Improving Domain-specific Language Reuse with Software Product-line Techniques

Jules White, James H. Hill, Sumant Tambe, Aniruddha Gokhale, and Douglas C. Schmidt
Vanderbilt University
Nashville, TN, USA
{jules, hillj, sutambe, gokhale, schmidt}@dre.vanderbilt.edu

Jeff Gray
University of Alabama at Birmingham
Birmingham, AL, USA
gray@cis.uab.edu

Abstract

It is time consuming to develop a domain-specific language (DSL) or a composition of DSLs to model a system concern, such as deploying and configuring software components to meet real-time scheduling constraints. Ideally, developers should be able to reuse DSLs and DSL compositions across projects to amortize development effort. Reusing DSLs is hard, however, since they are often designed to precisely describe a single domain or concern. This paper presents an approach that uses techniques from software product-lines (SPLs) to improve the reusability of a DSL, DSL composition, and/or supporting tool by providing traceability from of language concepts to DSL design. We present a case study of four DSLs we developed to evaluate the need for—and benefits of—applying SPL reuse techniques to DSLs.

Keywords. feature models, domain-specific languages, reuse, software product lines, domain analysis, domain hierarchy

1 Introduction

Complex software systems, such as traffic management systems and shipboard computing environments, possess a number of concerns (*e.g.*, performance, reliability, and fault-tolerence) that must be realized and managed throughout the software lifecycle. Domain-specific languages (DSLs) [8] have emerged as a powerful mechanism for making these diverse concern sets easier to capture and reason about. For each system concern, a DSL can be designed to precisely capture key domain-level information related to the concern, while shielding developers and users from implementation-level details of the technical solution space.

Emerging trends and challenges. To create a DSL, developers must perform a careful analysis of the domain to design the language and produce the supporting tooling infrastructure, for editing, compiling, running, and/or analyzing instances of the language. Not only are these DSL development activities complex, but developers may need to evolve a DSL over time to find the right abstractions. Each evolution can have anywhere from a small to a massive impact on the tooling depending on the infrastructure used and type of changes made. As a result, DSL-based development processes can incur relatively high overhead with respect to overall project time and effort [8]. One way to ameliorate this overhead is to amortize DSL development costs across projects.

For example, reusing existing DSL tooling infrastructure across development projects can help reduce the overall cost of these projects. A new development project, however, may have a unique set of concerns to model that do not precisely match the requirements for which existing DSLs were designed. A key question facing developers is therefore how to adapt an existing DSL or set of DSLs (*i.e.*, a DSL *composition*) to a set of requirements.

Reusing DSLs can be hard, however, since they are often designed to focus on specific system concerns. While the narrow scope of a DSL provides much of its power, it can also (overly) couple the DSL to a particular group of assumptions, making it hard to reuse for a new set of requirements. What is needed is therefore a technique for systematically reusing DSLs and DSL compositions to simplify their adaptation to new requirements.

Solution approach → Applying software product-line configuration techniques to DSLs. Software product-lines (SPLs) [5] are a systematic reuse technique that supports (1) building a family of software products such that variability can be customized for specific requirement sets, (2) capturing how individual points of variability affect each other, and (3) configuring product variants that meet a range of requirements and satisfy constraints governing variability point configuration. SPLs are used in domains where software development costs are high, safety and performance are critical, and redeveloping software from scratch is economically infeasible. SPLs have successfully been employed in domains such as avionics mission computing, automotive systems, and medical imaging systems.

This article provides two contributions towards improving reusability and decreasing language reuse errors for DSLs and DSL compositions. First, we show that a single DSL can have built-in variability and codified configuration rules to enable its refinement for multiple domains. Second, we show how SPL techniques can be used to codify the usage rules for a DSL composition's constituent DSLs, the concerns covered by the DSLs, and the variations in DSL usage. By codifying these DSL composition concepts, developers are provided with a map of how to correctly modify and reuse DSLs and DSL compositions across projects.

Our SPL-based reuse techniques for DSLs builds upon the following prior work on SPLs and DSLs:

• *Feature Models*, which codify the points of variability in a software product and the rules governing the settings for each point of variability [7]. A feature model is a tree-based structure where each node in the tree represents a point of variability or unit of functionality in the product. The root of the tree represents the most generalized concept in the product and successively deeper levels of the tree indicate refinement of the software. The parent-child relationships indicate configuration constraints that must be satisifed when choosing values for points of variability.

Kang et al. [7] and Beuche et al. [4] have successfully applied feature modules to manage SPL variability in a number of domains. Feature models provide a solid foundation for improving reusability by codifying reuse rules. Moreover, a number of techniques have been developed to formally analyze feature models and identify configuration errors [10], identify constraint inconsistencies [1], and automate feature selection [3].

• *DSL refinement*, which is the adaptation of a DSL for a new set of requirements. A DSL is defined by a *metamodel*, which is a specification of the DSL's key concepts and syntax. Voelter [9] has investigated the use of model transformations for refining an architectural DSL. His technique describes the variability in an architectural DSL using a feature model. To refine the architectural DSL, developers select a set of architectural modeling features that should be present in the refined language. Based on the feature selection for the new domain, model transformations automatically add/remove the corresponding metamodel elements.

Although prior work provides a good starting point for addressing DSL reusability challenges, there are a number of limitations. First, SPL techniques have been extensively studied in the context of software but not in the context of DSL design. New methodologies are therefore needed to codify how SPL techniques can be used to manage DSL refinement and DSL composition adaptation. Although some researchers have applied SPL techniques to individual DSLs [9], generalized methodologies for applying these techniques to arbitrary DSLs have not yet been extrapolated. Moreover, SPL variability management techniques have not been applied to DSL composition and reuse. This article presents a general methodology for using feature models to manage DSL and DSL composition reuse.

2 Experience Report: PICML, Scatter, CUTS, and CQML

The Institute for Software Integrated Systems (ISIS) at Vanderbilt University has developed many DSLs and associated tools for a wide range of modeling concerns, such as component-based application design, deployment and configuration of applications in distributed real-time and embedded (DRE) systems, and system execution modeling. We are frequently developing DSLs for new domains. To showcase the complexity of reusing DSLs and DSL compositions for new requirement sets, we provide an experience report based on four related DSLs we developed, as shown in Figure 1.

These DSLs have been built atop two different modeling platforms, the Generic Modeling Environment (GME)[1] and the Generic Eclipse Modeling System (GEMS)[2]. GEMS itself is built atop the Eclipse Modeling Framework (EMF)[3]. The first DSL we describe is PICML, which is used for visually composing CORBA Component Model (CCM) applications. PICML is a GME DSL that is focused on the solution domain. The second DSL is Scatter, which is a GEMS DSL for modeling the deployment of software components to hardware nodes in a distributed system and is focused on modeling the problem domain. The third DSL is CQML, which is a GME DSL for specifying QoS constraints on systems and is also aligned with the solution domain. The fourth DSL is CUTS, which is a GME DSL for analyzing the performance of DRE system architectures and is focused on the problem domain.

Significant effort has been expended developing the four DSLs and their associated tooling. PICML has been devel-

[1] www.isis.vanderbilt.edu/Projects/gme
[2] www.eclipse.org/gmt/gems
[3] www.eclipse.org/emf

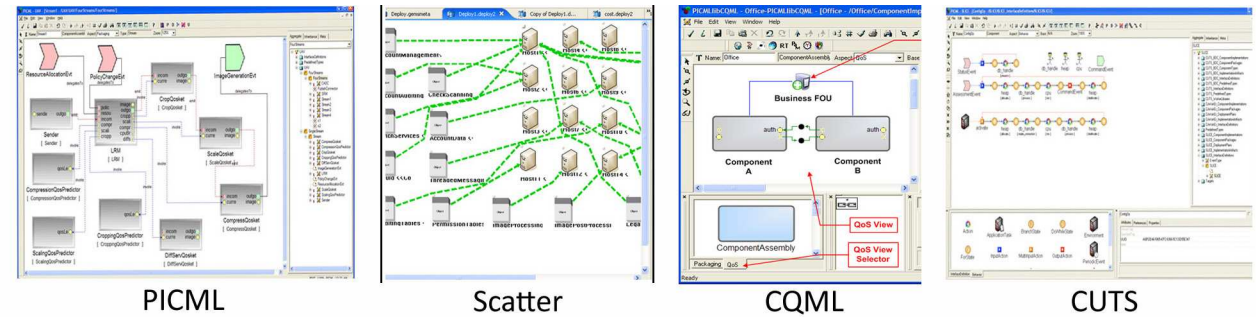

Figure 1. The PICML, Scatter, CQML, and CUTS DSL Family

oped over the course of five years and continues to evolve. Scatter and CUTS have also been developed over a period of four years. CQML is the youngest DSL with roughly two years of development.

The DSLs we chose for our investigation form a closely related family of DSLs. For example, a CUTS model of the behavior of DRE system QoS can be built and used to perform experiments to test the response time of critical end-to-end request paths through the system. CUTS models, however, depend on an external model of how the software should be mapped to hardware nodes. PICML and Scatter provide facilities for capturing this missing deployment information.

Scatter focuses on capturing deployment resources and real-time scheduling constraints and uses this information to automate the decision of how to map software to hardware. PICML focuses on allowing developers to manually specify software to hardware mappings, but does not capture resource or scheduling constraints. It can be augmented with CQML, however, to capture scheduling constraints.

We developed a complex DSL composition from PICML, CUTS, and Scatter in the context of the Lockheed Martin NAOMI project [6], which is studying the use of multiple DSLs to model the development of software for controlling traffic lights in intersections. NAOMI uses PICML to model the software components, Scatter to derive suitable deployment topologies in NAOMI, and CUTS to perform experiments to evaluate the QoS of the traffic software.

After development of NAOMI began, we addressed similar problems related to modeling deployment topologies and testing software performance in the context of the Air Force Research Labs (AFRL) SPRUCE[4] project. In SPRUCE, we modeled and tested the deployment of software to hardware in avionics systems. Due to the similarity between the NAOMI and SPRUCE requirements, we wanted to reuse as much of the original DSL composition as possible. The remainder of this paper uses PICML, Scatter, CQML, and CUTS to motivate the need for—and complexity of—reusing these DSLs for new requirements sets.

[4] www.sprucecommunity.org

3 Challenges of Domain-specific Language Reuse

There is a tension between a DSL's domain specificity and its reusability. On one hand, the more precisely a DSL is crafted to match its domain, the easier and more accurately it can describe a solution. On the other hand, DSLs and their supporting infrastructure can be expensive to develop, so reusability is desirable. This section explores the challenges of maintaining DSL specificity and accuracy, while simultaneously facilitating reuse.

3.1 Challenge 1: DSL Refinement

Developing a robust DSL that accurately describes domain concepts and is intuitive for domain experts can be a long and iterative process. An initial prototype of the DSL is developed and then over a period of time the DSL concepts and notations are refined by modeling existing and new systems. The DSL refinement process may take months. Developing code generators, constraint checkers, model execution engines, and other dependent tools also requires significant time and effort.

Developers often find a group of domains that exhibit substantial similarities but enough differences to warrant separate DSLs. For example, PICML was originally developed to model CCM applications. Over time, however, the need arose to model Enterprise Java Beans (EJB) applications, which have many similarities to CCM (*e.g.*, EJB has similar component and home concepts to CCM), but does not share event source/sink features. Similarly, Scatter (a DSL aimed at specifying deployment constraints and topologies) was originally developed to model deployment problems in the automotive domain. Since its original development, we needed to use Scatter in other domains (such as flight avionics) that did not share exactly the same types

of deployment constraints.

To reduce DSL development cost, PICML could be reused for EJB applications. Although this approach is possible, it would expose EJB developers to certain details, such as event sources and sinks, that are not relevant to their target domain. Reusing Scatter in the avionics domain would expose developers to crash survivability constraints that are not relevant for planes. This type of exposure to unnecessary details would eliminate many benefits of using a DSL.

Another approach to reuse would be to refine the PICML metamodel for EJB or generalize it for component-based software by eliminating CCM-specific modeling elements. For example, PICML provides a modeling element to represent event sources on components and event sinks on components that consume the events. The event source and sink are not directly applicable to EJB. Removing the event source and sink notations from the PICML metamodel is non-trivial, however, since PICML has over 700 interrelated metamodel elements. Eliminating the event source and sink notations requires removing over 30 other metamodel elements, *e.g.*, there are over 15 elements related to specifying properties of event channels that are not needed if event sources and sinks are removed.

Reusable code libraries, aspect-oriented programming, and other language features can help modularize the implementation of software. Similarly, various techniques, such as MetaEdit+'s fragments[5], GME's metamodel composition, the model management techniques of the ATLAS Model Management Architecture (AMMA)[6], or the aspect-oriented features of openArchitectureWare (oAW)[7] can help modularize DSLs. To properly leverage these implementation-level modularization techniques, however, developers must still have design-level information, such as composition rules for software components or traceability between a domain concept and a language element of the DSL.

A key problem in refining or modifying an existing DSL, regardless of the tool used to implement it, is having traceability information for mapping: (1) concepts to the metamodel or grammar for the DSL (requirements to design) and (2) the metamodel or grammar specification to its implementation in a particular tool (design to implementation), such as EMF. Moreover, additional information is needed to ensure that neither design integrity, such as the completeness of the representation of concepts, nor implementation correctness is violated by modification/refinement of the DSL. Capturing these elements of traceability and DSL design integrity is important and is an issue regardless of the tool infrastructure used to build a DSL. Section 4.1 describes how we address this challenge by using feature models to codify DSL semantic constraints and map concepts to DSL design.

[5] `www.metacase.com`

[6] `www.sciences.univ-nantes.fr/lina/atl`

[7] `openarchitectureware.org`

3.2 Challenge 2: Multi-DSL Composition

DSLs are often tightly aligned with a single narrow slice of system concerns. Multiple DSLs may therefore be needed to capture the important concerns relevant to a system's requirements. When developing a multi-DSL development process, developers must ensure that they provide adequate coverage of concerns through the DSLs. For example, developers must ensure that the DSL composition properly captures the real-time scheduling, deployment, and performance concerns of the NAOMI traffic light system outlined in Section 2. This system could potentially use a number of different DSLs to capture the information related to the capabilities of the system's hardware nodes.

For example, developers could use Scatter to model each piece of hardware, the real-time scheduling constraints on components, and the resources, such as RAM, available on each node. Developers could also instead opt to model the nodes through PICML. If developers need to ensure that the nodes have sufficient resources to host the provided components, the Scatter DSL is more applicable. Choosing PICML would not adequately cover the resource allocation concern. If real-time scheduling constraints were needed, either Scatter or a combination of PICML and CQML could be used.

In the traffic light system, there are roughly a dozen concerns related just to the deployment of software components to hardware that are captured by multiple DSLs implemented on several tooling platforms. For example, developers need to capture information related to component replication for fault-tolerance, node resource constraints, component real-time scheduling requirements, and cost information for budgeting. Crafting a DSL composition to properly cover a large set of concerns is not easy without traceability from the design concepts that must be covered to the individual DSLs that provide the concepts. This traceability challenge of mapping and understanding the relationships between concepts and DSL design decisions is independent of the tools used to implement the DSL.

Variability in the DSLs themselves further complicates the design of a DSL composition. For example, PICML can be refined for EJB by removing event and deployment information. Removing the deployment modeling capabilities from PICML to handle EJB, however, leaves CUTS without needed deployment information to generate experiments.

Developers must not only ensure that a DSL composition provides proper concern coverage, but also that the precise refinement of the DSLs being used provides the required concern coverage and adheres to any composition

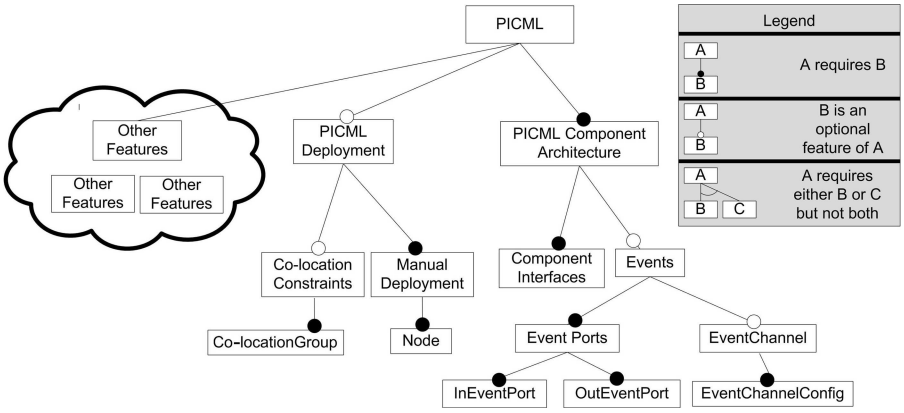

Figure 2. PICML Feature Model Snippet for Event Elements

constraints [2]. Managing this variability and adding this consideration into the adaptation of existing DSL compositions to new requirements is hard without explicit traceability from concepts to individual DSL design features. Section 4.2 describes how we address this challenge by capturing DSL composition configuration rules in feature models.

4 Applying SPL Configuration Techniques to DSL-based Development

DSLs and their associated development processes are often tightly-coupled to a single set of requirements or concerns. Although DSLs are domain-specific, they do possess points of variability, such as concepts that can be added or removed. For example, PICML can have metamodel elements removed as long as developers have traceability and constraint information to know how to perform the modifications properly. Moreover, if developers know why a DSL composition has a particular structure and how the structure can legally be modified, the composition can be adapted to new types of concerns. The missing ingredient that produces the reuse challenges summarized in Section 3 is that there is often no model that traces how concepts map to language design and that captures the points of variability and their inter-relationships in DSL refinement, composition, and tools. This section shows how SPL techniques can be used to fill in this gap and increase DSL, DSL composition, and DSL tool chain reusability.

4.1 Managing DSL Refinement via Feature Models

A key problem outlined in Section 3.1 is that developers do not have concept to design traceability information or the rules for modifying a DSL's metamodel to ensure that a semantically valid DSL refinement is produced. An approach to solving this problem is to build a configurable DSL and use a feature model to document (1) how concepts map to metamodel elements and (2) the semantic dependencies between metamodel elements. The feature model describes why specific DSL language elements exist, which elements are semantically related, the semantic constraints for adding/removing elements, and the rules for determining what is a valid metamodel refinement. The DSL elements are represented at the tool-independent level and mappings to tool-specific implementation can also be defined. Each refinement of the DSL's metamodel is mapped to a feature selection that can be checked for semantic validity.

Figure 2 shows a simplified feature model of the metamodel elements related to the PICML event elements discussed in Section 3.1. The feature model is constructed in stages, capturing the most general tool-independent concepts at the top levels and gradually refining more specific concepts until actual metamodel elements or tool-specific metamodel element mappings are reached at the leaves. For example, the general concept `PICML Component Architecture` is refined to the more specific concepts of `Component Interfaces` and `Events`. The leaves beneath `Events` capture concepts in terms of actual metamodel elements, such as `InEventPort` and `OutEventPort`, which in this case, map directly to GME metamodel elements.

Developers can use this feature model of PICML to build semantically correct refinements of the DSL. For example, if developers want to remove the concept of events to refine for EJB, they can find the `Events` feature and then remove all the language elements and mappings to metamodel elements that appear as children beneath the `Events` feature. Moreover, if a more precise refinement is desired, developers could keep the concept of events—possibly to model EJB's Java Messaging Service (JMS)—but remove the CCM-specific concept of event channels. The feature model precisely captures traceability from concepts to lan-

guage design and rules for correctly modifying the 700 language elements in the PICML metamodel to refine concept coverage.

4.2 DSL Family Configuration with Feature Models

The challenge outlined in Section 3.2 described how developers often do not know why a particular set of DSLs were composed and how the composition covered a set of concepts due to lack of traceability information. For example, it is not clear how using PICML to describe deployment capabilities differs in concern coverage from using Scatter. Moreover, when a DSL composition must be modified to cover a new concern (such as the SPRUCE aeronautics domain) developers do not have a roadmap of the interactions between DSLs, which makes it hard to determine which features can be added or removed.

To address this issue, feature models can be used to codify (1) what concerns are covered by each member of a DSL composition, (2) what dependencies or exclusions exist between DSLs, and (3) how DSL refinements affect concern coverage. Figure 3 presents a feature model of the DSL composition covering PICML, Scatter, CUTS, and CQML. The DSL composition is represented as the root feature in this figure. Beneath the root feature are features providing a general categorization (*e.g.*, `Deployment` and `Performance`) of the DSLs involved in the composition. Beneath the categorization features are the actual DSL concepts that can be used to capture the concern. For example, either `Scatter Deployment` or `PICML Deployment` can capture deployment information. At the leaves beneath the DSL concepts are modeling capabilities provided by the DSL. For example, Scatter provides `Automated Deployment`, but PICML does not.

The feature model not only tells developers what DSLs can be used and their capabilities, but also specifies how refinements of DSLs affect each other. For example, if PICML is refined to remove the `PICML Deployment` concepts, Scatter and PICML can be used together. If developers want to evaluate how different wide area network (WAN) properties affect performance, they need to use a refinement of CUTS that include `CUTS Emulab` and a refinement of PICML that includes `WAN` concepts.

4.3 The Cost of DSL Reuse

Using SPL techniques to produce reusable DSLs has a cost associated with it. Generally, we can represent the cost of developing a DSL using standard techniques as:

$$Cost(DSL) = Metamodel + Editors + Generators$$

To develop a reusable DSL and infrastructure, developers pay an up-front cost:

$$Cost(DSL_R) = C_1 Metamodel + C_2 Editors + C_3 Generators$$

where C_1 is a multiplier for the extra effort to build a feature model of the language or language family, C_2 is the cost of building a more advanced editing infrastructure that can be reconfigured based on the features selected for a variant of the DSL, and C_3 is the overhead of creating code generation/analysis infrastructure that can be reconfigured for different DSL variations.

In our experience, C_1 (the cost of producing the feature model of a language after the DSL's metamodel is developed) is not high. The cost of C_2 is also typically low, due to the excellent tool support for automatically generating graphical and textual editors for a DSL provided by tools, such as GME, GEMS, GMF[8], oAW's xText, and the Textual Concrete Syntax project (TCS)[9]. For example, if the networking concepts are removed from the Scatter metamodel, GEMS can automatically update the graphical editor, which results in the regeneration of roughly 4,700 lines of code. Voelter [9] has shown how model transformations and oAW's xText can be used to automatically regenerate a textual editor with code completion and syntax highlighting as a DSL's feature selection changes.

Our experience shows that the major difficulty in reusing DSLs lies in the modularization of the code-generation and analysis infrastructure, C_3. For some platforms, such as GME, code generators are written in third-generation languages, such as C++, that require more effort to achieve modularity. Tools that leverage other code generation platforms, such as oAW, can use advanced modularization features, such as oAW's support for aspect-oriented programming in code generation templates.

In the end, a cost benefit analysis must be performed. If three separate DSLs are developed without a reusable approach, a cost of:

$$3\ X\ Cost(DSL)$$

is incurred. With a reusable DSL approach, the initial DSL cost, $Cost(DSL_R)$, is higher, but subsequent DSLs cost less:

$$Cost(DSL_R) + 2\ X\ Reuse(DSL_R)$$

$$Reuse(DSL_R) < Cost(DSL)$$

The price of refining and reusing an existing DSL, $Reuse(DSL_R)$, is typically a function dominated by the impact of C_3. For each scenario, developers must estimate whether the DSL will be reused enough times to make the

[8] `eclipse.org/gmf`
[9] `www.eclipse.org/gmt/tcs`

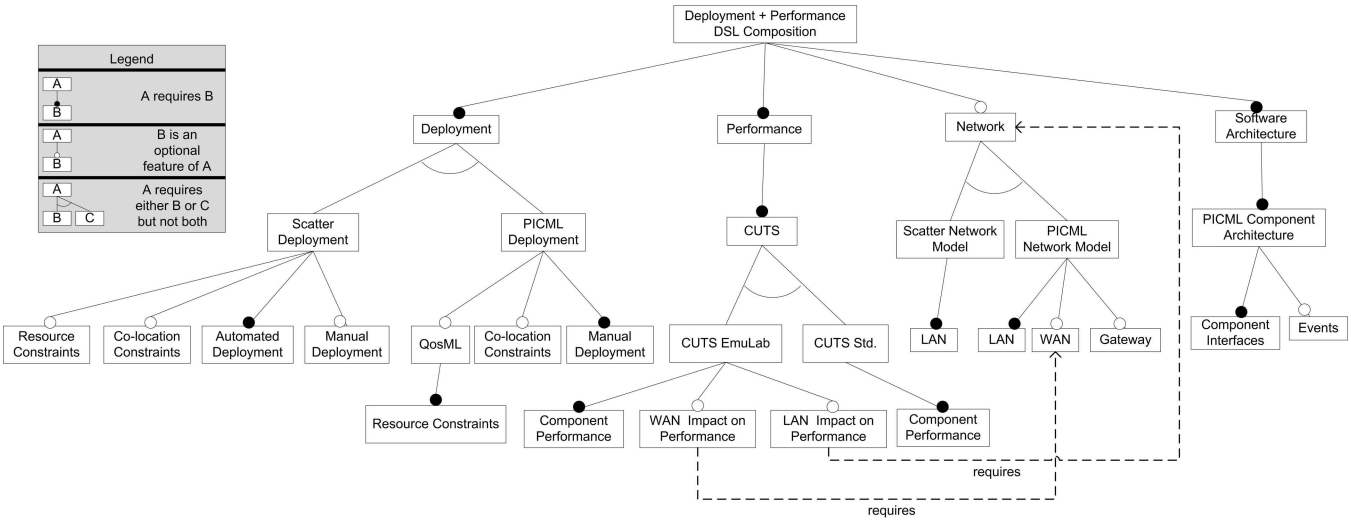

Figure 3. A Feature Model for the PICML/CUTS/Scatter/CQML DSL Family

reduced price, $Reuse(DSL_R)$, overcome the initial overheads of C_1, C_2, and C_3. In our work, we have found numerous instances where the initial price of reusability paid off.

5 Concluding Remarks

This article motivated the need for improving DSL reusability and showed how SPL reuse techniques can be applied to DSL refinement and DSL composition adaptation to improve reusability. In particular, we showed how feature modeling techniques can be used to document the semantic rules for modifying metamodels and DSL compositions.

The modeling tools, GEMS and GME, are available from `www.eclipse.org/gmt/gems` and `www.isis.vanderbilt.edu/Projects/gme`, respectively. The DSLs can be obtained from `www.dre.vanderbilt.edu`.

6 Acknowledgements

We would like to thank the Lockheed Martin Advanced Technologies Lab, the Air Force Research Laboratory, and the National Science Foundation (NSF-CAREER-0643725) for their support of this work.

References

[1] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, pages 7–20, Rennes, France, September 2005.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[3] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated Reasoning on Feature Models. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering*, pages 491–503, Porto, Portugal, June 2005.

[4] D. Beuche, H. Papajewski, and W. Schrder-Preikschat. Variability Management with Feature Models. *Science of Computer Programming*, 53(3):333–352, 2004.

[5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, USA, 2002.

[6] T. Denton, E. Jones, S. Srinivasan, K. Owens, and R. Buskens. NAOMI-An Experimental Platform for Multi-modeling. In *Proceedings of MODELS*, pages 143–157, Toulouse, France, October 2008.

[7] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, 2002.

[8] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[9] M. Voelter. A Family of Languages for Architecture Description. In *Proceedings of the OOPSLA Workshop on Domain-Specific Modeling*, pages 86–93, Nashville, TN, October 2008.

[10] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortez. Automated Diagnosis of Product-line Configuration Errors in Feature Models. In *Proceedings of the Software Product Lines Conference (SPLC)*, pages 225–234, Limerick, Ireland, September 2008.

7 About the Authors

Dr. Jules White is a Research Assistant Professor at Vanderbilt University. Dr. White's research focuses on ap-

plying a combination of modeling and constraint-based optimization techniques to the deployment and configuration of complex software systems.

James H. Hill is a PhD candidate in Electrical Engineering and Computer Science at Vanderbilt University. His research focuses on using model-based analysis to identify QoS flaws in DRE systems.

Dr. Jeff Gray is an Associate Professor in the Computer and Information Sciences Department at the University of Alabama at Birmingham, where he co-directs research in the SoftCom Laboratory. His research interests include model-driven engineering, aspect orientation, code clones, and generative programming.

Sumant Tambe is a PhD candidate in Electrical Engineering and Computer Science at Vanderbilt University. His current research interests include model-driven engineering for DRE systems.

Dr. Douglas C. Schmidt is a Professor of Computer Science and Associate Chair of the Computer Science and Engineering program at Vanderbilt University. He has published 9 books and over 400 papers that cover a range of topics, including patterns, DSLs, and DRE middleware.

Dr. Aniruddha S. Gokhale is an Assistant Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University. Dr. Gokhale's research combines model-driven engineering and middleware for DRE systems.