Improving Domain-Specific Language Reuse with Software Product Line Techniques

Jules White, James H. Hill, Sumant Tambe, Aniruddha S. Gokhale, and Douglas C. Schmidt, *Vanderbilt University*

Jeff Gray, *University of Alabama at Birmingham*

Techniques from software product lines can make DSLs, DSL compositions, and supporting tools more reusable by providing traceability of language concepts to DSL design.

Complex software systems, such as traffic management systems and shipboard computing environments, raise several concerns (such as performance, reliability, and fault tolerance) that developers must manage throughout the software life cycle. Domain-specific languages (DSLs)[1] have emerged as a powerful mechanism for capturing and reasoning about these diverse concerns. For each system concern, you can design a DSL to precisely capture key domain-level information while shielding developers and users from the technical solution's implementation-level details.

However, this narrow scope makes it hard to reuse a DSL for a new set of requirements (see the "Reusing and Adapting Domain-Specific Languages" sidebar).

We've developed two ways to improve reusability and decrease language reuse errors for DSLs and DSL compositions. First, a DSL can incorporate variability and codified configuration rules to enable its refinement for multiple domains. Second, we use software product line (SPL) techniques to codify the usage rules for a DSL composition's constituent DSLs, the concerns that the DSLs cover, and the variations in DSL usage. Codifying these concepts provides developers with a map of how to correctly modify and reuse DSLs and DSL compositions across projects.

Although previous research (see the "Related Research in Software Product Lines and Domain-Specific Languages" sidebar) provides a good starting point for addressing DSL reusability challenges, it has limitations. First, researchers have extensively studied SPL techniques in the context of software but not in the context of DSL design. So, we need new methodologies to codify how we can use SPL techniques to manage DSL refinement and composition adaptation. Although some researchers have applied SPL techniques to individual DSLs,[2] they haven't yet extrapolated generalized methodologies for applying these techniques to arbitrary DSLs. Moreover, they haven't applied SPL variability management techniques to DSL composition and reuse. In this article, we present a general methodology for using feature models to manage DSL and DSL composition reuse.

Four Related DSLs

Vanderbilt University's Institute for Software Integrated Systems has developed many DSLs and associated tools for a range of modeling concerns, such as component-based application design, deploy-

Reusing and Adapting Domain-Specific Languages

To create a domain-specific language (DSL), developers must carefully analyze the domain to design the language and produce the supporting tooling infrastructure for editing, compiling, running, and analyzing instances of the language. Not only are these DSL development activities complex, but developers might need to evolve a DSL over time to find the right abstractions. Each evolution can have from a small to massive impact on the tooling, depending on the infrastructure and type of changes. So, DSL-based development processes can incur relatively high overhead with respect to overall project time and effort.[1] One way to decrease this overhead is to amortize DSL development costs across projects (for example, by reusing existing DSL tooling infrastructure across development projects).

However, DSLs often focus on specific system concerns. Their narrow scope provides much of their power but can overly couple them to a particular group of assumptions, making it hard to reuse a DSL for a new set of requirements. Therefore, developers need a technique for systematically reusing DSLs and DSL compositions to simplify their adaptation to new requirements.

One solution is to apply software product lines. SPLs are a systematic reuse technique that supports

- building a family of software products such that you can customize variability for specific requirement sets,
- capturing how individual points of variability affect each other, and
- configuring product variants that meet a range of requirements and satisfy constraints governing variability point configuration.[2]

Developers use SPLs in domains where software development costs are high, safety and performance are critical, and redeveloping software from scratch is economically infeasible. SPLs have been successfully employed in domains such as avionics mission computing, automotive systems, and medical imaging systems.

References

1. M. Mernik, J. Heering, and A.M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, 2005, pp. 316–344.
2. P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.

ment and configuration of applications in distributed real-time and embedded (DRE) systems, and system execution modeling. We're frequently developing DSLs for new domains.

In this article, we focus on four DSLs we developed (see Figure 1):

- *PICML* (Platform-Independent Component Modeling Language) is for visually composing Corba Component Model (CCM) applications; it focuses on modeling the solution domain.
- *Scatter* is for modeling deployment of software components to hardware nodes in a distributed system; it focuses on the problem domain.
- *CQML* (Component Quality Modeling Language) is for specifying quality-of-service (QoS) constraints on systems; it focuses on the solution domain.
- *CUTS* (Component Utilization Test Suite) is for analyzing the performance of DRE system architectures; it focuses on the problem domain.

These DSLs are available at www.dre.vanderbilt.edu.

PICML, CQML, and CUTS are built atop the Generic Modeling Environment (GME; www.isis.vanderbilt.edu/Projects/gme). Scatter is built atop the Generic Eclipse Modeling System (GEMS; www.eclipse.org/gmt/gems), which is built atop the Eclipse Modeling Framework (EMF; www.eclipse.org/emf).

We've expended significant effort developing the four DSLs and their associated tooling. We've developed PICML over five years, and it continues to evolve. We've developed Scatter and CUTS over a period of four years. CQML is the youngest DSL, with roughly two years of development.

These DSLs form a closely related family. For example, we can build a CUTS model of the behavior of DRE system QoS and use it 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 map 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 letting developers manually specify software-to-hardware mappings but doesn't 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 (New Associative Object Model of Integration) project,[3] which involves using multiple DSLs to model software development for controlling traffic lights at intersections. Naomi uses PICML to model the software components, Scatter to derive suitable deployment topologies in Naomi, and CUTS to evaluate the traffic software's QoS.

After Naomi's development began, we addressed similar problems related to modeling deployment topologies and testing software performance in the context of the US Air Force Research Lab's Spruce (Systems and Software Producibility Collaboration and Experimentation Environment; www.

sprucecommunity.org) project. In Spruce, we modeled and tested the deployment of software to hardware in avionics systems. Owing to the similarity between the Naomi and Spruce requirements, we wanted to reuse as much of the original DSL composition as possible. Here, we use these four DSLs to illustrate the need for—and complexity of—reusing DSLs and DSL compositions for new requirements sets.

DSL Reuse Challenges

Tension exists between a DSL's domain specificity and reusability. The more precisely a DSL matches its domain, the easier and more accurately it can describe a solution. However, developing DSLs and their supporting infrastructure can be expensive, so reusability is desirable. Here, we explore the challenges of maintaining DSL specificity and accuracy while facilitating reuse.

DSL Refinement

Developing a robust DSL that accurately describes domain concepts and is intuitive for domain experts can be a long, iterative process. Developers create an initial prototype of the DSL; then, over a period of time, they refine the DSL concepts and notations by modeling existing and new systems. This refinement might take a substantial amount of time. 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, we originally developed PICML to model CCM applications. However, the need arose to model Enterprise JavaBeans (EJB) applications, which are similar to CCM (for example, they have similar component and home concepts) but don't share event source and sink features. Similarly, we originally created Scatter (a DSL aimed at specifying deployment constraints and topologies) to model deployment problems in the automotive domain. Since its original development, we needed to use it in other domains (such as flight avionics) that didn't share exactly the same types of deployment constraints.

To reduce DSL development cost, we could reuse PICML for EJB applications. However, this would expose EJB developers to certain details, such as event sources and sinks, that aren't relevant to their target domain. Reusing Scatter in the avionics domain would expose developers to crash survivability constraints that aren't relevant for planes. Such exposure to unnecessary details would eliminate many of the benefits of reusing a DSL.

Related Research in Software Product Lines and Domain-Specific Languages

Our reuse techniques for domain-specific languages (DSLs), which are based on software product lines (SPLs), build on previous research, particularly regarding feature models and DSL refinement.

Feature models codify a software product's points of variability and the rules governing the settings for each variability point.[1] A feature model is a tree-based structure in which each node represents a variability point or unit of functionality in the product. The tree's root represents the most generalized concept in the product, and successively deeper levels indicate software refinement. The parent-child relationships indicate configuration constraints that you must satisfy when choosing values for variability points.

Kyo Kang and his colleagues[1] and Danilo Beuche and his colleagues[2] have successfully applied feature modules to manage SPL variability in several domains. Feature models provide a solid foundation for improving reusability by codifying reuse rules. Moreover, other researchers have developed techniques to formally analyze feature models and identify configuration errors,[3] identify constraint inconsistencies,[4] and automate feature selection.[5]

DSL refinement 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. Markus Voelter has used model transformations to refine an architectural DSL.[6] His technique describes an architectural DSL's variability using a feature model. To refine the architectural DSL, developers select a set of architectural modeling features that should be in the refined language. On the basis of the feature selection for the new domain, model transformations automatically add or remove the corresponding metamodel elements.

References

1. K.C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering," *IEEE Software*, vol. 19, no. 4, 2002, pp. 58–65.
2. D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability Management with Feature Models," *Science of Computer Programming*, vol. 53, no. 3, 2004, pp. 333–352.
3. J. White et al., "Automated Diagnosis of Product-Line Configuration Errors in Feature Models," *Proc.12th Int'l Conf. Software Product Lines*, IEEE CS Press, 2008, pp. 225–234.
4. D. Batory, "Feature Models, Grammars, and Propositional Formulas," *Proc. 9th Int'l Conf. Software Product Lines*, LNCS 3714, Springer, 2005, pp. 7–20.
5. D. Benavides, P. Trinidad, and A. Ruiz-Cortes, "Automated Reasoning on Feature Models," *Proc. 17th Conf. Advanced Information Systems Eng.*, LNCS 3250, Springer, 2005, pp. 491–503.
6. M. Voelter, "A Family of Languages for Architecture Description," *Proc. OOPSLA Workshop Domain-Specific Modeling*, Springer, 2008, pp. 86–93, www.dsmforum.org/events/DSM08/Papers/15-Voelter.pdf.

Another reuse approach 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 sources and sinks don't apply directly to EJB. Removing the related notations from the PICML metamodel is nontrivial, however, because PICML has more than 700 interrelated metamodel elements. Eliminating these notations requires removing more than 30 other metamodel elements—for example, more

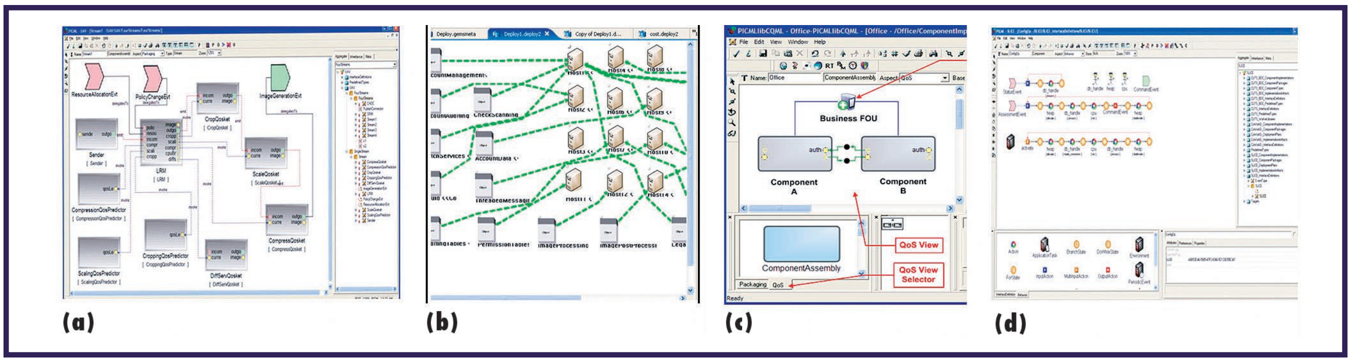

Figure 1. The (a) PICML (Platform-Independent Component Modeling Language), (b) Scatter, (c) CQML (Component Quality Modeling Language), and (d) CUTS (Component Utilization Test Suite) family of domain-specific languages.

than 15 elements are related to specifying properties of event channels that aren't needed if we remove event sources and sinks.

Reusable code libraries, aspect-oriented programming, and other language features can help modularize the software implementation. Similarly, various techniques, such as MetaEdit+'s fragments (www.metacase.com), GME's metamodel composition, the Atlas Model Management Architecture's model management techniques (www.sciences.univ-nantes.fr/lina/atl), or openArchitectureWare's (oAW) aspect-oriented features (www.openarchitectureware.org), 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 DSL language element.

A key problem in refining or modifying an existing DSL, regardless of the implementation tool used, is having traceability information for mapping

- concepts to the DSL metamodel or grammar (requirements to design) and
- the metamodel or grammar specification to its implementation in a particular tool, such as EMF (design to implementation).

Moreover, developers need additional information to ensure that the modification or refinement of the DSL doesn't violate design integrity, such as the completeness of the representation of concepts or the implementation correctness. 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.

Multi-DSL Composition

DSLs are often tightly aligned with a single, narrow slice of system concerns. So, to capture the concerns relevant to a system's requirements, multiple DSLs might be necessary. When devising a multi-DSL development process, developers must ensure that the DSLs adequately cover the concerns.

For example, in the Naomi project, developers must ensure that the DSL composition properly captures the traffic light system's real-time scheduling, deployment, and performance concerns. Naomi could potentially use several DSLs to capture the information related to the system's hardware nodes' capabilities. For instance, 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. Or, they could model the nodes through PICML. If developers must ensure that the nodes have sufficient resources to host the provided components, Scatter is a better choice. PICML wouldn't adequately cover the resource allocation concern. If developers needed real-time scheduling constraints, they could use Scatter or a combination of PICML and CQML.

The traffic light system has roughly a dozen concerns related just to the deployment of software components to hardware. These concerns are captured by multiple DSLs implemented on several tooling platforms. For example, developers must capture information regarding 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 isn't easy without traceability from the design concepts that must be covered to the individual DSLs providing 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 a DSL composition's design. For example, developers can refine PICML for EJB by removing event and deployment information. Removing the deployment-modeling capabilities from PICML, however, leaves CUTS without needed deployment information to generate experiments.

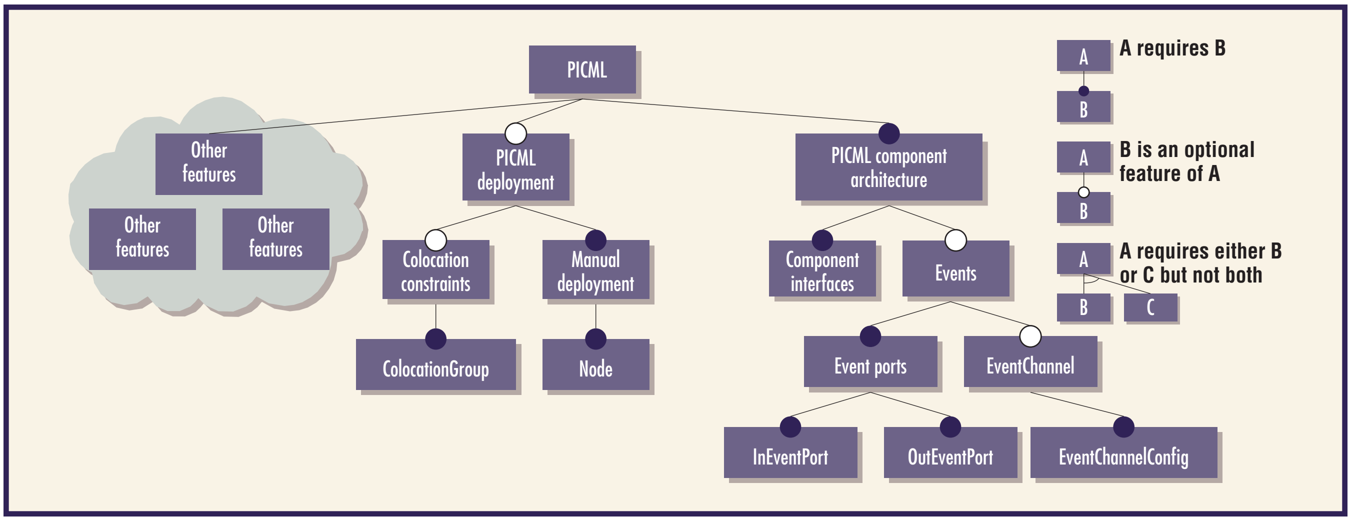

Figure 2. A PICML feature model snippet for event elements.

Developers must also ensure that refinement of the deployed DSLs provides the required concern coverage and adheres to any composition constraints.[4] Doing this is difficult without the explicit traceability we just discussed.

Applying SPL Configuration Techniques

Although DSLs are domain-specific, they 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 they can legally modify it, they can adapt the composition to new concerns.

The missing ingredient that produces the reuse challenges we mentioned earlier is that no model traces how concepts map to language design and captures the variability points and their interrelationships in DSL refinement, composition, and tools. Here, we show how to use SPL techniques to fill in this gap and increase DSL, DSL composition, and DSL tool chain reusability.

Managing Refinement via Feature Models

As we mentioned earlier, developers don't have concept-to-design traceability information or the rules for modifying a DSL's metamodel to ensure that they produce a semantically valid DSL refinement. One approach to solving this problem is to build a configurable DSL and use a feature model to document how concepts map to metamodel elements and what semantic dependencies exist between metamodel elements. The feature model describes why specific DSL language elements exist, which elements are semantically related, the semantic constraints for adding or removing elements, and the rules for determining what metamodel refinements are valid. The DSL elements are represented at the tool-independent level, and we can also define mappings to tool-specific implementations. Each refinement of the DSL's metamodel maps to a feature selection that developers can check for semantic validity.

Figure 2 shows a simplified feature model of the metamodel elements related to the PICML event elements we discussed earlier. 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 it reaches actual metamodel elements or tool-specific metamodel element mappings 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 PICML feature model 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 remove all the language elements and mappings to metamodel elements that appear as children beneath it. Moreover, if they want a more precise refinement, they can 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 language design and rules for correctly modifying the PICML metamodel's 700 language elements to refine concept coverage.

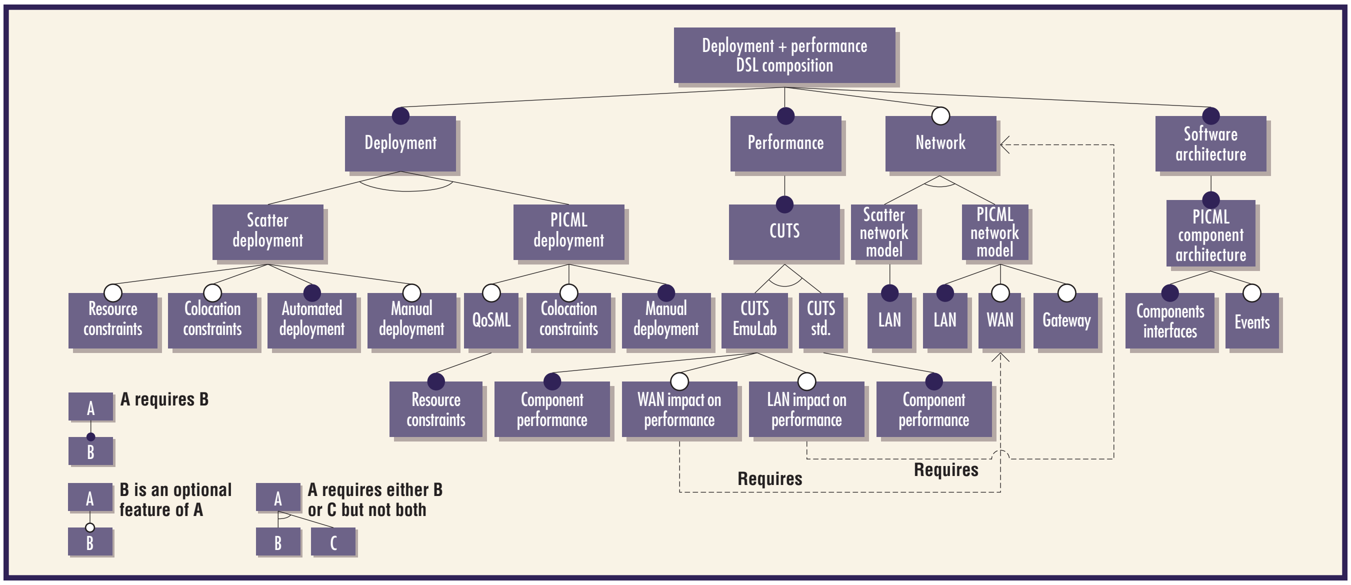

Figure 3. A feature model for the PICML/CUTS/Scatter/CQML DSL family. This figure shows how feature models can help capture the numerous DSL composition constraints that must be adhered to in order to reuse DSL compositions.

DSL Family Configuration with Feature Models

As we mentioned earlier, developers often don't know why a particular set of DSLs was composed and how the composition covered a set of concepts, owing to lack of traceability information. For example, it isn't clear how using PICML to describe deployment capabilities differs in concern coverage from using Scatter. Moreover, when developers must modify a DSL composition to cover a new concern (such as the Spruce aeronautics domain), they don't have a road map of the interactions between DSLs. This makes it hard to determine the features they can add or remove.

To address this issue, developers can use feature models to codify what concerns each member of a DSL composition covers, what dependencies or exclusions exist between DSLs, and how DSL refinements affect concern coverage. Figure 3 presents a feature model of the DSL composition covering the four DSLs. The DSL composition is the root feature. Beneath the root feature are features providing a general categorization (such as Deployment and Performance) of the DSLs in the composition. Beneath the categorization features are the actual DSL concepts that developers can use to capture the concern. For example, either Scatter deployment or PICML deployment can capture deployment information. The leaves beneath the DSL concepts are modeling capabilities the DSL provides—for example, Scatter provides Automated deployment, but PICML doesn't.

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

The Cost of DSL Reuse

Using SPL techniques to produce reusable DSLs has an associated cost. 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 upfront cost:

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

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 on the basis of the features selected for a variant of the DSL. C_3 is the overhead of creating code generation and 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) isn't high. C_2 is also typically low, owing to the excellent support for automatically generating graphical and textual editors for DSLs that tools such as GME, GEMS, GMF (Graphical Modeling Framework),[1] oAW's xText, and the Textual Concrete Syntax project

(TCS) provide.[2] For example, if developers remove the networking concepts from the Scatter metamodel, GEMS can automatically update the graphical editor, which results in the regeneration of roughly 4,700 lines of code. Markus Voelter has shown how to model transformations and use xText to automatically regenerate a textual editor with code completion and syntax highlighting as a DSL's feature selection changes.[2]

Our experience shows that the major difficulty in reusing DSLs lies in modularizing the code generation and analysis infrastructure, C_3. Some platforms, such as GME, have code generators 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, developers must perform a cost-benefit analysis. The cost of developing three separate DSLs without a reusable approach is

$$3 \times Cost(DSL).$$

With a reusable approach, the upfront cost is higher but subsequent DSLs cost less:

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

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

$Reuse(DSL_R)$, the price of refining and reusing an existing DSL, is typically dominated by the impact of C_3. For each scenario, developers must estimate whether the DSL will be reused enough to make the reduced price, $Reuse(DSL_R)$, overcome the initial overheads of C_1, C_2, and C_3. In our research, we've found numerous instances where the initial price of reusability paid off.

The need to reduce costs and time to market has motivated SPL-based reuse. We expect these needs will also motivate DSL-based reuse. With the growing complexity and functionality of DSLs and DSL tools, their development cost will also likely increase. In particular, model compositions that span multiple tools are hard to develop and maintain manually. These trends underscore the need for technologies that can manage and reuse DSL assets effectively.

Acknowledgments

We thank the Lockheed Martin Advanced Technologies Lab, the US Air Force Research Laboratory, and the US National Science Foundation (NSF-CAREER-0643725) for supporting this work.

About the Authors

Jules White is a research assistant professor in Vanderbilt University's Electrical Engineering and Computer Science Department. His research focuses on applying a combination of modeling and constraint-based optimization techniques to the deployment and configuration of complex software systems. White has a PhD in computer science from Vanderbilt University. Contact him at jules@dre.vanderbilt.edu.

James H. Hill is a research scientist in Vanderbilt University's Institute for Software Integrated Systems (ISIS). His research focuses on using model-based analysis to identify quality-of-service flaws in distributed real-time and embedded systems. Hill has a PhD in computer science from Vanderbilt University. Contact him at hillj@dre.vanderbilt.edu.

Jeff Gray is an associate professor in the Computer and Information Sciences Department at the University of Alabama at Birmingham, where he codirects research in the SoftCom Laboratory. His research interests include model-driven engineering, aspect orientation, code clones, and generative programming. Gray has a PhD in computer science from Vanderbilt University. Contact him at gray@cis.uab.edu.

Sumant Tambe is a PhD candidate in electrical engineering and computer science at Vanderbilt University. His research interests include model-driven engineering for distributed real-time and embedded systems. Tambe has an MSc in computer science from New Mexico State University. Contact him at sutambe@dre.vanderbilt.edu.

Aniruddha S. Gokhale is an assistant professor in Vanderbilt University's Department of Electrical Engineering and Computer Science. His research combines model-driven engineering and middleware for distributed real-time and embedded systems. Gokhale has a PhD in computer science from Washington University. Contact him at gokhale@dre.vanderbilt.edu.

Douglas C. Schmidt is a professor of computer science and an associate chair of Vanderbilt University's Computer Science and Engineering program. His research interests include patterns, domain-specific languages, and distributed real-time and embedded systems middleware. Schmidt has a PhD in computer science from the University of California, Irvine. Contact him at schmidt@dre.vanderbilt.edu.

References

1. M. Mernik, J. Heering, and A.M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, 2005, pp. 316–344.
2. M. Voelter, "A Family of Languages for Architecture Description," *Proc. OOPSLA Workshop Domain-Specific Modeling*, 2008, pp. 86–93; www.dsmforum.org/events/DSM08/Papers/15-Voelter.pdf.
3. T. Denton et al., "Naomi: An Experimental Platform for Multi-Modeling," *Model Driven Engineering Languages and Systems*, LNCS 5301, Springer, 2008, pp. 143–157.
4. D. Batory, J.N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Trans. Software Eng.*, vol. 30, no. 6, 2004, pp. 355–371.