Domain-Specific Software Engineering

Barrett R. Bryant
University of Alabama at Birmingham
Computer and Information Sciences
Birmingham, AL 35294-1170, USA
bryant@cis.uab.edu

Jeff Gray
University of Alabama
Department of Computer Science
Tuscaloosa, AL 35487-0290, USA
gray@cs.ua.edu

Marjan Mernik
University of Maribor
Smetanova 17
SI-2000 Maribor, Slovenia
marjan.mernik@uni-mb.si

ABSTRACT
This paper projects that an important future direction in software engineering is *domain-specific software engineering* (DSE). From requirements specification to design, and then implementation, a tighter coupling between the description of a software system with its application domain has the potential to improve both the correctness and reliability of the software system, and also lead to greater opportunities for software automation. In this position paper, we explore the impact of this emerging paradigm on requirements specification, design modeling, and implementation, as well as challenge areas benefiting from the new paradigm.

Categories and Subject Descriptors
D.2.1 [**Software Engineering**]: Requirements/Specifications – *languages*.
D.3.2 [**Programming Languages**]: Language Classifications – *very high-level languages.*

General Terms
Design, Languages

Keywords
Domain-specific languages, domain-specific modeling, requirements specification

1. INTRODUCTION
Programming languages and software engineering have throughout their short history moved from machine-oriented to human-oriented computing. This has been achieved through development of higher level abstractions, especially with respect to models that describe software systems and programmer-defined abstractions (e.g., object-oriented programming and modeling). One can argue that UML and Java are popular representations of modeling and programming languages, respectively. However, both UML and Java are general-purpose. We argue that the next level of abstractions will take place via domain-specific modeling (DSM) and domain-specific languages (DSLs). The move from general-purpose to domain-specific representation has the potential to greatly impact the field of software engineering by allowing domain experts and end-users (who are not software engineers and do not understand traditional programming languages) to describe their computational needs in a representation that is familiar to them (i.e., based on domain abstractions and notations). Domain experts, end-users and software engineers are already beginning to use domain-specific models and languages for describing solutions to their problem tasks. Our position for this paper is grounded in the belief that this trend toward domain-specificity will continue, but the current state-of-the-art has many challenges that must be addressed.

In this paper, we ask the following questions based upon this premise:

1. Can domain-specific approaches augment formal methods to improve automated processing of requirements specification, early validation of requirements, and higher reliability of completed software products?
2. What is needed to bring meta-configurable domain-specific modeling environments closer to the maturity level of traditional integrated development environments (IDEs) of general-purpose programming languages?
3. If non-software engineers, such as domain experts and end-users, are able to develop parts of a software system, what kind of tool support is needed to ensure quality and reliability of such systems?
4. What specific areas of software engineering might be impacted by this paradigm?

We explore each of these topics in the following sections.

2. DOMAIN-SPECIFIC REQUIREMENTS LANGUAGES
It is well-known that requirements engineering cannot be conducted effectively without domain engineering [1]. However, Bjørner [1] has also indicated that the detailed engineering of any significant domain is a "grand challenge" problem that may take many years to resolve. While expecting the results of such an undertaking to have significant impact, we propose that the requirements engineering process may be taken a step further in the interim (i.e., requirements specification should be carried out in a domain-specific manner).

Domain-specific requirements specification requires that there be a framework for expressing domain entities at the specification level; namely, in the form of domain-specific requirements languages (DSRLs). Such languages would allow requirements to be specified in terms of the application domain abstractions. We believe that such an approach can enhance existing requirements specification languages by providing appropriate domain-level abstractions for the systems to be built and their inherent requirements, such as security (e.g., RedSeeds [11] provides domain abstractions expressed in natural language, but it is hard to reason about such abstractions or incorporate them into formal

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

specifications). Providing requirements specification in terms of domain abstractions will also make such specifications easier for domain experts who are not software engineers to validate, because the specification will be expressed in terms of concepts which they understand. Software engineers may then concentrate on the formal specifications needed to model the appropriate domain behavior. This will increase the possibility of automated tools to produce software artifacts from the specification and improve the reliability of these artifacts.

The research issues that arise from this approach are:

1. How can domain-specific abstractions be integrated into existing requirements specifications methods so that such specifications can still be reasoned about (e.g., model checking) in the absence of a formal domain model?
2. Can this approach improve research in domain engineering and help lead toward a solution of Bjørner's "grand challenge" problem?
3. What properties are required of domains at this level to allow procession to other aspects of the software lifecycle?

3. DOMAIN-SPECIFIC MODELING LANGUAGES

Over the past decade, UML has been the standard notation for modeling software systems. As a general-purpose modeling notation, UML provides multiple languages (e.g., object models and interaction models) that can be used to specify an application from different perspectives. Although UML is useful for describing software architecture, it is often not the best for end-users desiring a more familiar notation. Furthermore, the large size of the language can also make UML challenging for traditional software practitioners [12].

The emergence of domain-specific modeling (DSM) [6][12] has challenged the notion of general-purpose modeling (as done with the UML). DSM has enabled both end-users and software developers in describing the key characteristics of a system from the perspective of the problem space, without getting overwhelmed by the accidental complexities of the solution space. By providing a notation that is often visual and graphical in nature, while also matching the abstractions of the domain, the essence of the problem can be captured in a way that removes the coupling with implementation concerns. The idea of model-driven engineering (MDE) has championed the idea of modeling with high-level abstractions and then generating other artifacts needed further down in the software lifecycle [19] (e.g., source code, test cases, or simulation scripts).

In DSM, a metamodel is used to define the essence of a specific modeling language, including the syntax and visualization provided by the language. A metamodeling tool can interpret the metamodel to provide a domain-specific modeling environment. Model transformations are used to translate a source model into some other form (e.g., a more refined model, or generated source code). Despite a number of successful industrial applications of DSM [12], there are still many issues that need to be resolved:

1. The maturity level of most metamodeling tools is at a level that is comparable to programming environments from the 1960s. Capabilities that are common in modern IDEs for programming languages (e.g., version control, testing, visualizing differences among different models) are still in the research stage of DSM. What seems to be a simple concept - model comparison/differencing - is at the heart of current research to improve the tool support available in metamodeling tools. As also mentioned in Section 4, the ability to generate supporting tools from language descriptions could be a useful way to improve the general maturity of modeling tools.
2. Most metamodels only capture the syntax and structural rules of a modeling language. The semantics and behavioral description is often delegated to the model transformation or model compiler. This is similar to defining a programming language by referring a user to "what the compiler says" rather than a formal definition of a language. A current research area in DSM is focused on new techniques for defining modeling languages based on new translation techniques that borrow from the experiences of defining programming languages. The maturity and future success of DSM depends on the ability to precisely define the meaning of a modeling language in order to support reasoning and automated generation of support tools.
3. Like all software artifacts, models also evolve over time. Within the context of DSM and MDE, evolution has several challenging issues: a) as the metamodel evolves, those model instances that depend on the metamodel must also co-evolve [19], b) as models evolve, the underlying legacy source code and other artifacts depending on the model must also evolve in order to maintain the causal connection with the models.

4. DOMAIN-SPECIFIC PROGRAMMING LANGUAGES

Domain-specific languages (DSLs) are languages tailored to a specific application domain [13]. They offer substantial gains in expressiveness and ease of use compared with general-purpose languages in their domain of application. Due to the narrow domain, unique possibilities for domain-specific optimization and verification are indeed feasible. Among other advantages of DSLs are enhanced reuse, productivity and software quality [4][15]. The intentions of domain-specific languages are numerous. On one hand, DSLs are powerful tools for software engineers and professional programmers for raising software productivity. While on the other hand, DSLs also enable end-user programming. DSLs assist programmers and end-users to write more concise, descriptive, and platform-independent programs [10][15][20]. This is made possible because the domain knowledge is specified at an appropriate level of abstraction, which is independent of the implementation platform. The amount of written software continues to be overwhelming (e.g., maintainers of the Linux kernel "add 11,000 lines, remove 5500 lines, and modify 2200 lines every single day" [13]) such that software maintenance remains a prevalent activity in software engineering (e.g., it is estimated that there are 0.1 to 1 defects per 1,000 lines of code in open source projects).

It is clear that we need a paradigm shift in software development to manage the complexity of development and maintenance. The same system functionality must be achieved with less code, which is also often easier to validate and maintain. Modifications to domain-specific programs are easier to create and can be understood and validated by domain experts who do not know how to program in a general-purpose language. However, end-user programmers are more likely to introduce software errors than professional programmers because they lack software training and proper support tools [8][21]. Therefore, there is an

urgent need for quality assurance regarding end-user development. Despite the fact that DSLs have recently attracted more research interest, many problems still need to be solved before DSLs become fully integrated into software engineering practice and adopted by both mainstream developers and end-users. Some of the challenges of using DSLs are:

1. DSL development is hard, requiring both domain knowledge and language development expertise. Many current DSLs have been developed without proper domain analysis and there is an urgent need to (semi-)automate this process and make it more feasible for software engineers. Some future directions are: mining domain concepts from existing application code written in general-purpose languages, using other artifacts where domain analysis has been performed already and presented in different forms (e.g., ontologies), and grammatical inference (GI) [16]. Moreover, results from domain analysis must be well-integrated with the DSL design process. Tools like language design assistants may help.
2. Developing integrated development environments (IDEs) for DSLs from scratch is too costly. Such DSL IDEs should include features that are typical of general-purpose language environments (e.g., syntax-directed editors, debuggers, profilers, refactoring tools, and test engines). These indispensible tools for software development may be automatically built from DSL specifications.
3. What kind of tool support is needed to ensure quality and reliability of software developed by end-user programmers?

Solving these problems would open new horizons in end-user development and enable a new paradigm shift in software engineering. In this respect, DSLs offer much promise and we anticipate strong impact in the future from DSLs.

5. APPLICATIONS OF DSE

There are a number of areas that have the potential to benefit from the impact of domain-specific approaches. We examine four benefits of DSL adoption in this section.

5.1 Component-Based Software Engineering and Software Product Lines

Despite the existence of web services, service-oriented architecture and cloud computing, we are still far from able to build a large distributed software system from a collection of components. The reason for this is that such components do not inherently carry enough information in their deployment to facilitate their composition. Successful composition relies on two cross-cutting domains: application domain and technology domain. Application domain knowledge imparts what components would naturally compose with other components to build the application system. Technology domain knowledge provides the technical infrastructure on how the components should be composed, including generation of glue/wrapper code and Quality of Service parameters.

The impact of domain-specific software engineering on building such systems is that domain knowledge is an inherent part of software systems, including plug-and-play components. This domain knowledge may be used to facilitate composition as well as reasoning about the composition with respect to correctness, reliability and various other quality measures (e.g., security) [2].

Software product lines (SPLs) are focused on abstracting out the variability and commonality of a set of products in a certain domain. The importance of domain analysis, and subsequently domain modeling, is an important part of SPLs. More recently, domain-specific modeling has been explored as a technique to improve the alignment of SPLs and their domain abstractions [3]. As an example, Gray et al. [6] present an example based on an industrial case study that is focused on modeling a mobile phone product line. It is anticipated that DSM and DSLs will offer insight into improving the description of software product lines using specific notations aligned to domain concepts, rather than general-purpose concepts (as typified by the traditional feature models). Domain-specific approaches to software architecture are explored in [22].

5.2 Parallel/High-Performance Computing

The wide availability of parallel computers has not resulted in available best practices for building parallel software systems. Such systems tend to require a great deal of hand-coding with little regard for the types of abstractions that have proven so valuable in building other software systems. This is due to the mismatch between high levels of abstraction and the underlying high performance that is desired for these applications. To date, there are few high-level languages that are specifically designed for parallel/high-performance computing. Most languages used in practice have a “parallel extension” of an existing language, usually C or Fortran, not renowned for their abstraction. Such parallel extensions are by necessity low-level and are often very oriented toward the underlying parallel architecture (e.g., memory hierarchies, multi-core processors, etc.). At the same time, algorithms are programmed according to these underlying architectures and the modular structure to support adaptation and evolution is often lacking in such approaches.

We believe that domain-specific software engineering could revolutionize the way parallel/high-performance software systems are built, by allowing algorithms to be modeled and programmed in a more architecture-independent way with appropriate mappings to the underlying architecture. Because “domain” could be defined from both an application and a platform view, it would contain knowledge of how applications are tailored to specific platforms. Preliminary work in this area includes [5][18].

5.3 Security

Security of computer systems and engineering of secure systems have become of paramount importance. It has been recognized that security of a system must be engineered from the outset of requirements specification [14]. However, it remains unclear how to specify security and how to carry security requirements through to modeling and implementation. Furthermore, security may be specific to the application domain and platforms the system is to be deployed on. We expect that domain-specific software engineering would greatly improve the prospects for this and there has been initial experimentation with using DSLs to specify security [7].

5.4 Ultra-Large-Scale Software Intensive Systems (ULSSIS)

The characteristics of ULSSIS [17] are represented by thousands of platforms, continuous evolution, scaled-up validation, verification, and certification, policy-based modifications, human interactions, and orchestration and control. As indicated in [9], [17], DSLs are a promising technique for ULSSIS engineering.

6. CONCLUSIONS

Our position for this workshop is focused on the role that domain-specific software engineering plays with respect to requirements specification, modeling and implementation. We have identified four areas of software engineering that would benefit from this paradigm and open new directions in software engineering research. We expect this paradigm to impact development of software systems in numerous other application areas.

7. ACKNOWLEDGMENTS

We are grateful to the National Science Foundation for supporting work leading to these ideas through grants CCF-0811630 and CCF-0643725 (CAREER).

8. REFERENCES

[1] Bjørner, D. 2010. Domain engineering. In *Formal Methods; State of the Art and New Directions*, P. Boca, J. P. Bowen, and J. I. Siddiqi, Eds. Springer-Verlag, London, 1-41. DOI=http://dx.doi.org/10.1007/978-1-84882-736-3_1.

[2] Cao, F., Gray, J., and Bryant, B. R. 2009. Component-based software engineering. In *Wiley Encyclopedia of Computer Science and Computer Engineering*, B. Wah, Ed. John Wiley & Sons, Inc., Hoboken, NJ.

[3] Chastik, G. and McGregor, J. 2005. Integrating domain specific modeling into the production method of a software product line, In *Proceedings of the 5th OOPSLA Workshop on Domain-Specific Modeling* (San Diego, CA, Oct. 16-20, 2005). DSM '05. http://www.dsmforum.org/events/DSM05.

[4] van Deursen A., Klint P., and Visser J. 2000. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices* 35, 6 (June 2000), 26–36. DOI= http://doi.acm.org/10.1145/352029.352035.

[5] Di Pietro, D. A., Gratien, J., Häberlein, F., Michel, A., and Prud'homme, C. 2009. Basic concepts to design a DSL for parallel finite volume applications: extended abstract. In *Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing* (Genova, Italy, July 7, 2009). POOSC '09. ACM, New York, NY, 1-11. DOI= http://doi.acm.org/10.1145/1595655.1595658.

[6] Gray, J., Tolvanen, J.-P., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J. 2007. Domain-specific modeling. In *CRC Handbook on Dynamic System Modeling*, Paul Fishwick, Ed. CRC Press, Boca Raton, FL.

[7] Hamdi, H., Mosbah, M., and Bouhoula, A. 2007. A domain-specific language for securing distributed systems. In *Proceedings of the International Conference on Systems and Networks Communication* (Cap Esterel, France, Aug. 25-31, 2007). ICSNC '07. IEEE Computer Society, Los Alamitos, CA, 76. DOI= http://dx.doi.org/10.1109/ICSNC.2007.2.

[8] Harrison W. 2004. The dangers of end-user programming, *IEEE Software* 21, 4 (July/Aug. 2004), 5-7. DOI= http://dx.doi.org/10.1109/MS.2004.13.

[9] Heering, J. and Mernik, M. 2008. Domain-specific languages as key tools for ULSSIS engineering. In *Proceedings of the 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems* (Leipzig, Germany, May 10-11, 2008). ULSSIS '08. 1-2. DOI= http://doi.acm.org/10.1145/1370700.1370701.

[10] Hudak P. 1996. Building domain-specific embedded languages. *ACM Comput. Surv*. 28, 4 (Dec. 1996). DOI= http://doi.acm.org/10.1145/242224.242477.

[11] Kaindl, H. et al. 2007. *Requirements Specification Language Definition: Defining the ReDSeeDS Languages*. Technical Report. http://www.redseeds.eu.

[12] Kelly, S. and Tolvanen, J.-P. 2008. *Domain-Specific Modeling: Enabling Full-Code Generation*. Wiley-IEEE Computer Society Press, Hoboken, NJ.

[13] Kroah-Hartman, G. 2009. Interview with Greg Kroah-Hartman. *How Software is Built* (Blog), November 2009, http://howsoftwareisbuilt.com/2009/11/18/interview-with-greg-kroah-hartman-linux-kernel-devmaintainer.

[14] Mead, N.R. 2008. *Security Requirements Engineering*. Technical Report, Software Engineering Institute, Carnegie Mellon University.

[15] Mernik, M., Heering, J., and Sloane, A. M. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4 (Dec. 2005), 316–344. DOI= http://doi.acm.org/10.1145/1118890.1118892.

[16] Mernik, M., Hrnčič, D., Bryant, B. R., and Javed, F. 2010. Applications of GI in software engineering: DSL development. In *Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory*, C. Martín-Vide, Ed. Imperial College Press, London.

[17] Northrop, L., Feiler, P, Gabriel, R. P., Goodenough, J., Linger, R., Longstaff ,T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., and Wallnau, K. 2006. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Technical Report, Software Engineering Institute, Carnegie Mellon University, http://www.sei.cmu.edu/uls.

[18] de Oliveira Castro, P., Louise, S., and Barthou, S. 2010. A multidimensional array slicing DSL for stream programming. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems* (Krakow, Poland, Feb. 15-18, 2010). CISIS '10. IEEE Computer Society, Los Alamitos, CA, 913-918. DOI= http://doi.ieeecomputersociety.org/10.1109/CISIS.2010.135.

[19] Schmidt, D.C. 2006. Guest editor's introduction: Model-driven engineering. *IEEE Computer* 39, 2 (Feb. 2006), 25–31. DOI= http://dx.doi.org/10.1109/MC.2006.58.

[20] Sprinkle J., Mernik M., Tolvanen J.-P., and Spinellis D. 2009. What kinds of nails need a domain-specific hammer? *IEEE Software* 26, 4 (July/Aug. 2009), 15–18. DOI= http://dx.doi.org/10.1109/MS.2009.92.

[21] Sutcliffe, A. and Mehandjiev, N. 2004. End-User Development: Tools that Empower Users to Create their Own Software Solutions, *Commun. ACM* 47, 9 (Sept. 2004), 31-32. DOI= http://doi.acm.org/10.1145/1015864.1015883.

[22] Taylor, R. N., Medvidović, N. and Dashofy, E. M. (2008) Software Architecture: Foundations, Theory, and Practice. John Wiley & Sons, Inc., Hoboken, NJ.