Improving Domain-specific Language Reuse through Software Product-line Configuration 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. It can be hard to reuse DSLs, 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. We present a case study of four DSLs we have developed to evaluate the need for—and benefits of—applying SPL reuse techniques to DSLs.

Keywords. feature models, domain-specific languages, reuse

1 Introduction

Emerging trends and challenges. Complex software systems, such as traffic management systems and shipboard computing environments, possess both functional concerns (*e.g.*, execution correctness) *and* quality-of-service (QoS) concerns (*e.g.*, performance, reliability, and fault-tolerence) that must be realized and managed throughout the software lifecycle. Domain-specific languages (DSLs) [7] 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.

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 and each evolution can impact the tooling infrastructure significantly. As a result, DSL-based development processes can incur relatively high overhead with respect to overall project time and effort [7]. 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, therefore, is 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 set of requirements, making it hard to reuse for a new set of requirements. What is needed, therefore, is 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) [4] 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 is built 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 [6]. 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.

To adapt an SPL to a new set of requirements, developers create a *variant*, which is defined by a subset of the features from the feature model. An important property of an SPL is that the validity of a variant can be checked to ensure that the feature selection adheres to the feature model rules. The feature model documents reuse rules that facilitate the adaptation of the software for new requirement sets and provides a basis for checking reuse correctness.

Kang et al. [6] and Beuceh et al. [3] 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, identify constraint inconsistencies, and automate feature selection.

• *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 [8] 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 this 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 [8], 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 provides a general methodology for using feature models to manage DSL and DSL composition reuse.

2 Case Study: PICML, Scatter, CUTS, and CQML

The Institute for Software Integrated Systems (ISIS) at Vanderbilt 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 still constantly need to develop DSLs for new domains. To showcase the complexity of reusing DSLs and DSL compositions for new requirement sets, we provide a case study based on four related DSLs we developed, 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]. The first DSL we describe is PICML, which is a GME DSL for visually composing CORBA Component Model (CCM) applications. The second DSL is Scatter, which is a GEMS DSL for modeling the deployment of software components to hardware nodes in a distributed system. The third DSL is CQML, which is a DSL for specifying QoS constraints on systems. The fourth DSL is CUTS, which is a GME DSL for analyzing the performance of DRE system architectures.

Significant effort has been expended developing the four DSLs and their associated tooling. PICML has been developed over the course of five years and continues to evolve.

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

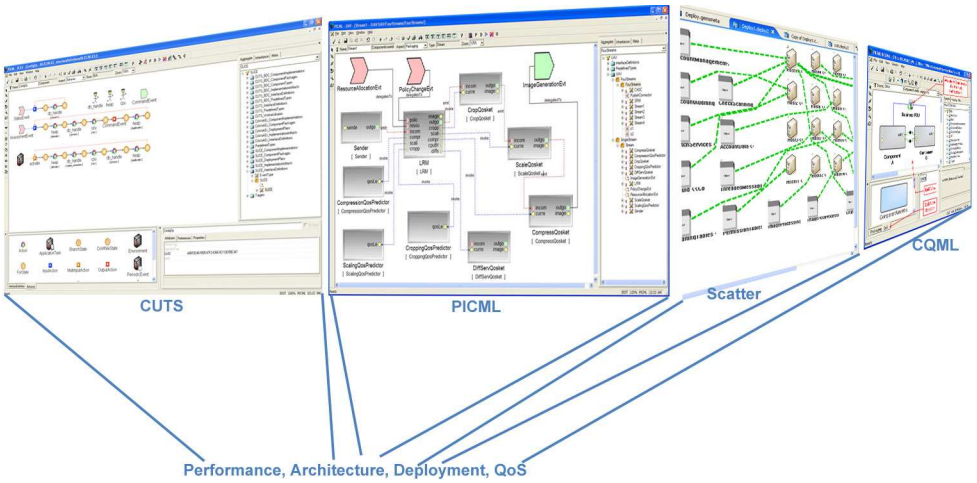

Figure 1. The PICML, Scatter, CUTS, and CQML DSL Family

Scatter and CUTS have also developed over a period of four years. CQML is the youngest DSL with roughly two years of development.

The DSLs we chose for our experience report 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 resource and real-time scheduling constraints and using 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 [5]. This project is studying the use of multiple DSLs to model the development of software for controlling traffic lights in intersections. PICML is used to model the software components, Scatter is used to derive suitable deployment topologies, and CUTS is used 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 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 requirement sets.

3 Challenges of Domain-specific Language Reuse

There is an inherent 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 checking, 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.

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. 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 to eliminate irrelevant 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 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. Determining how to adapt languages like PICML for a new domain is hard. Section 4.1 describes how we address this challenge by using feature models to codify semantic constraints between metamodel elements.

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. 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.

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 therefore 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 constraints [2]. Managing this variability and adding this consideration into the adaptation of existing DSL compositions to new requirements is hard. Section 4.2 describes how we address this challenge by capturing DSL composition configuration rules in feature models.

3.3 Challenge 3: Tooling Reuse

Refining a DSL or adapting a DSL composition can have a direct affect on the supporting tool infrastructure. Eliminating elements of a DSL requires also updating any editors or compilers of the DSL to reflect that the removed elements are no longer legal entities. Moreover, any code generators that depend on the removed elements will also need refactoring.

For example, Scatter can be refined to remove real-time scheduling constraints from its metamodel so that it can be used for EJB. Once Scatter's metamodel is refined, the Java implementations of its graphical editor and deployment solvers must also be modified to remove the references to the eliminated DSL elements. Scatter's deployment solver and graphical editor contain approximately 48,000 lines of code that must be searched to find any references to unnecessary elements. Without a clear model of how the elimination of real-time scheduling concepts affect the editor and solvers, it is hard to refine Scatter's supporting tool infrastructure. Section 4.3 describes how we address this challenge by using code generation and metaconfiguration to refactor existing tooling.

Figure 2 shows a simplified feature model of the metamodel elements related to the PICML event elements discussed in Section 3.1.

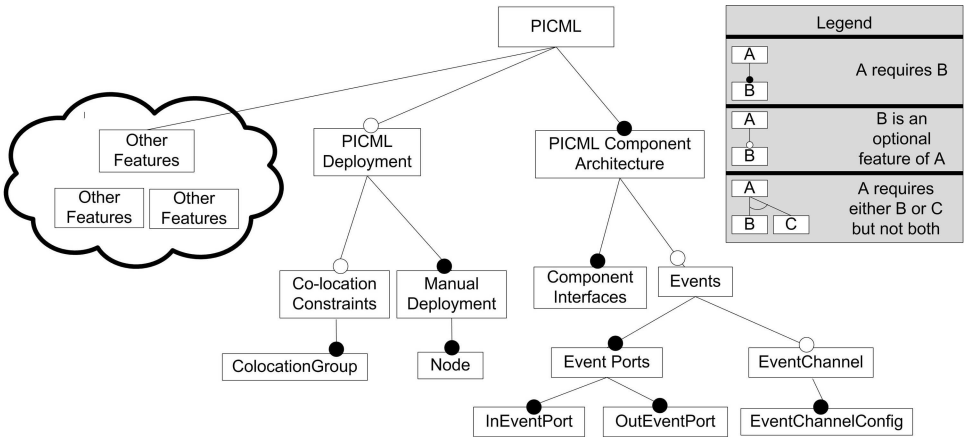

Figure 2. PICML Feature Model Snippet for Event Elements

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. For example, PICML can have metamodel elements removed as long as developers 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 of the variability 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 know 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 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. Each refinement of the DSL's metamodel is mapped to a feature selection that can be checked for semantic validity.

The metamodel is constructed in stages, capturing the most general concepts at the top levels and gradually refining more specific concepts until actual metamodel elements 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`.

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 metamodel elements that appear as leaves 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 how to modify the 700 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. 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

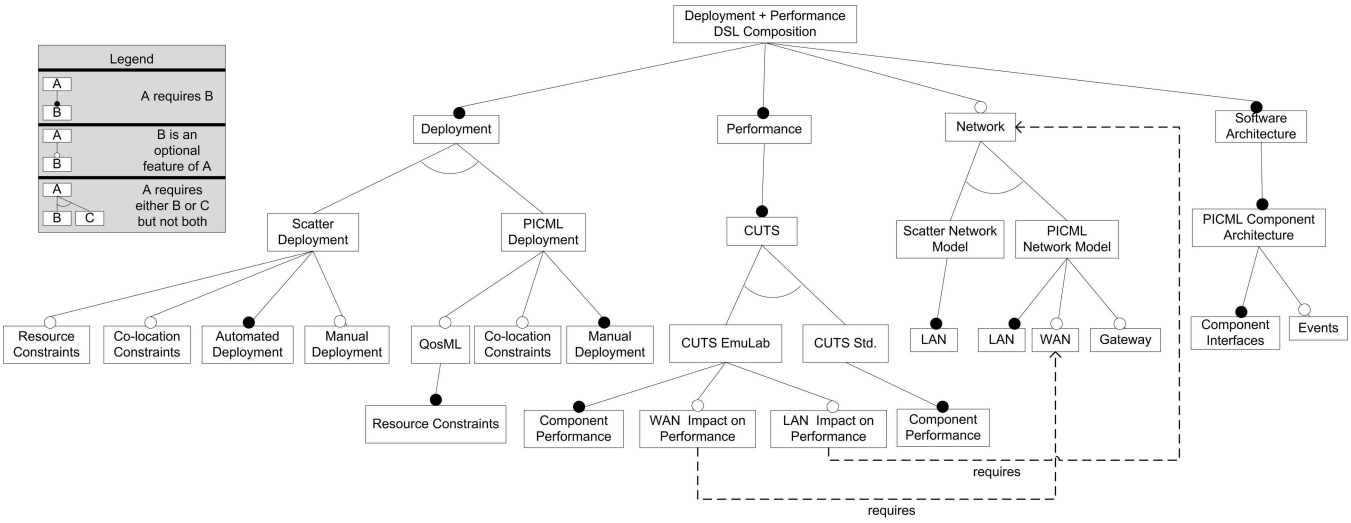

Figure 3. A Feature Model for the PICML/CUTS/Scatter/CQML DSL Family

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 tells developers not only 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 Tool Reuse

Section 3.3 summarized the challenges related to refactoring existing tool infrastructure when DSL refinement or DSL compositions change. SPL reuse techniques can be directly applied to this challenge. SPLs rely on a modular software architecture where feature configuration changes can be mapped to changes in the software configuration.

Building modular tool infrastructure is a complex task that can be guided by existing SPL techniques. Developers do incur an upfront cost to build modularity into tool infrastructure. At Vanderbilt, we have found the following techniques can help lessen the cost of implementing this modularity:

• **Tool refactoring via code generation**. This approach to building modularity into tools applies model-driven engineering techniques to develop editors for DSLs. Developers can therefore leverage the metamodel and feature model of the DSL to generate the code required to implement key tooling. Using code generation can significantly lessen the development burden of tool refactoring.

For example, Scatter is built on top of the GEMS modeling platform. When elements of the Scatter metamodel change, we use GEMS code generators to regenerate the underlying graphical editors for Eclipse and automate the refactoring. Figure 4 shows an example refinement of the Scatter DSL to remove the optional network concepts from the metamodel and the impact on a snippet of the underlying code. Performing the refactoring via code generation leads to the removal of approximately 4,700 lines of code when the network concepts are removed.

• **Metaconfigurable tools.** This is another technique that can lessen the impact of DSL and DSL composition changes on tooling [1]. These tools use a configuration file based on the metamodel of the DSL to automatically adjust themselves to metamodel changes. Rather than regenerating code to handle refactoring, the software is reconfigured for the new metamodel.

For example, GME is a metaconfigurable modeling tool that can dynamically reconfigure itself to display a graphical editor for different DSLs. When the PICML metamodel

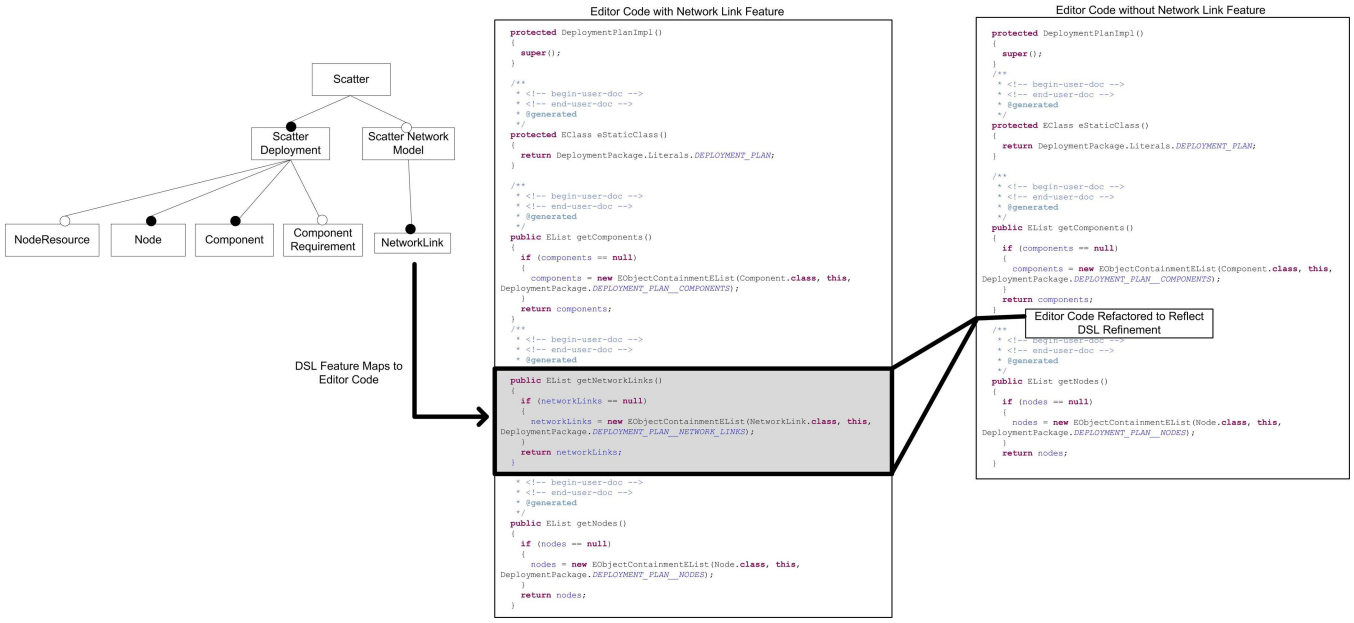

Figure 4. Feature Refinements in Scatter Changing Scatter Editor Code

is refined to remove event-related elements, a new metamodel configuration file is generated that tells GME how to configure itself to visualize the refined DSL. GEMS also takes this approach for specific parts of the graphical presentation of DSLs by using a stylesheet (based on the metamodel) to reconfigure the rendering of DSL elements.

4.4 Technique Analysis

Applying SPL reuse techniques to DSLs, DSL compositions, and tools incurs an initial overhead. A feature model must be built for each DSL and DSL composition following the processes outlined in Sections 4.1 and 4.2. Moreover, building metaconfigurable tools or code generators for tools takes time.

The payoff of applying these SPL techniques is when a DSL is used more than once. Since DSLs and tools are built with the intention of being reused multiple times, in many cases the extra effort to apply SPL techniques pay off. For example, the ability to regenerate the graphical editor infrastructure when the metamodel changes provides substantial savings in DSL tool refactoring effort for PICML, Scatter, CQML, and CUTS.

Some types of DSL tooling remain hard to modularize and reuse. For example, code generators and constraints are not often easily amenable to change. Our future work will investigate how to improve the reusability of code generators and constraints.

DSL reuse complexity metrics. From our experience adapting our DSLs and tools to new domains, we have distilled the following analysis of the number of refactoring steps as a function of the size of the metamodel and tool code base. Using standard ad-hoc DSL reuse, we observed:

1. $O(n)$ steps to analyze each DSL metamodel element for applicability to the new domain
2. $O(n)$ steps to delete unneeded DSL elements
3. $O(n^2)$ steps to analyze each deleted DSL element with each remaining element for dependencies violated by the deletion
4. $O(n)$ steps to delete any DSL elements that have unmet dependencies
5. $O(c)$ steps, where c is total lines of code, to refactor tooling

This yields a total initial refinement complexity of:

$$O(n + n + n^2 + n + c) = O(n + n^2 + c)$$

Further refinements for new domains also incur this full complexity.

With the feature model DSL refinement process, we observed:

1. $O(n)$ steps to analyze each DSL metamodel and create a corresponding feature
2. $O(n^2)$ steps to compare each DSL metamodel element against other elements to find interactions that must be added as constraints to the feature model

3. $O(n)$ steps to select DSL features for the new domain
4. $O(n)$ remove any DSL metamodel elements corresponding to unselected features
5. $O(c)$ steps, where c is total lines of code, to refactor tooling if code generation and metaconfiguration are not used

This analysis yields a total initial refinement complexity of:

$$O(n + n + n^2 + n) = O(n + n^2)$$

Further refinements for new domains, however, do not incur this full complexity. Future domains only require:

$$O(n + n + c) = O(n + c)$$

to select new features, refine the metamodel, and refactor tooling. If code generation or metaconfiguration are used to update tools, the manual code refactoring is eliminated:

$$O(n + n) = O(n)$$

For a single metamodel reuse through refinement, the feature model does not provide any complexity savings. When the metamodel is reused multiple times, however, a significant savings of:

$$O(n)\ vs.\ O(n + n^2 + c)$$

or

$$O(n + c)\ vs.\ O(n + n^2 + c)$$

is achieved.

5 Concluding Remarks

This article motivated the need for improving DSL reusability and has shown how SPL reuse techniques can be applied to DSL refinement and DSL composition adaptation to improve reusability. In particular, we have shown how feature modeling techniques can be used to document the semantic rules for modifying metamodels and DSL compositions. From our work applying SPL reuse techniques to DSLs, we have learned the following lessons:

- **Semantic constraints** can be enforced during DSL refinement by providing developers with a feature model of a DSL's metamodel.
- **Code generator modularization** is not easy to acheive with current SPL techniques. In future work, we plan to investigate how the reusability/modularity of code generators can be improved.
- **Applying model-driven engineering techniques to DSLs** greatly improves the speed at which new DSLs can be developed and existing DSLs can be refactored.

GEMS is a sub-project of Eclipse and is available from `www.eclipse.org/gmt/gems`. Scatter is an open-source DSL available from `www.sf.net/projects/gems`. PICML and CQML are also open-source DSLs available from `www.dre.vanderbilt.edu/cosmic`. Finally, CUTS is an open-source tool available from `www.dre.vanderbilt.edu/CUTS`.

References

[1] Ákos Lédeczi, Árpád Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[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. Beuche, H. Papajewski, and W. Schrder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333–352, 2004.

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

[5] 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.

[6] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented project line engineering. *IEEE Software*, 19(4):58–65, 2002.

[7] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[8] M. Voelter. A family of languages for architecture description. In *Proceedings of the OOPSLA Workshop on Domain-Specific Modeling*, pages 86–93, October 2008.

6 About the Authors

Dr. Jules White is a Sr. Research Scientist at Vanderbilt University's Institute for Software Integrated Systems. He received his BA in Computer Science from Brown University, his MS in Computer Science from Vanderbilt University, and his Ph.D. in Computer Science from Vanderbilt University. Dr. White's research focuses on applying a combination of model-driven engineering and constraint-based optimization techniques to the deployment and configuration of complex software systems. Dr. White is the project leader for the Generic Eclipse Modeling System (GEMS), an Eclipse Foundation project.

James H. Hill is a PhD candidate in the Department of Electrical Engineering and Computer Science at Vanderbilt University. James's research focuses on using model-driven engineering techniques to assist with locating design-time flaws related to quality-of-service (QoS) for component-based distributed real-time and embedded systems earlier in the software development lifecycle as opposed to integration time. James' research in this area has lead to the

creation of an tool called the Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS), which is an architecture-, language-, and platform-independent open-source system execution modeling tool for component-based 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. Jeff received a PhD in computer science from Vanderbilt University and both the BS and MS in computer science from West Virginia University. He is a member of the ACM and a Senior Member of the IEEE. Contact him at gray@cis.uab.edu.

Sumant Tambe is a PhD candidate of Electrical Engineering and Computer Science program at Vanderbilt University. His current research interests include model-driven engineering, its applications to the development of distributed real-time and embedded systems using QoS-enabled middleware, and improving productivity of model-driven development process.

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, optimization techniques, and empirical analyses of software frameworks and domain-specific modeling environments that facilitate the development of distributed real-time and embedded (DRE) middleware and applications. Dr. Schmidt has over fifteen years of experience leading the development of ACE, TAO, CIAO, and CoSMIC, which are open-source middleware frameworks and model-driven tools that implement patterns and product-line architectures for high-performance DRE systems.

Dr. Aniruddha S. Gokhale is an Assistant Professor of Computer Science and Engineering in the Dept. of Electrical Engineering and Computer Science at Vanderbilt University, Nashville, TN, USA. He received his BE (Computer Eng) from Pune University in 1989; MS (Computer Science) from Arizona State University, Tempe, AZ in 1992; and D.Sc (Computer Science) from Washington University, St. Louis, MO in 1998. Prior to joining Vanderbilt, he was a Member of Technical Staff at Bell Labs, Lucent Technologies in New Jersey. Dr.Gokhale is a member of IEEE and ACM. Dr. Gokhale's research combines model-driven engineering and middleware for distributed, real-time and embedded systems.