Role-based Object Constraint Programming

Jules White[1], Douglas C. Schmidt[1], Andrey Nechypurenko[2], and Egon Wuchner[2]

[1] Vanderbilt University,
Department of Electrical Engineering and Computer Science
{jules, schmidt}@dre.vanderbilt.edu
[2] Siemens AG,
Corporate Technology (SE 2)
{andrey.nechypurenko, egon.wuchner}@siemens.com

Abstract. Feature models help to simplify the development of product-lines for component-based systems by capturing the key commonalities and variabilities of an application and how they relate to component configuration and composition. Performing requirements-driven feature selection from a feature model, such as selecting a maximal set of features whose constituent components fit within the target infrastructure's memory, motivates the use of a constraint solver. Building a constraint solver-based feature selection engine is hard, however, since solvers typically have low-level APIs and require complex transformations on data encapsulated in multiple features. To address these challenges, we developed an approach called Role-based Object Constraints (ROCs).
This paper provides the following contributions to the study of building reusable feature selection engines for component-based applications: (1) it presents the techniques that can be used to decouple applications and constraint solvers to make solver-based solutions reusable, (2) it shows how to select and coordinate multiple constraint solvers dynamically to increase performance, and (3) it presents results from applying ROCs to a case study that selects software features for mobile devices and shows how ROCs increases modularity, reusability, and performance compared with a tightly-coupled single-solver solution.

1 Introduction

Product-line architectures (PLAs) enable the development of families of software packages that can be retargeted for different requirement sets by leveraging common capabilities, patterns, and architectural styles [1, 2]. The design of a PLA is typically guided by scope, commonality, and variability (SCV) analysis [3]. SCV captures key characteristics of software product-lines, including their (1) scope, which defines the domains and context of the PLA, (2) commonalities, which describe the attributes that recur across all members of the family of products, and (3) variabilities, which describe the attributes unique to the different members of the family of products.

One approach to describing the commonality and variability in a product line is *feature modeling* [4, 5]. A feature model describes the indivisible units of functionality that can be enabled or disabled in an application. From a PLA perspective, features are the high-level capabilities of the application that manifest from different configurations of the underlying implementation components. For example, selecting a MessageDriven Enterprise Java Bean (EJB) and a `MessageBasedAccount` EJB could result in the higher-level feature *AsynchronousMessagingCapable*. The features chosen for the product-line variant drive the configuration and composition of the underlying implementation components.

Although feature models simplify the development of new applications, it is hard to find a valid feature set from a product-line that meets a set of requirement goals, which we term *requirements-driven feature selection*. For example, if a developer wishes to choose a maximal set of features that fit within the resource constraints (such as available memory and CPU cycles) on the target infrastructure there may be an enormous number of possible feature sets to try. In particular, if the solution space for a feature set composed of N unique features is viewed as a binary string where each position represents whether or not the ith feature is present in the set, there are 2^N distinct values that the feature set can have.

In production scenarios [6], there may be not only numerous components but numerous types of requirements governing feature selection. These requirements can range from resource restrictions and OS types to feature inter-dependencies and composition rules. Manual approaches to perform requirements-driven feature selection do not scale well with these large solution spaces and complex constraints.

A constraint solver is a more scalable approach to select requirements-driven features based on complex resource, composition and other constraints. A considerable amount of work [7–10] has been done on constraint optimization. As a result, this technique has been applied successfully to a large number of domains ranging from scheduling and configuration to resource management and route planning. Expert systems are used in medical domains for diagnosis assistance. Logistical operations, such as UPS, Fedex, and DHL, make extensive use of constraint solvers to plan, route, and schedule shipments.

Tools for solving constraint problems, however, are often stove-piped solutions that are tightly coupled to each application and constraint solver implementation and can thus rarely be reused across feature models. Moreover, constraint solvers tend to have low-level APIs that are either not object-oriented (OO) or that provide OO abstractions that are not aligned with the feature selection domain. Considerable work must done to translate each high-level requirement into a constraint satisfaction problem, such as an instance of a 0-1 programming problem.

Another challenge with constraint solving tools is that data must be extracted from the feature model and the requirement goals and transformed to the native constraint solver formats. This transformation from (1) requirement definition to (2) constraint satisfaction problem, data extraction, and API leveraging is

typically duplicated for each constraint solver type needed to handle the various high-level requirement types. Since solutions are often tightly-coupled and hard to reuse, however, this endeavor is not only expensive but the costs cannot be amortized across feature models.

While developing feature selection engines that leverage constraint-solvers [11], we have wrestled with these problems of constraint-solver integration. To address these challenges, we have developed an approach called *Role-based Object Constraints* (ROCs) programming that helps simplify the development of feature selection engines that leverage constraint solvers. This paper shows how ROCs allows developers to: (1) leverage a high-level of abstraction to specify complex feature requirements using types aligned with the feature rather than the constraint solver domain, (2) allow developers to build reusable feature selection engines that can be retargeted for different feature models, (3) leverage a pluggable architecture to automate data transformation from arbitrary feature models to native constraint solver formats, (4) to specify high-level requirement to constraint satisfaction problem mappings that can be reused across feature models and constraint solver implementations, and (5) dynamically select and coordinate multiple constraint solvers based on requirement types and problem instance characteristics.

The remainder of this paper is organized as follows: Section 2 presents a case study for automated feature selection in mobile device product-lines; Section 3 introduces ROCs programming and shows how it uses key patterns and abstractions to address the complexities of building feature selection engines based on constraint solvers; Section 4 analyzes the results of experiments that apply ROCs to our case study; Section 5 compares our work on ROCs programming with related work; and Section 6 presents concluding remarks.

2 Motivating Example: FeatureBind

As part of the ROCs project, we developed an application called *Scatter* for modeling product-lines for mobile device software. *Scatter* provides developers with the capability to capture the structure of a PLA, the non-functional requirements of the components, and to use constraint solvers to derive an optimal variant from the product-line with respect to a cost function. *Scatter* allows on-demand software deployment of customized variants targeted specifically for an individual mobile device. After developing Scatter, we took its core constraint solving engine and built a new application, called *FeatureBind*, that performs dynamic software configuration for mobile devices based on feature models.

For example, if a customer carries a mobile device onto a train and requests that a food services application be provided to the device, *FeatureBind* analyzes the various feature composition and non-function requirements and determines which features to enable in the application delivered to the customer. If the customer's device is a laptop, resources constraints may not be an issue when selecting features. If the device is a Treo phone, however, optimizing the features to minimize memory and CPU usage may be much more important.

FeatureBind's feature selection engine may also have to make decisions based on business data. For example, two different menus may be available for first class and coach class passengers. If the passenger is seated in first class, the feature selection engine must ensure that first class features, such as the enhanced menu, are active in the application deployed to the device.

Conventional ways of identifying valid feature sets [5] involve software developers manually determining the software features that must be active, the components to configure, and how to compose and deploy the components. In addition to being infeasible in an on-demand setting, such as our mobile software deployment environment, where the characteristics of the target devices are not known *a priori*, such manual approaches are tedious, error-prone, and a significant source of system downtime [12].

As a case study for this paper, we present the work we did refactoring *FeatureBind* to use our ROCs infrastructure. When an application variant is requested, *FeatureBind* analyzes the target device's infrastructure that the application will be running on, including its resources (such as memory) and its configuration (such as OS and middleware stacks) and determines the appropriate set of features to be used by the application. This section shows how *FeatureBind* transforms high-level resource requirements into a format that can be operated upon by a constraint solver. The remainder of the paper then shows how ROCs helps to address the challenging transformation and problem specification steps involved in *FeatureBind*.

FeatureBind transforms the composition, configuration, and resource requirements of the application's features into a constraint satisfaction problem and solves for valid feature combinations that can be run on the target infrastructure. This design allows *FeatureBind* to ensure that both a compositionally correct set of features are chosen for the application and that the underlying components fit within the available memory and CPU provided by the hardware/software infrastructure.

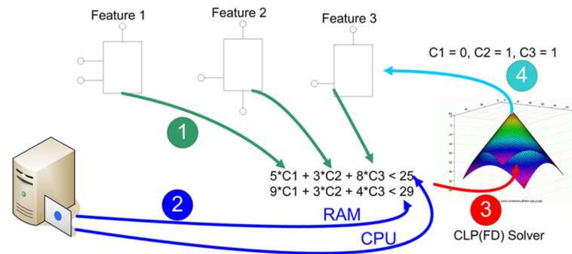

Fig. 1. Mapping Feature Selection to a Constraint Satisfaction Problem for CLP(FD)

For *FeatureBind* to select features that adhere to the resource constraints of the target infrastructure, it employs a *Constraint Logic Programming Finite Domain* (CLP(FD)) [8] solver, as shown in Figure 1. *FeatureBind* transforms

the resource requirements into a bin-packing problem that is operated on by the CLP(FD) solver. For each *Feature* C_i available in the model, a presence variable DC_i, with domain [0,1] is created to indicate whether or not the *Feature* is present in the chosen feature set. For every modeled resource type, such as CPU, the individual *Feature* resource requirements, $C_i(R)$, when multiplied by their presence variables and summed cannot exceed the available amount of that resource, $Dvc(R)$, on the target *Infrastructure*.

If the presence variable is assigned 0 (which indicates that the feature is not in the variant) the resource demand by that feature falls to zero. The $\sum C_i(R) * DC_i < Dvc(R)$ constraint is created to enforce this rule. The solver supports multiple types of composition relationships between *Features*. For each *Feature* C_j that C_i depends on, *FeatureBind* creates the constraint $C_i > 0 \rightarrow C_j = 1$. *FeatureBind* also supports a selection composition constraint that allows exactly N components from the dependencies to be present. The selection operator creates the constraint $C_i > 0 \rightarrow \sum C_j = N$.

When new features are added to the application, the linear equations must be modified to incorporate new DC_i presence variables. As features are added to the application, developers must also capture the values of the resource demands of the various features and map them to new $C_i(R)$ coefficients. At runtime, the application must collect the status of the various types of resources available on the target infrastructure and translate the statuses into values for the right-hand sides of the equations $Dvc(R)$.

It is clearly tedious and error-prone to manually create a mapping from an application's resource requirements and the resources available on the target infrastructure to a system of linear equations. As shown in Section 3, however, leveraging a constraint solver in this fashion also presents numerous challenges to a developer. What is needed, therefore, is a mechanism for raising the level of abstraction for requirements specification, automating data and requirements transformation to and from the constraint solver, and decoupling solvers, constraint satisfaction problems, and application components to make solutions reusable, as described in Section 3.

3 Addressing the Complexity of Leveraging Constraint Solvers with ROCs Programming

To address the challenges of building constraint-solver based feature selection engines, we developed an approach called *Role-based Object Constraint* (ROCs) programming that has the following benefits:

- Subject-oriented design [13] analysis techniques are used to raise the level of abstraction of requirement specification and align it with the feature modeling domain.
- The transformation from requirement to constraint satisfaction problem is decoupled from the native formats of constraint solvers and constraint satisfaction problems can be expressed in terms of features and their interrelationships.

- Complex requirements can be specified as workflows across multiple constraint satisfaction problems.
- The transformation of data from feature models to and from the solver is automated by the underlying infrastructure by leveraging a common abstraction.
- Solvers are decoupled from constraint satisfaction problem specifications allowing solvers to be plugged-in depending on the solving QoS characteristics, such as optimality and speed, needed by the application.

This section describes the challenges we faced when developing a feature selection engine that leverages a constraint solver to make requirements-driven selection decisions. We examine each challenge in the context of our example case study presented in Section 2 and show how it is addressed by ROCs.

3.1 Challenge 1: Mapping Constraints to Application Components

Context. To express requirements to the constraint solver, constraints must be mapped to the various components in the application. Requirements over a high-level concept, such as a feature, may be mapped to multiple actual application components in the application.

Problem. Requirements are typically based on high-level concepts, such as "if the *AsynchronousMessaging* feature is present, it requires that an *AsynchronousMessageService* be installed." If *AsychronousMessaging* is present in a J2EE context it could mean that an Enterprise Java Bean was contained in the application that inherited from *MessageDrivenBean*. In turn, deciding to install *AsynchronousMessageService* might require analyzing the J2EE application server that the application was running in to ensure that the Java Messaging Service (JMS) was installed and running.

For each new high-level concept that a requirement depends on, a mapping must be created down to the actual implementation details to either capture the data required to analyze the state of the concept, such as how much memory is in the target system, or to enact the concept, such as make a feature active. Requirements typically cover numerous concepts that map to a large number of member variables and methods spread throughout the application. To allow constraint solving to take place, these complex mappings from high-level concepts to implementation details must be used to collect all the data represented by the concepts from the numerous components spread throughout the application.

Moreover, once constraint solving is finished, the high-level concepts must be mapped back to the original implementation details to enact the feature selections from the solver. For example, if the *AsynchronousMessaging* feature maps to the presence variable DC_1, then $DC_1 = 1$, could indicate that several *MessageDrivenBeans* should to be loaded into the application container. For each seemingly simple assignment of a variable DC_i, numerous complex changes of the underlying application may need to be performed.

Writing the complex code to traverse the application components and its environment and collect the required data is challenging. Moreover, after constraint solving is finished, more complex traversals must be implemented to walk

the object graph underlying the components and enact the appropriate implementation changes to enable the appropriate features.

Solution → Subject-oriented design classes. To address these mapping challenges, ROCs uses subject-oriented design [13] as the abstraction for viewing an application. Subject-oriented design provides a decomposition abstraction based on *design subjects*, which are units that encapsulate a single coherent piece of functionality. The design subject abstraction helps align system design and requirements. In ROCs, the components and the high-level concepts that the requirements depend on are design subjects.

For each design subject present in the requirements, a *DesignSubject* class is created. The design subject classes expose role-based relationships that they may participate in. For example, a feature may have a role-based relationship *Dependencies*, which specifies a list of other features that are required for the feature to be active. Properties of a feature can also be expressed as role-based relationships. For example, if a feature requires 10 Mbytes of RAM, the requirement could be expressed as a *RequiresRam* relationship with value 10. *Features* and high-level datatypes, such as *AsynchronousMessagingInstalled*, are mapped to *DesignSubjects* in ROCs.

A key property of the *DesignSubject* classes is that they possess the code required to call the appropriate methods and set the needed member values to map changes in the design subject's relationships to the underlying application components. For example, a *DesignSubject* class for our *AsynchronousMessaging* feature would load the appropriate *MessageDriven* beans when its *Present* relationship, modeled by the variable DC_1 was set to 1.

This use of design subjects cleanly separates the application logic and the logic required for collecting constraint solving data and implementing constraint solving results. The *design subject classes encapsulate the data and logic to make the appropriate implementation level changes to enable/disable features* in the application. Existing applications can have automated feature selection integrated into them without having to modify the existing components and their logic. Finally, the subject-oriented design provides a direct mapping between the high-level concepts of the requirements and objects in the application, thereby simplifying the mapping the output from the constraint solver to the feature model of the application.

3.2 Challenge 2: Raising the Level of Abstraction for Requirement Specification

Context. To handle non-functional composition, configuration, and deployment requirements, an application must map these requirements to concrete constraints that are understood by a constraint solver, as seen in step 1 of Figure 1. For example, an application developer must be able to specify that a feature requires a certain amount of memory when present. At binding time, the application must be able to query the constraint solver for a set of features that fits within the resources provided by the target infrastructure.

Problem. Feature models allow developers to raise the level of abstraction and express solutions using notations that are aligned with the capabilities of their application. Constraint solvers, however, either provide low-level non-OO interfaces or provide interfaces that expose concepts that are a decomposition of the solver domain not the application domain [14] and thus create a large semantic gap, as shown in Figure 2.

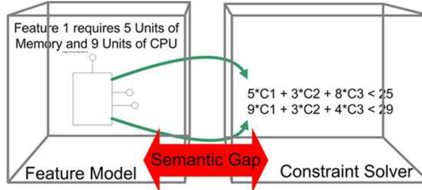

Fig. 2. High-level Requirements and Constraint Solver Semantic Gap

The semantic gap creates two key conceptual challenges that limit the ability of developers to use constraint solvers:

1. *Structural misunderstanding.* The semantic gap makes it hard for developers to relate system structures, such as features, to their representation in a constraint solver. Without a clear understanding of how solutions presented by a constraint solver relate to their features and the implementation decisions underlying the features, domain experts tend to misinterpret and incorrectly implement the recommendations from the solver.
2. *Requirements specification inconsistencies.* Without a clear understanding of how a constraint solver input maps to system structure, it is hard for developers to create and map requirements to constraints on input variables provided to the constraint solver. This misunderstanding yields situations where developers do not correctly translate the requirements into constraint satisfaction problems and cause the constraint solver to provide no results. Even worse, a result may be produced that is correct with regards to the constraint satisfaction problem, but incorrect with regards to the actual requirements, which is particularly dangerous.

Solution → High-level problem objects. ROCs provides a catalog of high-level problem data structures, that serve as the target for to developers map their requirements. This catalog can be expanded by adding new constraint solvers that publish their new capabilities as abstract problem classes.

For example, *ResourceProblem*, is an abstract problem provided in our implementation of ROCs. A high-level requirement, such as “the features do not exceed the resource constraints of the target infrastructure” can be mapped to these problem objects, such as the *ResourceProblem*. This arrangement allows developers to express their requirements in terms of high-level problems that can be much more easily related to the requirements.

Developers provide typing information to the problems by specifying the problems as assignments of the endpoints of the role-based relationships of the design subjects in their application. For example, consider a resource problem where we want to choose features for a feature set. The sum of the resource demands of the features, however, must not exceed the amount of any resource provided by the target infrastructure.

A *ResourceProblem* object can be constructed that maps the problem's *ResourceProvider* role to the *TargetInfrastructure* design subject and the *ProvidedResources* role to the infrastructure's CPU and RAM roles. *Features* can then be mapped to the *ResourceConsumer* role of the problem object and *Features*' *RequiredCPU* and *RequiredMemory* roles can be mapped to the *ConsumedResources* role of the problem. This problem tells the solver that the output should be a map that associates the *TargetInfrastructure* design subject with a set of feature design subjects.

This approach allows developers to specify problems at a very high-level of abstraction and to provide strong typing to the inputs and outputs of the problems by mapping them to specific roles on their design subjects. Moreover, the results that the constraint solver produces are assignments of the roles that were specified as input to the problem. This alleviates developers from having to write complex transformations from the design subject classes to an non-intuitive constraint solver input format, such as the various $DC_i, Dvc(R), andC_i(R)$ variables from Section 2.

Our approach also provides strong typing on the problems, which makes it hard to create problem mappings incorrectly or incorrectly pipe input or output from the solver. The results are returned as sets of design subject objects in the application domain, *i.e.*, the results are translated back into the *Feature* and *TargetInfrastructure* classes native to the feature model's domain.

3.3 Challenge 3: Providing Reusable Problem Abstractions

Context. As described in Section2, *DesignSubjects* must be transformed into inputs, such as the variables $C_i(R)$ and DC_i, that the constraint solver can understand. Once constraint solving is finished, the output provided by the constraint solver, such as the assignment of values to the DC_i variables, must be mapped back to the design subjects or features in the application.

Problem. Transformation of the data in the application's components into a format that can be used by a constraint solver, such as an instance of a *mixed integer programming* (MIP) [15] problem, requires significant development effort and is tedious and error prone. Custom transformation code must be written to convert between the two formats.

Solutions to this mapping problem tend to involve handcrafted transformation of the requirements to a constraint satisfaction problem in a specific solver's input format. This handcrafted transformation tightly couples the requirement translation to the native representation of that constraint satisfaction problem for a single constraint solver. In many cases, multiple types or implementations

of constraint solvers may be need to handle all of the requirement types of the application.

For example, a resource requirement may need a bin-packer while a configuration requirement may need a Binary Decision Diagram (BDD) [16]. With a manually produced transformation approach, the requirement specification is tightly coupled to the representation of the corresponding constraint satisfaction problem in a specific solver. Custom transformations must therefore be written for each solver that must be leveraged.

Often, individual instances of the same abstract problem may be amenable to entirely different types of constraint solver implementations. For example, in one scenario, an application may be distributed across several nodes connected through a network and the constraint solver must find a way to pack all of the features onto the multiple nodes. In this case, a bin-packing algorithm that uses a heuristic based-approach to explore the solution space and try packing configurations may be the most appropriate.

In another situation, there may be a large group of features and only a single node. The goal in this situation is to maximize a cost function when selecting the feature set for the application. In this case, a simplex method or CLP(X) approach may be the best option. The problem, however, is that although both are different flavors of bin-packing, the solver's will have different representations of the constraint satisfaction problem for bin-packing.

Separate complex transformations will therefore be needed to translate the requirements into constraint satisfaction problems. Moreover, separate transformations must be written to extract the appropriate data from the feature model. This data could include values for the coefficients $C_i(R)$ that specifically formulate the variables of each phrasing of the satisfaction problem.

Transformations must also be bi-directional. Significant effort must be expended mapping the output of the constraint solver back to the feature model and application components. The output format could range from a list of object ids to a complex tree structure representing a graph. These outputs must again be mapped to numerous design subjects in the feature model. When either the application's feature structure or the solver's input format changes, extensive work must be done to update the mapping between the two domains.

The tight coupling between the application's feature decomposition and transformation code makes constraint solver solutions rarely reusable. Each time a stove-piped tightly-coupled solution is developed for an application, it cannot be reused in other applications. Often, many common problems, such as bin-packing, occur over and over in various feature modeling domains. Tightly coupled application-solver solutions that handle these common problems require costly reinvention and rediscovery of existing solutions. Limiting reuse prevents an organization from amortizing the cost of the feature selection engine across multiple applications.

Solution → Constraint solver decoupling and transformation automation. To decouple requirement or constraint specification from the specific input format of a constraint solver, ROCs employs the Adapter pattern [17], which

adapts the interface of the problems to the native constraint satisfaction problem interface of the constraint solver. Developers express their requirements in terms of the problem data structures described in Section 3.3. Constraint solver developers provide adapters that transform the abstract problem definitions into concrete implementations of a constraint satisfaction problem in the native solver's format.

For example, the resource problem described in Section 2 could be transformed into a set of constraint network objects for a Choco [14] based bin-packer or an invocation of a Prolog-based First Fit Decreasing bin-packing [18] rule. The adapter architecture can be seen in Figure 3. This adapter-based design allows

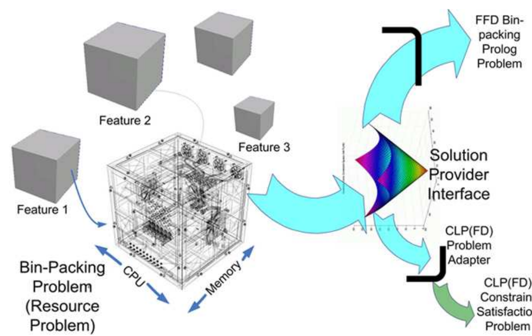

Fig. 3. Using Adapters to Map Problems to Constraint Satisfaction Problem Specifications for Solvers

problem implementations for a specific solver to be reused and also allows constraint solver implementations to be swapped in and out without affecting the application, feature model, or mapping from requirements to abstract problems. Each constraint solver only needs to define the mapping from the abstract problem types to the solver's native input format. Intelligence can also be used to select solvers based on the characteristics of particular problem instances, such as selecting a solver on the fly depending on whether or not the application is being deployed to multiple or a single node.

Once the results are obtained and mapped by ROCs to changes in the role-based relationships of the subject classes, the classes themselves take over the low-level coordination of making the appropriate method calls or other implementation changes to express the state of the high-level concepts, such as selected features. A benefit of this approach is that since the satisfaction of the high-level requirements and the mapping to and from the constraint solver's format is independent of how the high-level concepts, such as features, are mapped to the underlying infrastructure, the automated feature selection mechanism can easily be ported to different implementations of the same application.

For example, if one version of the application is based on EJB and another on CORBA, the implementations of the feature subject classes can be changed and

the high-level requirements solving reused. In the EJB version of the application, enabling the *AsynchronusMessagingFeature* may lead the *AsynchronousMessagingFeature* class to load a set of *MessageDriven* EJBs. Enabling the same feature in the CORBA version may lead the *AsynchronousMessagingFeature* class to start an instance of the OMG Data Distribution Service (DDS) [19]. This decoupling is obtained by using the Bridge pattern [17], which allows the interface exposed by the design subjects to remain constant while the implementation of the interface varies.

3.4 Challenge 4: Managing DesignSubjects and Constraint Solvers

Context. There may be hundreds or more design subject objects in an application and numerous constraint solvers. The application must have a way of managing the various subject objects and solvers, providing services for solving abstract problems, managing solver adapters, and gathering application state from the design objects.
Problem. Although design subjects provide a mechanism for extracting high-level concept state from low-level application state, an abstract problem may require data from numerous design subjects. Determining which design subjects to interrogate, traversing the numerous subject objects, and mapping their relationship values to abstract problem parameters can take considerable effort. Moreover, the application may need to track the state of the high-level concepts, such as *AsynchronousMessagingInstalled*, and as their state changes reconsider feature selection decisions. For example, if the Java Messaging Service on an application server fails, the application can change *AsynchronousMessagingInstalled* to false, rerun the feature selection engine and continue operating in the new environment. Providing this complex state tracking and feature reselection capabilities requires significant effort.

Another challenge is that several solvers may be capable of solving an abstract problem. Logic must be provided to determine which solver to use and under which conditions. Once the appropriate solver is found the correct adapter must be found and applied to map the abstract problem to the solver's native format and the result back to the application domain.
Solution → Complexity container. To integrate these design subjects, adapters, problems, and solvers, ROCs uses the concept of a *container*, which is a familiar element in component-based technologies [20]. *DesignSubjects* are added to an instance of a container and the application objects can use a *SolutionProviderFinder* service to obtain a handle to a *SolutionProvider* for a problem instance. The *SolutionProvider* interface provides a common interface to solutions instances of problems, as shown in Figure 4.

When the *SolutionProviderFinder* service is queried for a *SolutionProvider* for a particular problem instance, it evaluates the characteristics of the problem, as well as any solving policies provided with the problem, to determine which solver to return as a SolutionProvider. The solving policies of a problem allow the application to express requirements, such as optimality or max solving time that should be used to select a solver. Although the container may return different

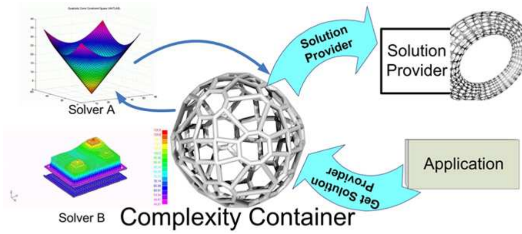

Fig. 4. The Complexity Container Architecture

solvers as the *SolutionProvider* the application component uses the solver with the same interface and is unaware of which solver was returned.

For example, a bin-packing problem that did not require optimality could be mapped to a bin-packing approximation algorithm to increase solving time. Conversely, if optimality is required the problem could be mapped to a CLP(FD) implementation of a solver that guaranteed optimal packing but provided slower solving speed. Internally, the container can view the selection of a solver as a constraint satisfaction problem where the input is the problem instance, the constraints are the solving policies and solver capabilities, and the output is a solver that provides the appropriate characteristics for the problem.

After selecting the appropriate solver, the container gathers the subject objects that are referenced by the problem and maps their relationship values to the various properties of the problem. This alleviates developers from having to traverse the design subject object graph and find the appropriate instances to gather data from.

In cases where constraint solving needs to be rerun and feature selections reconsidered if results change, developers can mark problems as active. The container registers itself as a listener on the various subject objects contained within it. When a role-based relationship of a subject object changes and that role is referenced by an active problem, the container automatically reruns the constraint solver. If the solution changes from its previous value, the container updates the appropriate role-based relationships on the feature design subjects, which in turn may update properties of the application. This design enables requirements-driven feature reselection at runtime.

For example, in *FeatureBind*, feature selection can be made an active problem. When the values of the underlying resources of the target infrastructure change beyond a certain threshold, the *TargetInfrastructure* design subject can be updated, automatically triggering the reselection of application features. This automated event-based constraint solving could allow an application to shutdown unessential functionality when memory runs low. Another example use case would be to shutdown dependent features when a feature, such as *AsynchronousMessagingInstalled*, failed.

3.5 Challenge 5: Specifying High-level Requirements that Require Multiple Solvers

Context. High-level requirements, such as “valid feature selection,” may not map to a single problem abstraction. For example, “valid feature selection” may require checking that resource constraints are met and that features are only selected if the target infrastructure has the appropriate OS and middleware stack to support the feature. In this example, the resource constraints map to bin-packing and the configuration constraints map to a local inference problem to figure out which features are valid in the context of the target configuration.
Problem. When high-level requirements require the solving of multiple problem types, the application may need to coordinate multiple constraint solvers. For example, to solve the “valid feature selection” requirement, the application may need to first solve the configuration problem to reduce the items (features) that are considered during the bin-packing (resource constraint solving).

To achieve this coordination, the design subject data must be transformed into the input format of the first solver. After solving is performed, the output must be translated into an input for the second solver. Finally, the output is mapped back to the application. Providing these transformations between constraint solvers and specifying workflows between solvers takes time and effort. Moreover, the workflows and transformations between solvers can easily become tightly-coupled to the input formats of the solvers and thus limit reusability.

Another challenge is that the solvers that may need to be coordinated may change for different instances of the same requirement. In our earlier example for bin-packing from Section 3.3, depending on the number nodes the application is being deployed to, different bin-packers are chosen. In this type of situation, the inference solver may be coordinated with either a CLP(FD) based bin-packer or a heuristic-bin packer with different interfaces. Providing workflows that can dynamically coordinate various types of constraint solvers with heterogeneous interfaces is challenging.
Solution → Problem pipelining. ROCs allows problem abstractions to be expressed as workflows of existing problems, called *ProblemPipelines.* To build a high-level problem workflow, developers create a problem class that implements the ProblemPipeline interface. The problem pipeline specifies an ordered list of problems that must be solved and how the outputs of the problem P_i map to the inputs of problems $P_{i+1}..P_n$. Moreover, adapters can be installed between problems to transform the output of one problem into the format that is suitable for the next problem in the pipeline. Each problem P_i can itself be a *Problem-Pipeline*, which allows the construction of even higher-level problem workflows from existing workflows, as seen in Figure 5.

For example, “valid feature selection” can be decomposed into two distinct type of problems. First, there are the configuration constraints governing the OS type, middleware, or other infrastructure that must be present on the target node that a feature is selected for. These constraints are based on the static properties of the target node. These configuration constraints can be very efficiently solved using Prolog’s standard inferencing. The second type of constraints are resource

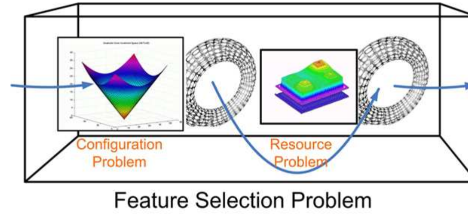

Fig. 5. Feature Selection as a *ProblemPipeline*

constraints that govern the amount of resources, such as CPU, consumed by the features selected. These types of constraints must be solved by a bin-packer. The goal of "valid feature selection" is to find a solution that satisfies the conjunction of these constraint sets.

The results returned by the satisfaction of the configuration constraints is the lists of features that can be supported by the target node. The result returned from the bin-packing is a set of features that fit into the resource constraints of the node. The output of the configuration solver can be used to prune the solution space for the resource problem by mapping the output of the inference solver, a set of valid features, to the ResourceConsumers input on the resource problem. This *FeatureSelectionProbem* creates a reusable higher-level problem abstraction that is defined as a workflow over a configuration and resource problem.

When the container is asked for a *SolutionProvider* for a *ProblemPipeline*, it dynamically builds a *SolutionProvider* that calls the appropriate sequence of solvers in the appropriate order. Moreover, the *SolutionProvider* built by the container manages the mapping of the results from one solving step to the input of later solving steps and applies the appropriate adapters where necessary. A *problemPipeline* allows developers to create complex solver workflows without writing complex transformations between solvers or working directly with solver input formats. Moreover, solver workflows can be reused and even higher levels of abstraction can be built on existing workflows.

The ROCs container handles the coordination between solvers by itself leveraging the *SolutionProvider* interface. At each stage in the *ProblemPipeline*, it queries itself for a *SolutionProvider* for the current problem. After solving is performed and results obtained, it retrieves a solution provider for the P_{i+1} problem stage and maps the output from *SolutionProviders* $Sp_0..Sp_i$ to the input for *SolutionProvider* Sp_{i+1}. Using the *SolutionProvider* abstraction internally allows the container to coordinate multiple solution providers with heterogeneous data input formats using a single common interface. Once again, ROCs leverages the Bridge pattern [17] to manage the heterogeneity of constraint solving.

The *ProblemPipelines* provided by ROCs also provide type safety. Since a *SolutionProvider* is created by the container, it checks to ensure that the mapping of outputs to inputs does not violate type restrictions. For example, if the inference problem was mapped to a *Foo* subject class and then the output was mapped to a resource problem where the *ResourceConsumer* had been typed as

the *Feature* subject, an exception would be thrown. If however, the *Foo* subject class derived from *Feature*, no exception would be thrown.

3.6 Challenge 6: Handling Requirement and Feature Conflicts

Context. It is possible in some cases the requirements that are being used to drive feature selection cannot be met. For example, when selecting a feature set for a resource-constrained target infrastructure, the infrastructure may not have sufficient memory to run even the most minimal feature set. If the target is running several other applications and cannot provide enough memory to support even the most minimal feature set, the solver will not be able to make a selection. In these cases, it is crucial that the feature selection mechanism provide a method of diagnosing the failure and negotiating resource or configuration changes on the target to make feature selection possible. We call this process of handling feature selection failure *model repair*.
Problem. The first challenge of providing automated repair is creating a way of diagnosing the cause of a feature selection failure. With numerous complex composition rules guiding the selection process, it is extremely hard to figure out *why* there is no valid feature set and *how* to repair the target or relax the selection driving requirements to overcome the problem. Simply failing to select a feature set and not providing an explanation would leave the reasoning of the underlying cause to the application user, without any hints on possible modifications (such as resource expansions) to make it work. In these situations, deducing the cause of selection failure could be as hard as finding a valid feature set manually.

A key question was what type of feedback should be provided to users. One approach we evaluated was marking non-functional requirements, such as CPU demand, that could not be satisfied and then returning a list of failed requirements as an error message. This approach is unsatisfactory, however, for the following reasons:

- For global constraints, such as resource constraints, the overall state of the application and target infrastructure determines whether or not the constraint succeeds. During feature selection, if the target infrastructure does not provide sufficient resources to host all of the features required for a feature set, it is not necessarily a single requirement that is causing the problem. Marking the first requirement that could not be met would not make sense since different packing or selection search orders could result in different requirements marked as the cause of failure.
- Even if the cause of the failure was marked in some manner, users would still need to manually determine how to modify the target infrastructure from its present state to make it compliant with the failed constraints. Although fixing the problem, by taking an action such as shutting down other applications, might appear trivial when the failing constraint was identified, changing the device state could have unforeseen affects on the other domain constraints. Once again, manual approaches do not scale for these types of constraint satisfaction problems.

Solution → Model repair operators. To address this challenge and allow for automated model repair when feature selection fails, ROCs provides a mechansism for specifying *model repair operators*. These operators can be applied to the feature model and selection driving requirements to make feature selection possible, as shown in Figure 6. ROCs defines a *model repair operator* as a function *repair(M,R)* that takes the original feature model, *M*, and requirements, *R*, and maps them to a new feature model, *M'*, and requirement set *R'*, such that feature selection is possible.

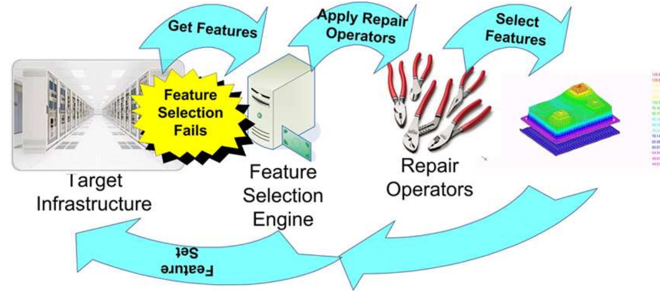

Fig. 6. Using Repair Operations

For example, if feature selection is not possible, due to insufficient resources, a repair operation *IncreaseMemory* can be created that shuts down unessential applications on the target infrastructure to increase memory. On the constraint solver side, this repair operator is translated into an increase on the bound $Dvc(Memory)$ that governs the amount of memory that can be consumed by the feature set. For example, if the repair operator sees that it can free up 50 Mbytes of memory, the constraint would be rephrased for the CLP(FD) solver as $\sum C_i(R) * DC_i < Dvc(R) + 50$.

Repair operations can also allow for unbounded relaxation of constraints. For example, rather than specifying that 50 Mbytes can be freed, the constraint can be removed entirely to allow the solver to select any feature set regardless of the memory it consumes. This type of unbounded repair operation can be leveraged with an optimization strategy, such as minimize consumed memory, to tell the solver to find the most minimal memory consuming feature set that is valid for the target infrastructure.

These two types of repair are distinguished as *Guaranteed Repair Operations* and *Conditional Repair Operations*. A *Guaranteed Repair Operation* is a repair function that ensures that the new model and constraints, *M'* and *R'*, allow a valid feature set to be found. A *Conditional Repair Operation* is a function that creates a new model and constraints, *M'* and *R'*, that allow a valid feature set to be found if and only if a repair constraint *RC* can be satisfied.

In ROCs, problem instances can have repair operations attached to them so that solvers have a fall back course of action when no solution is found. Each

repair operation is again an abstract repair that is specific to the type of problem it is attached to. For example, *IncreaseMemory* would be a specialization of the *IncreaseResource* repair operation that is valid in the context of a ResourceProblem. Repair operations are published in the problem catalog along with the problem types.

ROC's repair operation mechanism provides the following key characteristics that make automated conflict resolution for feature models possible:

- diagnosing and providing a meaningful explanation of feature selection failure to a user is not required
- the target infrastructure and feature model developers control repair by expressing only modifications that the application or target infrastructure is willing or capable of making
- repair is automated and does not require error-prone user intervention and
- repair can be optimized according to a cost function.

ROC's repair mechanism removes any necessity for user to diagnose failures. Even though repair is automated, the target infrastructure, feature model developers, and the application can still control the feature selection process by exposing allowed modification preferences, such as whether or not applications can be closed, third-party software downloaded, or memory reservations increased. Finally, this mechanism allows developers of the feature models to provide criteria for choosing the best feature set in the face of failures, which is important when costs are associated with feature sets.

4 Experimentation Results

To demonstrate the capabilities of ROCS, we took an initial implementation of a stovepiped feature selection engine (referred to as *FeatureBindOne*) developed in Java using a Prolog CLP(FD) solver for constraint satisfaction and refactored it to use our ROCs implementation in Java. We refer to our refactored, ROCs-based version of *FeatureBind* as *ROCFeatureBind*. This section presents results showing the lines of code needed to refactor our implementation, as well as performance results comparing the two implementations. The results show that the solvers created using ROCs provided better performance and reusability compared with the initial stovepiped implementation.

4.1 Analyzing Development Effort Using ROCs

FeatureBindOne used the Java Prolog Library (JPL) to make calls to an SWI Prolog engine [21]. It consisted of ∼1,400 lines of Java code and 180 lines of Prolog code. The selection engine can handle arbitrary local constraints based on <, =, and > comparisons on Feature and target infrastructure properties. For example, a constraint TargetInfrastructure.OS = "Linux", could be added to restrict a feature for use only on Linux systems. The selection solver could also evaluate dependencies between features.

Features can specify composition rules by specifying that N of a set of M other features must be present to activate the feature. A feature could also specify features that it was incompatible with. Finally, the selection engine had a resource constraint solver allowing feature modelers to express the amount of each type of resource on the target infrastructure consumed by a feature. The resource constraint solver ensured that for any feature set selected, the sum of the resource demands of the features did not exceed the resources on the target.

Roughly 200 lines of code were needed to refactor our implementation to use the ROCs infrastructure. This code included classes to make the multiple constraint solving rules in our initial implementation modular so that the resource solver or the configuration engine could be used independently of one another. By adding this 200 lines of code, the solvers from *FeatureBindOne* were decoupled from the original application feature model.

The refactored feature selection engine is known as *ROCFeatureBindOne*. By adding ∼20 lines of code to *ROCFeatureBindOne*, we allowed it to leverage another feature model implementation based on different Java classes. We also performed the same reimplementation with *FeatureBindOne*, without using ROCS, to connect it to the separate feature model implementation. This version of the refactoring required ∼295 lines of code.

The total lines of code to refactor *FeatureBindOne* using ROCs and connect it to the new feature model implementation required a total of 220 lines of code. Only ∼20 lines of this code could not be reused, however. Connecting it to further feature model implementations would thus require a similar ∼20 or so lines of code while with *FeatureBindOne*, an amount closer to 295 would be required. For large projects, with numerous types of solvers, constraint satisfaction problems, requirement types, and feature model implementations, the code savings would be much more significant. Moreover, code developed to solve various types of constraint problems and integrate diverse solver implementations would form a reusable catalog of solvers and constraint problems. New projects would simply map their requirements and features to existing primitives, rather than reinvent solver and other infrastructure from scratch.

4.2 Analyzing the Performance of ROCs

The next aspect of ROCs we tested was performance. We reimplemented our resource solver using the Java Choco constraint solver. This revision required ∼220 lines of code and allowed us to compare the solving performance of the strictly Prolog solution with our ROCS solution that first solved the configuration constraints in Prolog and then the resource constraints in the Java Choco solver. Our experiments were conducted on an IBM T43 laptop with 1 gigabyte of RAM and a 1.86ghz Pentium M CPU. More powerful hardware could be used to run the feature selection engine but the tests show that even with a laptop, *FeatureBind* provides good performance.

• **Experiment one: Comparing feature selection performance of FeatureBindOne and ROCFeatureBind.** Our performance tests evaluated the time required for the feature selection engines to find a maximal set of features

that fit into the resource and configuration constraints of the target infrastructure. Our application was based on the scenario described in Section 2, and involved selecting features from a train services application for a software variant for a mobile device. The services application included features ranging from the ability to order food and tickets for further travel legs to checking destination weather conditions. The feature model contained constraints based on configuration, such as OS and middleware installed on the target, business rules, such as the class of the owner's ticket, and resource constraints, such as the amount of memory and CPU consumed by features.

Since solving the resource constraints was the most time consuming portion of the feature selection process, the results evaluate the total time taken by the feature selection engine versus the number of features still in consideration during the resource constraint solving phase. We present the results in this manner since the configuration solver quickly eliminated candidate features based on incompatibilities in the target infrastructure, which drastically reduced solving time.

For example, for a device, such as a Treo phone, although the feature model contained 50 total features, 40 or more of them could be eliminated during the configuration phase. The feature set could therefore be found far more quickly than for a device where 20 or 30 possible features were left after solving the configuration constraints and were present during resource constraint solving. This result was expected since the resource solving is a form of bin-packing, an NP-Hard problem [18]. Thus the results show the time with respect to the number of features left after this initial solution space pruning. The results are shown in Figure 7. As can be seen from this figure, for 10 or less features,

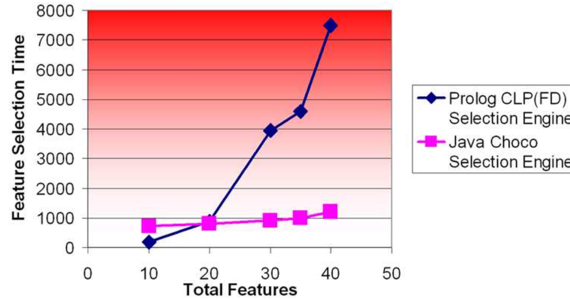

Fig. 7. Comparing FeatureBindOne and ROCFeatureBindOne

FeatureBindOne outperformed *ROCFeatureBindOne*. Past 10 features, however, *ROCFeatureBindOne* exhibited significantly superior performance and scaling. These results underscore the need to use different solvers depending on the characteristics of individual problem instances, *e.g.*, for small numbers of resource consumers, the Prolog-based solver should be used, whereas for numbers ¿ $\sim$10, the Choco solver should be used.

• **Experiment two: Comparing solution time of a hybrid dual-solver solution versus two single solver solutions.** Our next experiment looked at the feasibility of creating a hybrid solver to use the Prolog-based resource solver for small numbers of features and the Java solver for 11 or more features. We chose 11 as the breaking point to switch between solvers since this appeared to be the count at which the Choco and Prolog solutions were evenly matched, as shown in Figure 7. Since both solvers were already implemented in ROCs, this experiment required adding a simple two line `if` statement to select the appropriate solver based on the number of resource consumers in the resource problem. The results comparing the results of the three solvers are shown in Figure 8.

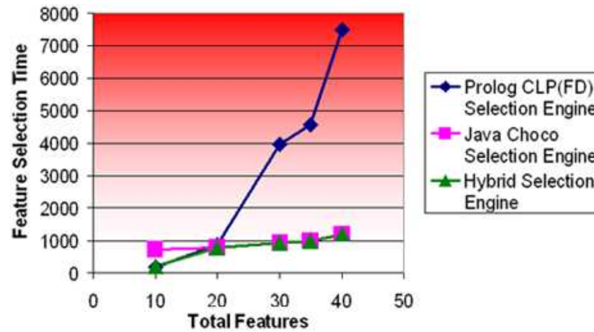

Fig. 8. Comparing FeatureBindOne, ROCFeatureBind, and a Hybrid Solver

As can be seen from the figure, the hybrid solver exhibited the best case performance of the two individual solvers. For small numbers of items, it delegates to the Prolog solver and obtains its performance. For larger numbers of resource consumers, it delegates to the Java Choco resource solver and exhibits its solving characteristics.

• **Interpretation of the results.** The results in Figures 7 and 8 underscore that for a small one-time initial development cost to integrate a new solver into the framework, the solver can be reused with substantially less development effort than a stovepiped solution. Moreover, by integrating multiple solvers into ROCs, powerful hybrid solvers can be built that not only delegate to different solvers based on constraint types but also on individual problem characteristics, such as the number of resource consumers in a resource problem. Choosing the right solver for each problem instance is critical to obtain the best performance. By decoupling solvers from both the feature model and the requirement specification, ROCs improves reusability and enables the dynamic construction of "solving pipelines" that use the right solver for each phase of the constraint solving involved in satisfying a high-level requirement. For example, if a requirement, R, is mapped to the conjunction of the constraint satisfaction problem instances, C_1,C_2,...C_n, rather than requiring the same solver to solve all of C_1 through C_n,

the best performing solver can be chosen for each individual problem. Although crossing solver boundaries and transforming data from one solver's format to the ROCs intermediary format and then to another solver's format incurs some overhead, this time penalty is insignificant for complex constraint satisfaction problems compared with the time required to solve the actual problems. For larger problem sizes, therefore, choosing the right solver is more important than minimizing trips across solver boundaries, as shown in Section 4.2.

5 Related Work

This section compares and contrasts our work on ROCs and using constraint solvers to guide binding decisions with related work on requirements specification and constraint solving.

In [22], Mannion et al presents a method for specifying PLA compositional requirements using first-order logic. The validity of a variant can then be validated by determining if a PLA satisfies a logical statement. The ROC's approach to PLA composition, based on features, expands on this idea by specifying PLAs as compositions of features using AND and XOR. ROCs also extends the work in [22] by including the ability to evaluate non-functional requirements not related to composition in validation. In particular, ROCs automates feature selection process using these boolean expressions and augments the selection process to take into account resource constraints, as well as optimization criteria. Although the idea of automated theorem proving is enhanced in [23], this approach does not provide a requirements-driven optimal feature selection engine like can be built with ROCs. Moreover, unlike ROCs, [23] does not provide an integration framework and decoupling mechanism to enable dynamic multi-solver solutions, which can provide significantly better performance than single-solver solutions, as shown in Section 4.2.

Mylopoulos et al. [24] present an extension to OO analysis and requirements capture called *Soft Goals* that allows developers to formally state the goals of their systems and perform analysis on them. The Soft Goals approach uses an abstraction based on entities that is similar to ROCs. Likewise, their approach proposes the use of goals to analyze the correctness of a system. ROCs extends Soft Goals in several ways. First, the Soft Goal approach does not provide a concrete method of defining goal types and checking their satisfaction, whereas ROCs defines how new goal types are declared (abstract problem specification) and how these goals are mapped to solvers. Moreover, [24] does not describe automated mechanisms for transforming a system description into a form that can be operated on by a constraint solver or checker, which as we have shown, is an important portion of reducing solver integration costs. Finally, the focus of [24] is on augmenting OO analysis, rather than the broader types of models and decompositions that ROCs supports.

Many complex modeling tools are available for describing and solving combinatorial constraint problems, such as those presented in [25, 10, 26–28]. These tools provide mechanisms for describing domain-constraints, a set of knowledge,

and finding solutions to the constraints. Unlike ROCs, however, these tools do not automate the transformation of data to and from the solver domain. Likewise, these tools do not support integration across multiple solver platforms and implementations, as ROCs does. Finally, problem abstractions and workflows developed in these tools are not portable across constraint solvers and must be redeveloped for each platform, whereas ROCs allows workflows to be described across multiple solver implementations and constraint satisfaction problem types.

6 Concluding Remarks

This paper presents a technique and toolset called *Role-based Object Constraint* (ROCs) programming that leverages subject-oriented design to decouple constraint specification from the feature model of an application and allow users to specify requirements in a high-level declarative format. ROCs provide a comprehensive framework for integrating constraint solvers with modeling tools while decoupling the solution algorithms from the metamodel of the language. The work presented in this paper addresses the challenges of requirements-driven feature selection and the high integration cost of using a constraint solver to automate feature selection.

From our experience developing ROCs and applying it to our case study and experiments, we learned the following lessons:

- For realistic size feature models, manual requirements-driven feature selection has serious scalability issues. These scalability issues become even more problematic when global constraints, such as resource constraints, are used to drive feature selection. As discussed in Section 2, by utilizing a constraint solver, requirements can not only be used to check whether a manually defined feature set is correct but can be leveraged to suggest and automate feature selections.
- Since, as described in Section 2 it is infeasible to perform requirements-driven feature selection manually, constraint solvers are therefore needed. There are numerous barriers to the inclusion of a constraint solver in a practical modeling environment that must be overcome. Even with ROCs, writing solvers is still challenging. As discussed in Section 3.3, by decoupling a solver from a specific feature model, however, solvers can be reused and their cost amortized across applications.
- When conflicting requirements are defined or a valid feature set cannot be found for the target infrastructure, repair operations, such as *IncreaseMemory*, can be used to specify the rules the constraint solver can use to resolve the conflict. As presented in Section 3.6, automating conflict resolution helps eliminate the need for manual intervention, which could be just as challenging as manually trying to find a way of satisfying the constraints. The repair mechanism prevents the application from having to fall back to tedious and non-scalable human intervention.

– As discussed in Section 4.2, by decoupling requirement specification from a specific input format of a constraint solver, high-level reusable problem specifications can be created, as well as complex workflows across solvers. Workflows across solvers allow the production of hybrid solvers that can evaluate and select solvers based on individual problem instance characteristics to provide the best performance.

In future work, we plan to more thoroughly explore the autonomic capabilities of ROCs using active problems and model repair. ROCs is part of the Generic Eclipse Modeling System (GEMS) open-source project and is available from www.sf.net/projects/gems.

References

1. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons, New York (2004)
2. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2002)
3. Coplien, J., Hoffman, D., Weiss, D.: Commonality and Variability in Software Engineering. IEEE Software **15**(6) (November/December 1998)
4. Antkiewicz, M., Czarnecki, K.: Featureplugin: feature modeling plug-in for eclipse. In: eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange, New York, NY, USA, ACM Press (2004) 67–72
5. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-specific Reference Architectures. Annals of Software Engineering **5**(0) (January 1998) 143–168
6. Sharp, D.C.: Reducing Avionics Software Cost Through Component Based Product Line Development. In: Proceedings of the 10th Annual Software Technology Conference. (April 1998)
7. Van Hentenryck, P.: Constraint Satisfaction in Logic Programming. MIT Press Cambridge, MA, USA (1989)
8. Jaffar, J., Maher, M.: Constraint Logic Programming: A Survey. constraints **2**(2) 0
9. Hao, J.K., Chabrier, J.J.: A modular architecture for constraint logic programming. In: CSC '91: Proceedings of the 19th annual conference on Computer Science, New York, NY, USA, ACM Press (1991) 203–210
10. Cohen, J.: Constraint logic programming languages. Commun. ACM **33**(7) (1990) 52–68
11. Nechypurenko, A., Wuchner, E., White, J., Schmidt, D.C.: 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)
12. D. Oppenheimer, A. Ganapathi, D.P.: Why do Internet Services Fail, and What can be Done about It? Proceedings of the USENIX Symposium on Internet Technologies and Systems (March 2003)

13. Clarke, S., Harrison, W., Ossher, H., Tarr, P.: Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code. Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (1999) 325–339
14. : Choco constraint programming system. http://choco.sourceforge.net/
15. Linderoth, J., Savelsbergh, M.: A computational study of search strategies for mixed integer programming. INFORMS Journal on Computing **11**(2) (1999) 173–187
16. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3) (1992) 293–318
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)
18. Coffman Jr, E., Galambos, G., Martello, S., Vigo, D.: Bin Packing Approximation Algorithms: Combinatorial Analysis. Handbook of Combinatorial Optimization. Kluwer Academic Publishers (1998)
19. Object Management Group: Data Distribution Service for Real-time Systems Specification. 1.0 edn. (March 2003)
20. Szyperski, C.: Component Software — Beyond Object-Oriented Programming. Addison-Wesley, Reading, Massachusetts (1998)
21. : Swi prolog. http://www.swi-prolog.org
22. Mannion, M.: Using First-order Logic for Product Line Model Validation. Proceedings of the Second International Conference on Software Product Lines **2379** (2002) 176–187
23. Mannion, M., Camara, J.: Theorem Proving for Product Line Model Verification. Fifth International Workshop on Product Family Engineering, PFE-5, Siena (2003) 4–6
24. Mylopoulos, J., Chung, L., Yu, E.: From object-oriented to goal-oriented requirements analysis. Communications of the ACM **42**(1) (1999) 31–37
25. Michel, L., Hentenryck, P.V.: Comet in context. In: PCK50: Proceedings of the Paris C. Kanellakis memorial workshop on Principles of computing & knowledge, New York, NY, USA, ACM Press (2003) 95–107
26. Smolka, G.: The oz programming model. In: JELIA '96: Proceedings of the European Workshop on Logics in Artificial Intelligence, London, UK, Springer-Verlag (1996) 251
27. Caseau, Y., Josset, F.X., Laburthe, F.: Claire: Combining sets, search and rules to better express algorithms. Theory and Practice of Logic Programming **2** (2004) 2002
28. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Duxbury Press (November 2002)