Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

Automated Configuration of Component-based Distributed Real-time and Embedded Systems from Feature Models

Jules White and Douglas C. Schmidt

Vanderbilt University
Department of Electrical Engineering and Computer Science
e-mail: {jules,schmidt}@dre.vanderbilt.edu).

Abstract: Component-based distributed real-time and embedded (DRE) systems facilitate the reuse of software artifacts across applications. To achieve a high-level of reuse, component-based DRE systems leverage late binding to allow dynamic system assembly at deployment time (*e.g.*, via configuration scripts) rather than statically at compile-time. The complexity of deriving a correct manual configuration of an arbitrary set of components, however, is a key source of system failures, downtime, and missed deadlines.
This paper presents a model-driven engineering tool called Fresh that uses (1) feature models to codify the configuration rules of components, (2) a constraint solver to derive a correct application configuration, and (3) an XML annotation engine to inject configuration decisions from the constraint solver directly into an application configuration. We use an avionics mission computing case study based on the Lightweight CORBA Component Model (CCM) to (1) demonstrate the complexities of configuring components in/out of a component-based application and (2) motivate the reduction in the configuration complexity when Fresh is used. The results show that Fresh achieves ∼80-90% reduction in manual configuration effort, while also ensuring that the derived configuration is correct with respect to application configuration constraints and QoS requirements.

1. INTRODUCTION

Distributed real-time and embedded (DRE) systems are increasingly being built using component-based technologies. Component technologies facilitate software reuse across applications by allowing the dynamic assembly of applications at deployment time via configuration scripts. The late-binding properties of component technologies allow application developers to reuse existing software and reduce costs by leveraging commercial-off-the-shelf (COTS) components.

Application developers have traditionally used tightly-coupled proprietary solutions to handle the tight requirements and resource restrictions of DRE systems. Composing a component-based application from components that are not specifically designed for the individual application poses a number of challenges. For example, highly specialized components can make assumptions, such as the what type of underlying operation system will be used, that reusable components cannot make. These assumptions can help improve performance (*e.g.* using specialized APIs) at the cost of reusability. Because DRE systems often operate in environments with little resource slack, being unable to make these key assumptions makes it difficult to find a configuration that meets the required timeliness, safety, and other non-functional properties.

A further challenge of configuring DRE systems is that the configuration process must integrate the concerns of numerous participants divided into multiple roles, such as component developers and hardware developers. Each role has a unique viewpoint on what it considers the ideal solution. Thus, each role attempts to pull the solution in the direction that best meets the requirements it is responsible for, such as power consumption or security functionality. These multiple opposing viewpoints make it hard to find a configuration that satisfies the requirements of each role simultaneously.

For example, in applications developed using the Lightweight CORBA Component Model (CCM) (BEA Systems, et al. [1999], Wang et al. [2000a]), component developers often prefer to host the applications on the most powerful processing hardware available and be allocated as much network bandwidth as possible to make their realtime scheduling deadlines easier to meet. Hardware developers, in contrast, will attempt to use the least powerful processors that are adequate for the job to minimize power consumption, weight, and cost to make the system more efficient. Component assemblers (the role that creates instances of components and wires them together) will want to have the widest array of component types and implementations available to compose a solution. Testers and certification engineers, conversely, will want to limit the number of possible application parts to reduce testing and verification complexity.

Even after a configuration is found that satisfies the numerous/-competing concerns of the roles, implementing the configuration can be tedious and error-prone. In particular, multiple roles must coordinate and correctly edit configuration scripts required to assemble the application. Component developers instruct component assemblers on the port functions and requirements. Component assemblers wire the components together and dictate the CPU and memory requirements to application deployers (the role responsible for placing components on nodes). Deployers obtain the correct binaries from application packagers and place them onto the appropriate nodes. Miscommunication between roles, subtle mistakes in configuration scripts, and other hard-to-diagnose errors can allow configuration errors to creep into applications and are thus a major contributor to application failure (D. Oppenheimer [2003]).

This paper extends our previous work White et al. [2007a] on simplifying the configuration of enterprise Java applications. We include new contributions that show how our original Java-baed approach can be generalized to other types of component-based systems. In particular, the paper shows the complexity of configuring DRE component-based systems through a Lightweight CCM avionics application. We demonstrate how the same challenges that plague enterprise Java configuration extend into DRE component-based systems (and are possibly even more challenging). Moreover, the paper presents results showing that the same reductions in manual configuration effort we achieved applying Fresh to enterprise Java can be obtained by applying Fresh to Lightweight CCM.

At the heart of our approach is a model-driven engineering (MDE) tool called *Fresh* that is designed to reduce the complexity of deriving a correct application configuration and implementing the configuration in configuration scripts. Fresh simplifies and improves the correctness of configuring DRE component-based applications by:

(1) Capturing configuration rules through feature models, which describe application variability in terms of differences in functionality.
(2) Translating an application's feature models into a constraint satisfaction problem (CSP) and using a constraint solver to automatically derive a correct application configuration for a requirements set,
(3) Facilitating configuration optimization for a requirements set by providing a configurable cost function to the constraint solver to select optimal configurations, and
(4) Providing an XML configuration file annotation language that allows it to inject configuration decisions into configuration scripts directly and reduce configuration implementation errors.

Fresh uses feature models (Kang et al. [1990]) to describe the rules for configuring an application. Feature modeling can be used to describe an application's configuration rules in terms of variations in functionality. For example, an avionics mission computing application that could be built using different satellite positioning systems could be described by feature models in terms of its:

(1) Variations in functional capabilities (*e.g.*, GPS vs. Galileo satellite positioning sensors),
(2) Variations in non-functional properties (*e.g.*, processor power consumption, weight, etc.), and
(3) Constraints between features (*e.g.*, ARM binaries for the Galileo positioning sensor require an ARM processor)

Feature modeling provides an intuitive model for describing application variability and has been applied to a number of domains ranging from automobiles Nechypurenko et al. [2007] to applications for mobile phones White et al. [2007b]. Deriving a valid configuration from a feature model involves:

(1) Selecting required features (*e.g.*, Galileo),
(2) Selecting features corresponding to the capabilities of the target platform (*e.g.*, ARM), and
(3) Deriving any remaining features needed to create a complete and valid configuration (*e.g.*, ARM Galileo binaries)

The remainder of this paper is organized as follows: Section 2 presents a component-based avionics application, called BasicSP, which we use as a motivating example of a DRE system throughout the paper; Section 3 describes the challenges of configuring the BasicSP application; Section 4 introduces Fresh and describes how it addresses the challenges of configuring component-based applications in DRE systems; Section 5 presents empirical results of applying Fresh to the configuration of BasicSP and shows that Fresh produces ∼80-90% decrease in configuration effort; and Section 6 presents concluding remarks.

2. AVIONICS APPLICATION EXAMPLE OF A DRE SYSTEM

As a representative example of a component-based DRE system, we use the BasicSP scenario, which is based on the Boeing Bold Stroke avionics mission computing platform (Sharp and Roll [2003]) shown in Figure 1. The BasicSP application includes several Lightweight CCM components. One component is an avionics navigational display that receives updated airframe position coordinates from a positioning sensor. The rate generator component sends out a periodic pulse that causes the positioning sensor to update its current coordinates. Once the coordinates are updated, the positioning sensor sends a ready signal to the display component to update its coordinates.

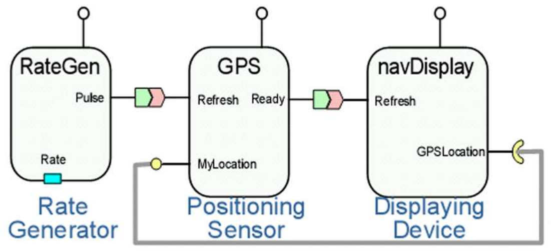


Fig. 1. Architecture of the BasicSP Avionics Example

Lightweight CCM supports the deployment and configuration of components based on XML configuration files. An emerging trend in the development of avionics systems is to use component-based middleware along with a product-line architecture (PLA) (Clements and Northrop [2002]). A PLA consists of a group of core assets, such as reusable software components and test cases, and a set of rules for composing the assets into a product variant. When an application for a new set of requirements is needed, an application variant is configured from the reusable assets to meet the new requirement set. A PLA helps reduce development costs by reusing existing core assets and codifying the process of correctly configuring assets into an application variant.

The BasicSP product-line. To demonstrate the complexity of declaratively configuring a set of assets into a variant, we created a product-line from the BasicSP example. The modified BasicSP example includes multiple satellite-based positioning systems that can be leveraged as the positioning sensor to provide the coordinates of the airframe. Moreover, the product-line includes different variations in the processors that can be leveraged to run the rate generator, positioning sensor, and display.

Configuring a variant from the BasicSP product-line involves several participants divided into different roles (Wang et al.

[2000b]). For example, component developers are responsible for producing software components, application assemblers composes software components into applications, application deployers determine which processing units host which components, and infrastructure developers determine what processing units are available in the airframe. Each role has its own viewpoint and concerns regarding the properties of the configuration. For example, component developers are focused on the functional aspects of the components and their real-time scheduling, whereas infrastructure developers are geared towards the weight, power consumption, and cost of the available processing units.

A valid BasicSP variant must integrate the concerns of each viewpoint into a functioning application. To codify the rules for configuring a proper variant, we produced feature models that relate how the different points of application variability (such as the number and types of processing units) affect each other (*e.g.*, the available processing power will restrict the components that can be used). Feature modeling describes an application's points of variability in terms of variations in functional and non-functional capabilities. Moreover, feature modeling provides a method of codifying the rules that restrict how selecting one feature affects how other features can be selected.

An overview of the BasicSP feature modeling notation. Figure 2 shows the feature model for BasicSP. BasicSP requires the *Rate Gen*, *Position Sensor*, and *Display* features, which is denoted by the filled oval above each of these features. Moreover, BasicSP requires one to three processors, which is denoted by the "[1..3]" cardinality label applied to the Processor feature. Figure 3 contains additional feature modeling notations. The

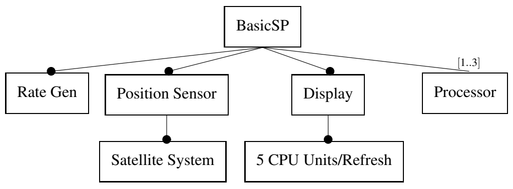


Fig. 2. Feature Model of BasicSP

Rate feature requires exactly one (an XOR relationship) of the features *20hz*, *25hz*, and *30hz*. Finally, Figure 6 contains the

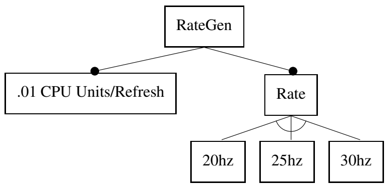


Fig. 3. Feature Model of the RateGen

notation for optional features. The *x86* feature can (but is not required to) include the *GPS* feature, which is denoted by the unfilled oval.

3. CHALLENGES OF CONFIGURING COMPONENT-BASED APPLICATIONS FOR DRE SYSTEMS

This section outlines the key challenges of configuring a component-based application (such as BasicSP) for DRE systems (such as avionics mission computing). In general, it is

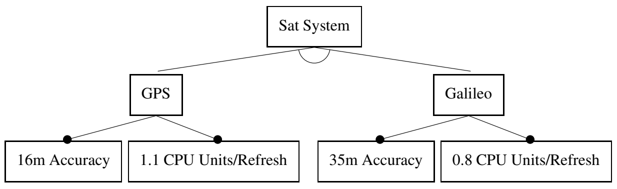


Fig. 4. Feature Model of the Available Satellite Systems

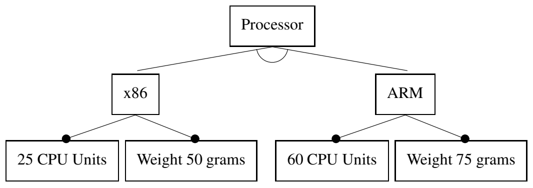


Fig. 5. Feature Model of the Processor Options for BasicSP

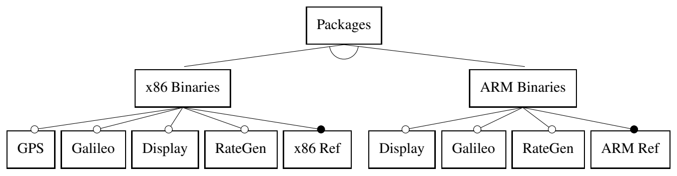


Fig. 6. Feature Model of the Packaging Options for BasicSP

hard to configure component-based applications for DRE systems due to the numerous competing concerns, such as balancing processor power consumption against required processing power. This problem is exacerbated by the multiple roles and viewpoints in the configuration process.

3.1 Challenge 1: Configuration Complexity

Each configuration choice in a component-based application may affect numerous other decisions that can be made by other roles. In many cases, no formal documentation of these cause/effect relationships exists. Even when semi-formal documentation, (*e.g.*, feature models) exists, the large number of components, numerous cause/effect relationships, and complex global constraints (*e.g.*, limitations on available memory), make it hard to derive a valid configuration manually.

In the BasicSP application, for example, selecting the GPS component has numerous side effects on further configuration decisions. The total number of CPU Units consumed per second cannot exceed the rated CPU Units per second of the processors. If the GPS component is selected along with a RateGen at 25hz, the GPS component will consume 27.5 CPU Units on its host. This combination of a GPS at 25hz precludes using the x86 based processor.

The problem with the feature combination outlined above, however, is that there are no binaries to run the GPS component on the ARM processor. Although the configuration appears correct, a subtle combination of a resource constraint and a packaging limitation (that may not be realized until deployment time) makes the combination invalid. These long chains of cause/effect relationships are hard to predict and handle manually.

3.2 Challenge 2: Incorrect configuration implementation

Configuring a component-based application involves correctly editing numerous configuration files (*e.g.*, CCM XML deployment descriptors), preparing the target infrastructure (*e.g.*, installing required libraries and starting supporting processes), and installing the application's own binaries on its target hosts. These configuration tasks are spread across multiple roles participating in the application's configuration. For example, the application deployer will install the application's binaries on the correct hosts and the application assembler will create the XML configuration files specifying how to connect components together.

The BasicSP example uses multiple XML deployment descriptors, which provide standardized Lightweight CCM mechanisms to specify configuration directives. Numerous changes must be made to BasicSP's XML deployment descriptor, however, to change the satellite system used as a position sensor. First, the specification of the component used to implement the position sensor must be changed (performed by component assemblers). The new implementation specification of the position sensor must also include the ids of its associated implementation artifacts (*e.g.*, dynamic link libraries). The ids for these artifacts are produced by component packagers. If the new position sensor uses a different interface than the previous position sensor, the component assembler must also update the wiring of the components by changing the ports and facets involved in the position sensor's refresh signal, the display's coordinates input, and the display's refresh signal.

The numerous configuration activities that must be coordinated across the various participating roles makes manual configuration of a component-based application tedious and error-prone. Simple mistakes, such as packaging the application with binaries for the wrong processor architecture, can cause the application to crash at launch. More subtle mistakes, such as accidentally using the identifier for the 30hz RateGen instead of the 20hz RateGen, will produce an application that launches correctly but fails under load. Figure 7 shows the multiple dependencies between roles responsible for configuring BasicSP. As shown in this figure, coordinating multiple roles and execut-

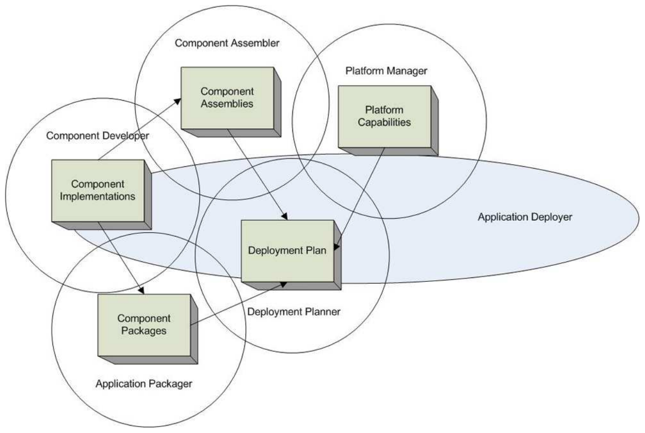


Fig. 7. Configuration Dependencies between Roles for BasicSP

ing a complex configuration is tricky.

4. SOLUTION APPROACH: AN AUTOMATED CONFIGURATION ENGINE FOR LIGHTWEIGHT CCM APPLICATIONS

This section describes the *Fresh* configuration engine and how it addresses the challenges of configuring component-based applications for DRE systems described in Section 3.

4.1 Capturing Configuration Rules in Feature Models

One of the key steps towards correctly configuring a component-based application is to capture the rules for configuring the application. Fresh uses feature models (Kang et al. [1990]) to describe the rules for configuring an application.

Fresh's feature modeling language is implemented as both a textual Domain-Specific Language (DSL) and a graphical modeling tool in Eclipse. The graphical modeling tool is based on top of the Generic Eclipse Modeling System (GEMS) (Nechypurenko et al. [2007]), which is an MDE tool for rapidly creating diagram-based modeling tools from a metamodel.

4.2 Automating Configuration Derivation

In addition to providing an intuitive interface for documenting configuration rules, previous research (Benavides et al. [2005]) has demonstrated reductions from feature models to constraint satisfaction problems (CSPs). Once a CSP formulation of a feature model has been obtained, a constraint solver can be used to derive a correct application configuration. Using a constraint solver to derive an application solver addresses Challenge 1 from Section 3 by eliminating manual derivation. Moreover, using a constraint solver to derive an application configuration has the following benefits over a manual configuration process:

- The correctness of derived configurations is guaranteed with respect to application constraints,
- The solver can identify if no valid solution exists that meets the requirements,
- A cost function can be used to select a configuration that optimizes key properties of the solution,
- No manual effort is required to reconcile the complex cause/effect relationships described in Section 3.1, and
- The solver can find a solution that reconciles opposing viewpoints and concerns involved in configuration (if such a solution exists).

A missing element of existing mechanisms for translating feature models into CSPs and satisfiability problems (Mannion [2002]), is that these approaches do not take into account resource constraints, which are important in DRE systems. In previous work (White et al. [2007a]), we have extended the work in (Benavides et al. [2005]) to incorporate resource constraints and show that it is feasible to consider them for certain size problems. The exact upper bound on a feasible resource problem varies from problem instance to problem instance but is typically not a limitation of automated configuration from CSPs.

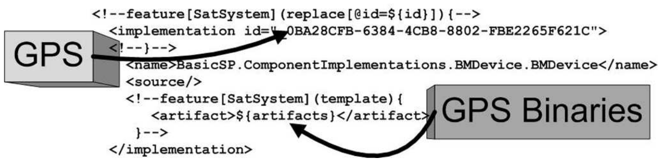
4.3 Configuration Injection

Along with the difficulty of deriving a valid configuration, Section 3 described the complex coordination needed to implement a valid configuration in an application's configuration scripts. To help decrease the complexity of implementing a configuration, Fresh includes an XML configuration file annotation language that can be used to inject a derived configuration directly into an application's configuration files.

Fresh's configuration annotation language includes a number of annotations that can be used to match an XML configuration file to a derived solution, including mechanisms for:

(1) Inserting different attribute values based on the selected feature set,
(2) Removing configuration sections,
(3) Conditionally inserting configuration sections based on the selection of specific feature combination, and
(4) Performing template-based duplication of configuration directives for specific feature types.

Fresh's annotation language is based on XML comments and does not change the structure or semantics of the original configuration language, as can be seen in Figure 8. If the application must be configured without Fresh in certain circumstances, therefore, the Fresh annotations need not be removed to configure the application normally. By automatically injecting configuration decisions directly into XML configuration scripts, Fresh significantly reduces manual configuration effort, and configuration errors, as shown in Section 5.



```
<!--feature[SatSystem](replace[@id=${id}]){-->
  <implementation id="_0BA28CFB-6384-4CB8-8802-FBE2265F621C">
  <!--}-->
    <name>BasicSP.ComponentImplementations.BMDevice.BMDevice</name>
    <source/>
    <!--feature[SatSystem](template){
       <artifact>${artifacts}</artifact>
     }-->
  </implementation>
```

Fig. 8. Fresh XML Annotations

A final benefit of directly injecting configuration decisions into application configuration files is that the bindings for each configuration decision can be unit tested. For example, a unit test can be built to ensure that when the GPS component in BasicSP is selected, the correct XML configuration directives in the component deployment descriptor are produced. After validating the injection of each feature into the configuration files, application developers can be certain that future configurations involving the tested features will be implemented correctly.

With a manual configuration process, conversely, each time a new configuration is produced the configuration files must be checked to ensure that no mistakes are made. In some cases, an application may be delivered to customer who are responsible for properly implementing a configuration, which they may not do correctly. Using Fresh's automated approach, in contrast, enables customers that receive an application to ensure it is configured correctly to meet its requirements.

5. EMPIRICAL RESULTS

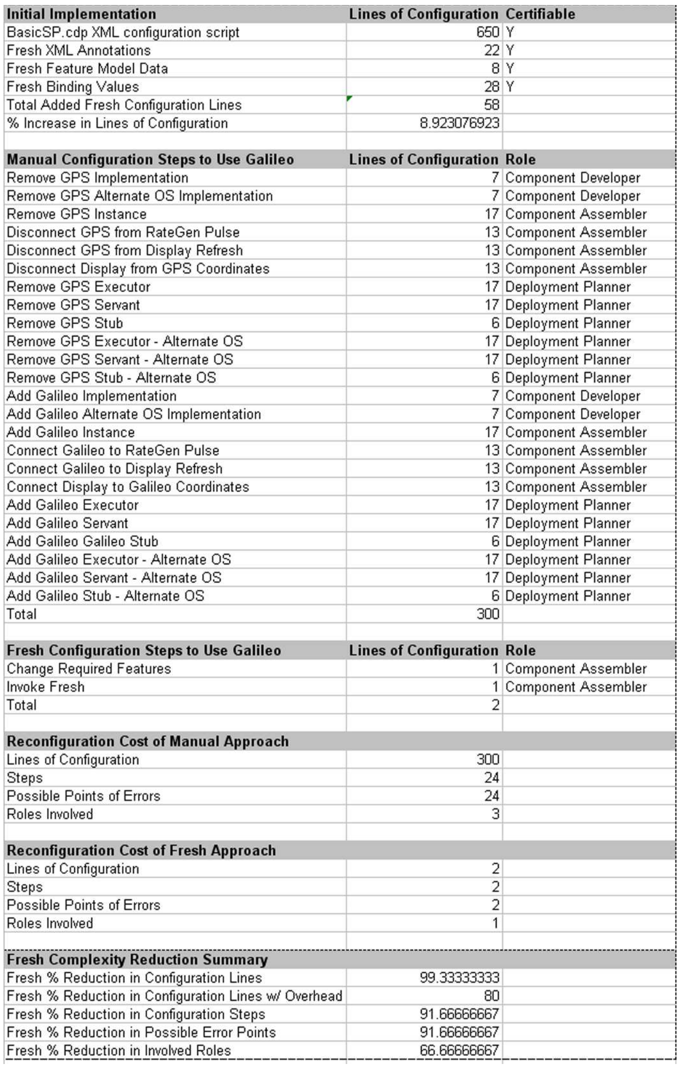
To demonstrate the reduction in manual configuration complexity provided by Fresh, this section evaluates a scenario in which the BasicSP example from Section 2 has the position sensor changed from GPS to Galileo. In this scenario, BasicSP has a base deployment descriptor (the out-of-the-box descriptor included with the CIAO Lightweight CCM container implementation) that must be modified to:

(1) Add the required implementation of Galileo,
(2) Create an instance of the Galileo component,
(3) Connect the Galileo component to the RateGen and Display, and
(4) Add Galileo to the deployment plan by specifying its servant, executor, and stub along with their associated implementation artifacts.

The Galileo and GPS position sensors possess the same basic functionality but name their ports/facets slightly differently. Thus, although the two can be swapped, their connections and various deployment descriptor configuration lines must also be swapped. We evaluate the reduction in manual configuration complexity in terms of the total lines of configuration directives, total steps, possible points of where mistakes can be made, and total roles that must be coordinated to acheive the swap. Although we assert that using a constraint solver to derive configurations adds a mental complexity reduction, this cannot be quantified readily and is thus not included in our results.

A key characteristic that we evaluate is the number of possible steps at which a configuration error can occur. With a manual approach, each time a new configuration is produced, it must be tested to ensure that the configuration file producer has not made any errors, which adds significant overhead. With the Fresh approach, conversely, the injection of each feature into the configuration file can be unit tested. Once it is certified that Fresh correctly injects each feature into the configuration files, therefore, Fresh is guaranteed to produce a correct configuration.

As seen in the inital implementation section of Figure 9, the base configuration file for BasicSP contains 650 lines of configuration directives. Adding Fresh XML annotation directives,



Initial Implementation	Lines of Configuration	Certifiable
BasicSP.cdp XML configuration script	650	Y
Fresh XML Annotations	22	Y
Fresh Feature Model Data	8	Y
Fresh Binding Values	28	Y
Total Added Fresh Configuration Lines	58	
% Increase in Lines of Configuration	8.923076923	
Manual Configuration Steps to Use Galileo	**Lines of Configuration**	**Role**
Remove GPS Implementation	7	Component Developer
Remove GPS Alternate OS Implementation	7	Component Developer
Remove GPS Instance	17	Component Assembler
Disconnect GPS from RateGen Pulse	13	Component Assembler
Disconnect GPS from Display Refresh	13	Component Assembler
Disconnect Display from GPS Coordinates	13	Component Assembler
Remove GPS Executor	17	Deployment Planner
Remove GPS Servant	17	Deployment Planner
Remove GPS Stub	6	Deployment Planner
Remove GPS Executor - Alternate OS	17	Deployment Planner
Remove GPS Servant - Alternate OS	17	Deployment Planner
Remove GPS Stub - Alternate OS	6	Deployment Planner
Add Galileo Implementation	7	Component Developer
Add Galileo Alternate OS Implementation	7	Component Developer
Add Galileo Instance	17	Component Assembler
Connect Galileo to RateGen Pulse	13	Component Assembler
Connect Galileo to Display Refresh	13	Component Assembler
Connect Display to Galileo Coordinates	13	Component Assembler
Add Galileo Executor	17	Deployment Planner
Add Galileo Servant	17	Deployment Planner
Add Galileo Galileo Stub	6	Deployment Planner
Add Galileo Executor - Alternate OS	17	Deployment Planner
Add Galileo Servant - Alternate OS	17	Deployment Planner
Add Galileo Stub - Alternate OS	6	Deployment Planner
Total	300	
Fresh Configuration Steps to Use Galileo	**Lines of Configuration**	**Role**
Change Required Features	1	Component Assembler
Invoke Fresh	1	Component Assembler
Total	2	
Reconfiguration Cost of Manual Approach		
Lines of Configuration	300	
Steps	24	
Possible Points of Errors	24	
Roles Involved	3	
Reconfiguration Cost of Fresh Approach		
Lines of Configuration	2	
Steps	2	
Possible Points of Errors	2	
Roles Involved	1	
Fresh Complexity Reduction Summary		
Fresh % Reduction in Configuration Lines	99.33333333	
Fresh % Reduction in Configuration Lines w/ Overhead	80	
Fresh % Reduction in Configuration Steps	91.66666667	
Fresh % Reduction in Possible Error Points	91.66666667	
Fresh % Reduction in Involved Roles	66.66666667	

Fig. 9. Results of Configuring BasicSP with Fresh vs. a Manual Approach

building a simple feature model of BasicSP, and creating values to be injected into the configuration file by Fresh adds a total of 58 configuration directives. Fresh thus adds ∼8% to the total lines of configuration directives required for BasicSP.

Modifying the BasicSP configuration file to use Galileo requires removing the old GPS implementation, connections, etc. As seen in the "Manual Configuration Steps to Use Galileo" section in Figure 9, a significant number of steps and lines of configuration directives are involved. At each step in the process, the role modifying the configuration directives can make mistakes and introduce errors.

The "Fresh Configuration Steps to Use Galileo" section in Figure 9 shows the total lines of configuration directives to reconfigure the BasicSP configuration file with Fresh. Fresh requires the addition of one configuration directive to enable the Galileo feature and the execution of Fresh from the command line to regenerate the BasicSP deployment descriptor.

The "Fresh Complexity Reduction Summary" section in Figure 9, compares the total manual configuration effort of the manual approach versus the Fresh approach. If the initial overhead of setting up Fresh is included in the calculations, Fresh yields an 80% reduction in the total lines of configuration directives. If the intial overhead is not considered (for cases where the application is configured by a customer), Fresh creates a 99.3% reduction in total lines of configuration directives.

In the manual approach, if component assemblers decide to change to the Galileo component, the component developers and deployment planners must be involved in updating the deployment descriptor. With the Fresh approach, component developers and deployment planners initially encode their expertise into the configuration file as Fresh XML annotations. Thus, each time application assemblers need to swap a component, Fresh uses the XML annotations produced by the other two roles and does not require their involvement. As can be seen in the "Fresh Complexity Reduction Summary" section in Figure 9, Fresh reduces the total roles involved in the change by two-thirds. Limiting the number of roles required to implement a change reduces the cost of coordinating the participants and the chances of miscommunication.

Finally, as shown in the "Fresh Complexity Reduction Summary" section in Figure 9, Fresh reduces the total number of configuration steps that must be performed by 91.67%. Moreover, each eliminated manual configuration step was a potential source of errors in the process, so the overall number of steps where errors can be made are also reduced by 91.67%. Although an intial cost is incurred by adding Fresh configuration directives, it allows for the configuration process to be unit-tested and certified. After the Fresh configuration process is certified correct, there is a large reduction in the potential sources of configuration errors, which are a major contributor to system downtime and failure (D. Oppenheimer [2003]).

6. CONCLUDING REMARKS

Component-based DRE systems achieve a high-level of software reuse and flexibility by assembling applications dynamically at deployment time via configuration scripts rather than statically at compilation time. Configuring a component-based applications, however, involves integrating the opposing concerns and requirements of numerous unique viewpoints, such as component developers and application deployers. The large number of competing concerns, conflicted roles, and configuration steps required to configure component-based applications makes it hard to find and implement correct configurations.

This paper describes how Fresh leverages feature modeling to capture the configuration rules of a component-based application. Fresh transforms the application's feature models into a CSP that allows a constraint solver to be leveraged to derive a correct (and possibly optimal) configuration. Fresh also includes an XML annotation language that allows Fresh to inject a derived configuration directly into an application's configuration scripts and eliminate manual configuration implementation steps.

The results of an experiment based on the Boeing BasicSP avionics mission computing scenario shows that Fresh can reduce the total configuration cost (in terms of lines of configuration directives) by ∼80-90%. Moreover, Fresh ensures that derived configurations are correct with respect to the application's configuration constraints and produces a 91% drop in the number of possible points in the configuration process were errors can be introduced. Finally, Fresh helps to further decrease configuration effort by eliminating the involvement of two of the original three roles (*i.e.*, deployment planner and component developer) required to update avionics application deployment descriptors.

REFERENCES

BEA Systems, et al. *CORBA Component Model Joint Revised Submission*. Object Management Group, OMG Document orbos/99-07-01 edition, July 1999.

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.

Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

D. Patterson 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.

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.

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.

Andrey Nechypurenko, Egon Wuchner, Jules White, and Douglas C. Schmidt. Application of Aspect-based Modeling and Weaving for Complexity Reduction in the Development of Automotive Distributed Realtime Embedded System. In *Proceedings of the Sixth International Conference on Aspect-Oriented Software Development*, Vancouver, British Columbia, March 2007.

David C. Sharp and Wendy C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. In *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

Nanbor Wang, Douglas C. Schmidt, and David Levine. Optimizing the CORBA Component Model for High-performance and Real-time Applications. In *'Work-in-Progress' session at the Middleware 2000 Conference*. ACM/IFIP, April 2000a.

Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. An Overview of the CORBA Component Model. In George Heineman and Bill Councill, editors, *Component-Based Software Engineering*. Addison-Wesley, Reading, Massachusetts, 2000b.

Jules White, Krzysztof Czarnecki, Douglas C. Schmidt, Gunther Lenz, Christoph Wienands, Egon Wuchner, and Ludger Fiege. Automated model-based configuration of enterprise java applications. In *EDOC 2007*, 2007a.

Jules White, Andrey Nechypurenko, Egon Wuchner, and Douglas C. Schmidt. Optimizing and Automating Product-Line Variant Selection for Mobile Devices. In *11th International Software Product Line Conference*, September 2007b.