine for Java Applications

This section describes the *Fresh* configuration engine and how it addresses the challenges of enterprise Java application feature selection described in Section II. Fresh frames the identification of valid configurations of enterprise Java applications for a specific target environment as a constraint satisfaction problem (CSP) and applies a constraint logic programming solver to automate and simplify application configuration. Fresh also uses code generation to rewrite application configuration files on-the-fly to implement the correct-by-construction configuration deduced by the constraint solver.

A. Addressing Configuration Challenges

As discussed in Section I, our Fresh approach to configuring enterprise Java application involves constructing a formal model of the feature decisions that have been made, determining what variabilities have been constrained, and setting values for the remaining component variabilities that are consistent with the constrained variabilities. This approach helps address the four challenges of a decentralized configuration process described in Section II-B and the limitations with existing methods of enterprise Java application configuration described in Section II-C, as follows:

1. Use probes to identify constrained variabilities. A probe is executable Java code that measures a property value of the application's environment or of the application itself. Fresh uses probes to automate the discovery of decisions made by each role, which allows the feature selection process to ensure that the selected feature set conforms to points of variability that are already constrained. To prevent one role from misunderstanding the configuration decisions made by another role, Fresh uses probes to automatically identify what configuration decisions have been made.

For example, if database developers have chosen Oracle, component developers cannot mistakenly think that MSSQL was chosen if a probe is used. Just as unit tests can be written to test the functionality of features, probes can be created for each feature to validate dependent features and properties.

2. Formalize configuration as a CSP and use a constraint solver to derive values for unconstrained variabilities. Fresh's constraint solvers can handle the combinatorial complexity and interdependencies of feature selection more effectively that a manual process, thereby addressing challenge 1 from Section II-B. Moreover, a constraint solver will produce a correct selection that honors the constraints, assuming a correct configuration exists.

3. Generate configuration files from a feature selection. Fresh uses code generation to create correct configuration files automatically from the solution produced by the constraint solver. This approach allows developers to annotate their configuration files to show how features are bound to actual configuration decisions. Moreover, this design allows the Fresh configuration engine to regenerate the configuration files when the feature selection changes.

B. Overview of Fresh and Its Feature Modeling Capabilities

We use the Fresh feature selection engine to demonstrate our approach for automating the collection of feature modeling decisions, phrasing a feature selection problem as a CSP, and using a constraint solver. Each role can use Fresh to describe the functional and non-functional requirements of enterprise Java application configuration, along with a fitness function for choosing a configuration when multiple solutions exist. Fresh combines this information with the Choco constraint logic programming solver [1] to derive a complete feature selection for a partially configured application.

Fresh also provides an XML annotation language that can inject the feature selection decisions into XML configuration files. These files then determine what application components are loaded, how they are wired together, and what values are set for their configurable properties. The XML annotation language maps features to elements and attributes in XML configuration files.

Fresh provides two interceptors that can be used with the factories in the Spring framework. When a Spring application factory attempts to load an application's configuration files, a Fresh interceptor intercepts the call, probes the environment, runs the constraint solver, and rewrites the configuration files before they are returned to the Spring factory. Spring and the application components are therefore not aware of Fresh or that it is dynamically deducing an application's configuration. Moreover, the Fresh interceptor can be inserted/removed to/from applications without affecting their components or Spring. One of the two interceptors is designed to be used with Spring's Java servlet boot-strapping mechanisms and is the interceptor used for the experiments presented in Section IV.

Initially, the roles manually perform some configuration steps but do not completely configure the application. In step one of the Fresh configuration process, automated probes are run to discover the manual configuration actions that have been taken and how they map to the feature model of each role, as shown in Figure 4. In step two the decisions and feature

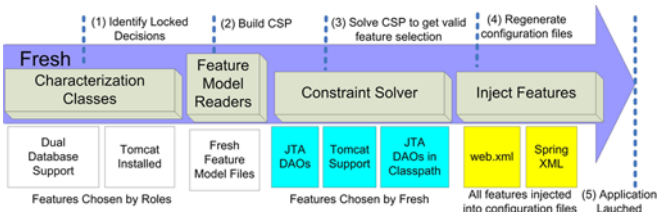

Fig. 4: Fresh Application Configuration Process

model roles are transformed into a CSP. In step three, Fresh uses the Java Choco constraint logic programming solver to solve the CSP for a valid feature set. The resulting feature set will indicate what further configuration steps are needed to complete the application's configuration. In step four, the configuration files for the application are regenerated with required additional components to complete the application's configuration. Finally, in step five control is passed to the Spring factory to initialize the application.

Fresh currently collects and solves for configurations (if no conflicts are present). If a valid feature selection that integrates all of the roles' decisions cannot be found then Fresh fails to start and notifies the user that no valid configuration exists. Priorities and interactivity (where the developer is prompted to resolve conflicts) are future work.

Fresh's configuration file annotation language is based on XML comments and does not interfere with the configuration file's directives. The annotation language can be used with any XML file and is not specific to the file formats used by Spring. These annotations can be added to existing files or removed from the application entirely without affecting it. Spring and application components developed atop it are therefore decoupled from Fresh's container extension and the XML annotations.

C. Using the Target Environment as a Common Language

As described in Section II-C, conventional techniques for enterprise Java application configuration fail to address key challenges, such as a configuration process must be able to relate how the actions of different roles affect each other. Previously developed approaches either attempt to use a single manually-produced large model to capture these interactions formally or rely on manually creating complex mappings across different models. The first approach suffers from the problems of a complex top-down approach, whereas the second approach forces the roles to specify complex cause-and-effect relationships explicitly across unfamiliar viewpoints.

Fresh's probing uses the target environment as a *lingua franca*. Each role expresses how changes in the target environment affect its model of the system. A probe checks a property of the environment and maps the property to a change in a role's model. For example, a probe can be used to detect automatically if JTA is installed and update the JTA feature in the component developer's model accordingly.

A benefit of the Fresh approach is that it avoids monolithic top-down modeling (problem 1 of Section II-C). Each role can use a model that is intuitive to the role's viewpoint. The models of each role are synchronized when the probes are run. The probes determine the changes that the roles have made to the target environment and update each role's model to reflect the configuration state. Each role therefore maintains a model reflecting its viewpoint and is not tightly-coupled to the models of other roles. Fresh currently does not resolve conflicts between models but instead notifies the deployer when there is no valid configuration that integrates all of the roles' decisions and constraints.

Another benefit is that the roles need not explicitly describe how changes in their models map to changes in the models of another viewpoint (problem 2 of Section II-C). Instead, each role specifies how changes to the target environment affect it. Since the mappings are based on executable code, the mappings have precise semantics. The mappings also do not require a participant in a role to understand another role's viewpoint. Each viewpoint instead maps its feature selections to changes in the target environment and each role's probes translate the environment modifications into changes in the role's model. The environment serves as the common language, as shown in Figure 5.

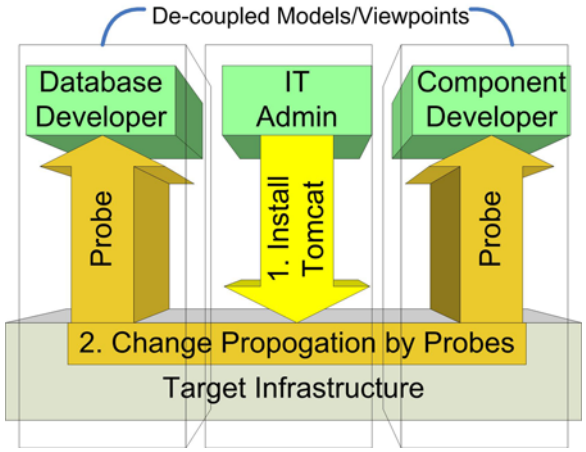

Fig. 5: Synchronizing Role/Viewpoint Models through Probes

Yet another benefit of the Fresh approach is that probes do not differentiate between human induced environmental changes and dynamic changes to the environment from containers or other runtime actors. Containers become another participant that may enact changes to applications at runtime. Since the probes are automated, they can be reused at runtime to detect changes to each role's feature model produced by the container. Runtime processes can become roles that provide feedback to the application (problem 3 of Section II-C).

Since the dissemination of information across roles is au-

tomated by the probes, Fresh can help address challenges 3 and 4 of Section II-B. Automated probes are more reliable than human inspection of the configuration and environment. Rather than pushing information to the roles that are affected by changes in the target environment, the probes pull the required information to each role, thus avoiding communication failures and misunderstanding.

D. Probing the Target Environment

The probes run by Fresh identify which features or components are present (*e.g.*, is JTA installed), what the values are for different properties of the target infrastructure (*e.g.*, application server vendor, OS, RAM, etc.), and what configuration steps have been performed (*e.g.*, does a specific JMS queue exist). The probes produce a series of values for the variabilities in the model. For example, if JTA is installed, a probe may enable the JTA feature or the JTAVersion attribute.

Fresh uses a plug-in architecture so application developers can create *characterization classes* to package with an application and run by Fresh to automate environment characterization. Each characterization class is a probe that determines the value of one or more of the variabilities in the model used for the configuration process. Before Fresh performs its constraint-based feature selection, each characterization class is invoked. A characterization class performs a test on the target environment and returns a list of variable/value pairs representing the target characteristics.

The values of the variables produced by characterization determine what points of variability have already been constrained by each role. Fresh then derives values for the other variabilities that are correct with respect to these fixed points and the feature model constraints. The following are examples of how Fresh can discover configuration decisions using characterization classes:

- *Local/remote addressing configuration.* For external addressing, such as JNDI names or service URIs, characterization classes can be created that attempt to resolve the object and if it cannot be resolved, disable the corresponding feature.
- *Library configuration.* Characterization classes use the Java Reflection API to resolve references to classes that a feature depends on. For example, to test for JTA, a characterization class can invoke `Class.forName("javax.transaction.Transaction")`, which throws an exception if JTA is not present.
- *Attribute configuration.* A characterization class can obtain values for various attributes from environmental context classes, such as `java.lang.Runtime`, `ServletContext`, or `ApplicationContext`. These context classes can provide critical infrastructural attributes, such as JVM version, OS, RAM, etc., for the CSP variables. A characterization class may also determine attribute values by instantiating one or more application components and using getter methods or the Java Reflection API to obtain member variable values.
- *Infrastructure configuration.* Characterization classes can be used to test that specific infrastructural features are running. For example a class can be created that attempts to connect and post a message to a required JMS queue or run a query against a database table. If the queue does not exist or an exception is thrown the feature variable for the queue can be disabled. Similarly, the database configuration can be checked by creating a class that obtains an instance of the DB driver and attempts to perform queries to check that the tables are configured properly.

The list above is not exhaustive. Numerous other types of characterization classes, such as running a CPU benchmark, can be used to obtain complex properties. In most cases, if the application is affected by a configuration decision, it can probe its environment to determine the value of that point of configuration variability.

Class characterization allows the Fresh feature selection engine to determine what variabilities have been constrained in the product. After correctly determining what variable parts are fixed, the constraint solver can select features to ensure the application functions properly with respect to these fixed parts and the application requirements.

E. Feature Selection as Constraint Satisfaction

Challenge 1 in Section II-B explains why the process of configuring enterprise Java applications is complex due to the large number of constraints and role viewpoints involved. Significant work [22], [24] has been done in applying different algorithmic techniques to manage this complexity. The probing techniques that we described in Section III-D can be used with these algorithmic approaches. For Fresh, we chose to apply the extensive research and tools for constraint logic programming [15] to manage this complexity.

Fresh transforms a feature model and set of non-functional requirements into a CSP. The feature model and the non-functional requirements are specified through Fresh configuration files that reside in the classpath of the Spring application. We use a reduction of feature selection that enhances the work of Benavides et al. [3] (see Section V for a comparison). By building a formal model of feature selection as a CSP [27], Fresh can use a constraint solver to (1) check the correctness of a configuration and (2) derive valid values for unconstrained variabilities in a partially configured application, which addresses challenge 1 from Section II-B.

Selecting a feature set for an application can be reduced to a constraint satisfaction problem. Fresh constructs a set of variables $P_0 \ldots P_n$, with domain $[0,1]$, to indicate whether or not the i^{th} feature is present in a feature set. A feature set thus becomes a binary string where the i^{th} position represents if the i^{th} feature is present. Satisfying a constraint satisfaction problem for feature selection involves devising a labeling of $P_0 \ldots P_n$ that adheres to the composition rules of the feature model.

The constraints in a feature model ensure that only a coherent set of features is selected. For example, if the *JtaDAOs*

feature (*i.e.*, DAOs that use the Java Transaction API) is chosen in the Pet Store application, the *JTA* feature must also be selected. To phrase this rule using the constraint satisfaction model of feature selection, we can say that if the JtaDAOs feature is represented by the variable P_1 and the JTA feature is represented by the variable P_2, then $P_1 = 1 \rightarrow P_2 = 1$. The CSP can also encode non-functional constraints, such as JtaDAOs requires that the target environment have at least 128mb of memory: $P_1 = 1 \rightarrow Env_{memory} \geq 128$. Env_{memory} is a variable introduced to store the amount of memory on the target host.

Constraint satisfaction models may incorporate constraints based on the conjunction or disjunction of several constraints on other features. One example of this approach is the extension to cardinality constraints on features proposed in [9]. Their approach extends cardinality constraints to include a sequence of intervals. For example, assume that the Pet Store can use [1..2] or [4..4] different remoting mechanisms from the remoting feature group. If the variable P_0 represents the Pet Store, and the variables $P_{15} \ldots P_{18}$ represent the remoting features, we can transform this interval sequence into the constraint: $P_0 = 1 \rightarrow (\sum Pt_{15} \ldots P_{18} > 0) \wedge (\sum Pt_{15} \ldots P_{18} \leq 2) \vee (\sum Pt_{15} \ldots P_{18} = 4)$.

F. Aggregating Feature Models and Feature Requirements

During application startup, Fresh inspects one or more directories that contain the feature models for each role, non-functional requirements, and configuration mechanisms for the application. It then constructs its constraint satisfaction problem by composing the feature models of each viewpoint and the non-functional requirements it discovers. Adapters are used to load the feature model and non-functional requirements. By default, Fresh provides adapters for reading feature models and non-functional requirements that use a syntax similar to cascading style-sheets. Adapters can be plugged-in to read other formats, such as XMI models produced by the Eclipse Modeling Framework (EMF) [6].

Since specifying feature dependencies and constraints using constraint satisfaction syntax can be overly complicated for many application developers, we developed a Domain-Specific Language (DSL) [18] for specifying feature models and constraints. The feature modeling language, called *Feature Styles*, allows application developers to specify the features in the model, the dependencies between features, and the non-functional requirements associated with each feature. The language uses a simple textual notation and is intuitive for developers who understand basic logic, *e.g.*, A requires B, B excludes C, etc.

Fresh supports the following constraint types:

- *Required* features that must be present for a feature to function properly. For example, JTADAOs *requires* JTAEnabled.
- *Excluded* features that cannot be present at the same time as a feature. For example, OracleSupport *excludes* SQLSequenceDAO.
- *Cardinality* constraints on required features. For example, OrderRemoting requires a user to *select* [1..*] of the features HessianRemoting, RMIRemoting, and BurlapRemoting.

Application developers use these dependency rule types to build complex feature models for a product. Section III-E describes how these rules are translated into a constraint satisfaction problem [27] for a Java Constraint Logic Programming (CLP(X)) solver [15]. The non-functional requirement specification language of Feature Styles allows product developers to specify constraints on the properties of the target environment. For example, the constraint $RAM > 128$ would allow a feature to only be enabled if the target had more than 128mb of RAM. The constraints can reference the value of any property that can be deduced from a probe. Fresh provides constraints based on conjunctions or disjunctions of $>, <, =, !=, =<, >=$.

A feature can be annotated with any number of constraints on the attribute values. Component developers use these constraints to encode the non-functional requirements of the features. As with the feature dependency rules, the constraints are encoded into the CSP provided to the feature selection engine.

The full feature specification for the JtaDAOs is shown below:

```
JtaDAOs {
  Requires: JTA, DatabaseDriver;
  Excludes: NonJtaDAOs;
  JTAVersion > 1.01;
  JTAVersion < 1.03;
}
```

IV. Results from Experiments with Fresh

To demonstrate the reduction in manual configuration complexity provided by Fresh, we devised a realistic configuration scenario for the Pet Store example in Section II-A. In this scenario, Pet Store has a base deployment descriptor (the out-of-the-box descriptor included with the Spring Pet Store) that must be modified to install the Pet Store on Tomcat with an Oracle Database, Email Notification, and RMI Remoting. Pet Store is then migrated to a new target where it is hosted on JBoss with an MSSQL database, no RMI Remoting (to avoid conflicts with the application server), and no Email Notification (email order notification is handled by a new payment processing application when the customer's credit card has been charged). The results in this section show that Fresh's automated configuration approach can reduce the total number of steps required to configure an enterprise Java application by 72% and the total lines of XML code by 92%.

A. Testing Configuration Complexity

In the test scenario, we compute the configuration cost in lines of XML code that must be changed. We assume that optional components, such as Email Notification's Email Advice, are not initially present in the deployment descriptor. When a role selects a feature requiring a component, the component is added to the configuration files. Table I shows the steps involved in configuring the Pet Store for the first deployment configuration.

Steps for Initial Deployment	Lines of XML Changed	Roles Involved	Location of Change
1. Change Datasource Driver Class/URI	1	Database Dev/IT Admin	dataAccessContext.xml
2. Remove Standard Sequence DAO	3	Database Dev/IT Admin	dataAccessContext.xml
3. Add Oracle Sequence DAO	3	Database Dev/IT Admin	dataAccessContext.xml
4. Add Mail Sender Bean to application.xml	3	IT Admin	application.xml
5. Add Insert Order Pointcut	1	Component Dev	application.xml
6. Add Email Advice	3	Component Dev	application.xml
7. Add RMI Remoting Service Export	6	Component Dev/IT Admin	application.xml
Total Steps: 7	Total Lines of XML: 20	Roles Involved: 3	Files Involved: 2
Steps for Second Deployment			
1. Change Datasource Driver Class/URI	1	Database Dev/IT Admin	dataAccessContext.xml
2. Remove Standard Order DAO	3	Database Dev/IT Admin	dataAccessContext.xml
3. Add MSSQL Order DAO	3	Database Dev/IT Admin	dataAccessContext.xml
4. Remove Oracle Sequence DAO	3	Database Dev/IT Admin	dataAccessContext.xml
5. Add Standard Sequence DAO	3	Database Dev/IT Admin	dataAccessContext.xml
6. Remove RMI Service Export	6	Component Dev/IT Admin	application.xml
7. Remove Mail Sender Bean	3	IT Admin	application.xml
8. Remove Insert Order Pointcut	1	Component Dev	application.xml
9. Remove Email Advice	3	Component Dev	application.xml
Total Steps: 9	Total Lines of XML: 26	Roles Involved: 3	Files Involved: 2

TABLE I: Cost of a Manual Approach to Configuration for the Scenario

As shown in Table I there are many steps, roles, and files involved. To migrate to the second target environment, the roles must remove some of the initially chosen components (*e.g.*, Oracle Sequence DAO, Email Advice, Order Pointcut, etc.) and add other new components (*e.g.*, MSSQL Order DAO). The steps involved in the migration are shown in Table I.

Table I also shows that there are a significant number of steps and changes required to migrate to the new setup. Each change in the target environment or desired feature set will necessitate similar reconfiguration costs. Moreover, if the application is widely used, the support team for each application *instance* must pay this configuration cost.

We then performed the same migration experiment using Fresh. Fresh required an extra initial investment of building a basic feature model for the features from the migration experiment. It also required the addition of comments to the Pet Store's XML configuration files that mapped features to XML configuration directives (so that the configuration files could be regenerated). The initial Fresh configuration overhead is shown in Table II.

Fresh requires an initial overhead of 33 lines of XML/Feature Model configuration. This extra configuration code allows Fresh to (1) detect the database type used (inferred from the data source driver class), (2) detect if a web container or application server is the container (by checking for EJB-specific classes), and (3) add/remove XML configuration directives for the components of enabled/disabled features, respectively. Although the initial cost of enabling Fresh is higher than a traditional manual approach, this price is paid only once, rather than each time the application is deployed.

Table II shows the steps required for installing the Pet Store on the initial target with Oracle and Tomcat. Only two configuration steps are required. First, the correct database driver class is added to the configuration and then the desired feature set is specified as Tomcat, Oracle, etc. Fresh performs all other XML configuration tasks, including deriving a valid feature selection with respect to the desired features.

Table II also summarizes the steps required to perform the second migration to the JBoss/MSSQL environment. Again, only two steps are required: setting the database driver and updating the desired features. These two steps provide a significant improvement over the manual approach, where 26 lines of XML were changed for the same migration.

Table III compares the totals for the manual vs. Fresh configuration approaches. Fresh initially incurs a marginal configuration cost for building a feature model and annotating the XML configuration files for the Pet Store. After the migration to the second target environment, however, Fresh reduced the complexity of configuring the Pet Store by 9 lines of XML configuration. Moreover, for each configuration, Fresh derived a valid feature set based on the desired features specified by the roles. With a manual approach, this derivation is not automated and can produce numerous types of errors, as shown in Section II-B. In contrast, Fresh assures that each configuration is correct by using a constraint solver to derive a configuration based on the feature model constraints and constrained variabilities.

When the cost of configuring the Pet Store over 100 separate deployments is analyzed, the benefits of the Fresh approach are amplified. At the minimum (assuming that each deployment uses the default configuration), the manual approach requires 200 configuration steps and 600 lines of XML changes. The total cost of the manual approach can be over 900 configuration steps and 2,600 lines of XML code, however, if the default configuration is not used on each deployment, which we assume is common.

With Fresh, conversely, the total configuration steps are fixed at 209 and the total lines of XML configuration at 233. At a minimum Fresh requires 62% less lines of XML configuration changes and a maximum of 92% less. Stepwise, Fresh uses at most 4.5% more steps but can also use 72% less total steps. As the number of deployments of the Pet Store increases, Fresh's development savings also increase. With increased numbers of deployments, the initial investment cost of Fresh becomes insignificant compared to the savings.

The intial cost paid to enable Fresh is incurred by the orig-

Steps to Enable Fresh	Lines of XML Changed	Roles Involved	Location of Change
1. Build Fresh Feature Model	6	Component Dev/IT Admin/Database Dev	petStoreFeatureModel.xml
2. Add Application Server Detection Probe	1	Component Dev	probes.xml
3. Add Database Detection Probe	1	Database Dev	probes.xml
4. Make Sequence DAO Switchable	4	Component Dev	dataAccessContext.xml
5. Make Order DAO Switchable	4	Component Dev	dataAccessContext.xml
6. Make Mail Sender Switchable	4	Component Dev	application.xml
7. Make Insert Order Pointcut Switchable	2	Component Dev	application.xml
8. Make Email Advice Switchable	4	Component Dev	application.xml
9. Make RMI Remoting Service Switchable	7	Component Dev	application.xml
Total Steps: 9	Total Lines of XML: 33	Roles Involved: 3	Files Involved: 4
Steps for Initial Deployment			
1. Change Datasource Driver Class/URI	1	Database Dev/IT Admin	dataAccessContext.xml
2. Change Desired Features	1	IT Admin	dataAccessContext.xml
Total Steps: 2	Total Lines of XML: 2	Roles Involved: 2	Files Involved: 1
Steps for Second Deployment			
1. Change Datasource Driver Class/URI	1	Database Dev/IT Admin	dataAccessContext.xml
2. Change Desired Features	1	IT Admin	dataAccessContext.xml
Total Steps: 2	Total Lines of XML: 2	Roles Involved: 2	Files Involved: 1

TABLE II: Fresh Configuration Cost for the Scenario

	Total Steps	Lines of XML Changed	Total Roles Involved	Files Changed
Initial Overhead				
Manual	0	0	0	0
Fresh	9	33	3	4
Configuring for Tomcat/Oracle				
Manual	7	20	3	2
Fresh	2	2	2	1
Migrating to JBoss/MSSQL				
Manual	9	26	3	2
Fresh	2	2	2	1
Scenario Totals				
Manual	16	46	3	2
Fresh	13	37	3	4
Configuration Cost Per Deployment				
Manual	2min to 9+max	6min to 26+max	3	2
Fresh (not counting initial overhead)	2	2	2	1min to 2max
Total Configuration Cost Over 100 Deployments				
Manual	200min to 900+max	600min to 2600+max	3	2
Fresh (including initial overhead)	209	233	3	4

TABLE III: Manual vs. Fresh Configuration Cost Totals

inal application developers. Applications are often developed by one group, yet have hundreds or thousands of instances installed and maintained by other groups, *e.g.*, testers and users. Moreover, the users often perform the final configuration, such as choosing the database, OS/middleware version, network configuration, etc. These users rarely possess the same intimate knowledge of the application, so they are more likely to make errors or produce poor configurations. With Fresh, conversely, the initial developers can package their intimate feature model, non-functional requirement, and configuration knowledge with the application.

Since this expert configuration information is packaged with the application, users focus on declaratively informing Fresh what they want, rather imperatively programming new configurations to provide what they want. Application users can therefore benefit from the expert configuration knowledge of the original developers, which is much harder with conventional manual approaches. Moreover, Fresh greatly reduces the configuration cost for users since they do not pay the initial Fresh integration cost, which is borne by the original application developers.

B. Fresh Performance Overhead

To determine the performance penalty for deriving a configuration with a constraint solver and rewriting an application's configuration files, we built a set of experiments to test the startup time of Pet Store. We first devised several new feature models of increasingly finer granularity to see how long application startup took with varying feature model sizes. Feature models of 60, 80, and 100 features were created. The 60, 80, and 100 feature models were actual feature models of the Pet Store. The 60 feature model did not account for features related to the web-tier of the Pet Store. The 80 feature model added features for the web-tier and Spring's Web Flow front end. The 100 feature model added features for the alternate Apache Struts front-end of the Pet Store's web-tier.

Each test was built so that the feature set derived from Fresh would lead to an identical application configuration, *i.e.*, produce the same set of XML configuration directives.

We also reproduced this configuration statically in XML to launch without Fresh and derive the overhead incurred by using Fresh. We launched Pet Store in Tomcat 6.0.9 using JDK 1.5.0_11 on an IBM Think Pad T-43 with a 1.86GHZ Pentium M processor, 1.5GB of RAM, and Windows XP. We then tested the time needed to launch Pet Store within Tomcat and configured it using Fresh with each feature model. The results were compared to the static configuration launched in Tomcat without Fresh and are shown in Figure 6.

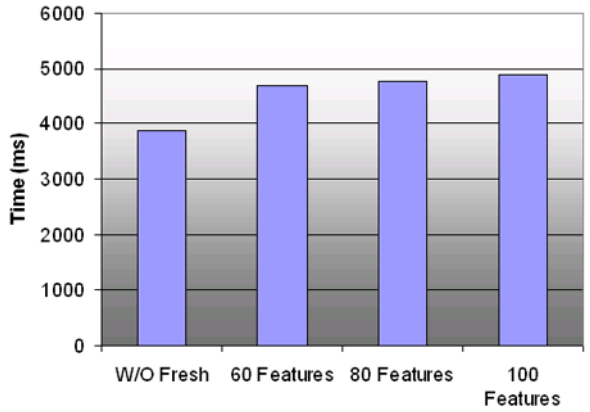

Fig. 6: Pet Store Initialization Time in Tomcat

Figure 6 shows that using Fresh with a 60 feature model required an extra ∼800ms to launch vs. a static configuration. For 100 features, the total penalty was ∼1,000ms. This overhead should be acceptable for many enterprise Java application deployment scenarios because it is only incurred once at application startup.

V. Related Work

A mapping from feature selection to a constraint satisfaction problem (CSP) is provided by Benavides et al. [3]. Fresh uses this same reduction but also extends it with the capability to handle feature references, cardinality constraints [9], and resource constraints. Moreover, the approach presented in [3] does not address the challenges presented by the decentralized nature of enterprise Java configuration outlined in Section II-C. Mannion et al. [20] also present a formal method for specifying Product-Line Architecture (PLA) compositional requirements using first-order logic. The correctness of a variant can then be tested by determining if a PLA satisfies a logical statement. Our Fresh approach to feature selection goes beyond the approaches presented by Mannion and Benavides. Fresh provides the ability to have multiple models that are automatically synchronized through probes. Both Mannion and Benavides approach uses a single model, which suffers from the problems outlined in Section II-C. Fresh also provides an XML annotation language specifically designed for enterprise XML configuration files, such as enterprise Java deployment descriptors. Finally, Fresh can leverage probes to provide runtime feedback from the application.

Pure::variants [4] is a commercial tool similar to Fresh that provides feature modeling capabilities. Pure::variants allows developers to specify features and feature constraints, validate feature selections, and to derive required completions of a feature selection. Pure::variants requires developers to manually specify how features from one feature model affect features in another feature model. Fresh, in contrast, automates the synchronization of feature models through probes, which as we have shown, is an important capability for enterprise application capability. Fresh's automated model synchronization through probes reduces manual mapping errors and increases developer concurrency by allowing multiple models. Furthermore, Fresh's probes can be used to receive runtime feedback to the feature model. Pure::variants does not support automated runtime feedback.

BigLever *Software Gears* [7] is another commercial feature modeling and software variant management tool. Software Gears supports features similar to Fresh including: feature modeling, automated feature selection completion, and configuration injection. BigLever requires manually developed mappings between features. As we have discussed, Fresh uses probes to help eliminate problems from manually produced cross-model feature mappings. Again, using probes allows Fresh to also automate the collection of the application's configuration state and receive runtime feedback.

Various approaches [22], [23] have been devised to handle the complexity of configuring applications. Other techniques have also been proposed for variant configuration in PLAs based on configuration rules for application components [26]. This related work focuses on how a configuration problem can be formalized as a CSP. Our work on Fresh extends these ideas, particularly those that describe a generic model of configuration as a CSP [22]. These approaches provide key building blocks of automated product configuration, but do not address the specific challenges related to decentralized enterprise Java configuration processes. These existing approaches also have not been integrated into enterprise Java frameworks to provide dynamic autonomous feature selection and application configuration at startup or runtime. In contrast, Fresh automates both the collection of configuration state through probes and can inject configuration decisions by regenerating application configuration files.

VI. Concluding Remarks

This paper demonstrated how the Fresh feature selection engine simplified enterprise Java application configuration at launch time via its use of automated probes, feature modeling, a constraint solver, and configuration file generation. By using an automated configuration approach at startup, Fresh can produce a coherent model of the diverse configuration decisions that the development roles have enacted. The results presented in Section IV show that Fresh can reduce the total number of lines of configuration XML code by between 62% and 92% and the total number of configuration steps by up to 72%.

From our experience applying Fresh's probing and constraint-based configuration capabilities to the Pet Store and

other representative applications, we learned the following lessons:

A constraint solver can derive a correct application configuration. Fresh alleviates the problems described in Section II-B by executing a series of Java probes at application launch to identify constrained variabilities, formalizing and solving a constraint satisfaction problem of the configuration problem, and dynamically rewriting the application's XML configuration files. The information on functional and non-functional properties collected by automated probing can be treated as a constraint satisfaction problem and a correct application configuration derived by using a constraint solver. Moreover, the constraint solver can produce a solution that is correct with respect to both the feature model and the decisions made by the roles.

Automated probing adds little complexity. Probes did not add significant complexity to Fresh's automated configuration approach. An application typically requires a probe for each point of variability. In some cases, a probe may be needed for each individual feature. In other cases, a single probe can identify what features are enabled in an entire feature group. As with unit test frameworks, such as JUnit, probes are relative straightforward to write. Although unit tests can often comprise a substantial amount of code compared to the application itself, this was not the case for probes.

Probes are configuration unit tests. Even if a full constraint-solver based solution is not deemed needed, using a configuration probing infrastructure can be useful. Creating probes to ensure that individual points of configuration are properly fixed can help improve the guarantees that an application is installed and configured properly. Since application misconfiguration contributes to a significant portion of application failures [10], developers should consider the use of automated configuration checking.

When roles have conflicting models or decisions, debugging the problem is hard. An aspect of complexity that Fresh does not yet not address is how to debug configuration errors. For example, if an IT administrator chooses Tomcat without JTA and component developers choose the JtaDAOs, which participant's action should be flagged as the error? In many cases, there may be numerous intermediary implications causing the conflict. When there are no possible values for the unconstrained variabilities to create a correct configuration, it is hard to provide meaningful feedback to participants as to why a correct configuration cannot be found. Fresh's automated configuration approach can catch configuration errors and reduce the complexity of completing many partial configurations. When a configuration cannot be completed, participants are still left with a complex process of deducing why the configuration failed. In future work, we plan to explore how tools and techniques developed for explanation-based reasoning and diagnosing the problems in over-constrained systems can be used to automate configuration failure diagnosis.

Fresh is available in open-source form as part of the GEMS project at www.sf.net/projects/gems.

REFERENCES

[1] Choco constraint programming system. http://choco.sourceforge.net/.

[2] The Java Pet Store. http://java.sun.com/developer/releases/petstore/.

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

[4] D. Beuche. Variant management with pure:: variants. Technical report, Pure-systems GmbH, http://www. pure-systems. com, 2003.

[5] J. Brittain and I. Darwin. *Tomcat: The Definitive Guide*. O'Reilly Media, 2003.

[6] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, Reading, MA, 2003.

[7] R. Buhrdorf, D. Churchett, and C. Krueger. Salion's Experience with a Reactive Software Product Line Approach. *Proceeding of the 5th International Workshop on Product Family Engineering. Nov*, 2003.

[8] I. Crnkovic. Component-based software engineering-new challenges in software development. *Information Technology Interfaces, 2003. ITI 2003. Proceedings of the 25th International Conference on*, pages 9–18, 2003.

[9] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2):143–169, 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 Techn̈ologies and Systems*, March 2003.

[11] G. Edwards, G. Deng, D. Schmidt, A. Gokhale, and B. Natarajan. Model-Driven Configuration and Deployment of Component Middleware Publish/Subscribe Services. *Generative Programming and Component Engineering (GPCE)*, pages 337–360, 2004.

[12] M. Fleury and F. Reverbel. The JBoss Extensible Server. *International Middleware Conference*, 2003.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[14] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: Configuration and Automatic Ignition of Distributed Applications. *HP Openview University Association conference*, 2003.

[15] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *constraints*, 2(2):0.

[16] R. Johnson and J. Hoeller. *Expert One-on-One J2EE development without EJB*. Wiley Pub., 2004.

[17] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute, Technical Report CMUSEI90TR21, Carnegie Mellon University, 1990.

[18] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[19] G. Lenz and C. Wienands. *Practical Software Factories in .NET*. Apress, Berkeley, CA, 2006.

[20] 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.

[21] V. Matena, S. Krishnan, B. Stearns, and L. Demichiel. *Applying Enterprise Javabeans: Component-based Development for the J2EE Platform*. Addison-Wesley, 2003.

[22] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 2:1395–1401, 1989.

[23] D. Sabin and E. Freuder. Configuration as composite constraint satisfaction. *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[24] D. Sabin and R. Weigel. Product configuration frameworks-a survey. *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, 13(4):42–49, 1998.

[25] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[26] T. van der Storm. *Variability and Component Composition*. Springer, 2004.

[27] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press Cambridge, MA, USA, 1989.