Panel
DSLs: The Good, the Bad, and the Ugly[1]

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

Kathleen Fisher
AT&T Labs, Inc. - Research
San Jose, CA USA
kfisher@research.att.com

Charles Consel
University of Bordeaux and INRIA
Bordeaux, France
charles.consel@inria.fr

Gabor Karsai
Vanderbilt University
Inst. for Software Integrated Systems
Nashville, TN USA
gabor.karsai@vanderbilt.edu

Marjan Mernik
University of Maribor
Department of Computer Science
Maribor, Slovenia
marjan.mernik@uni-mb.si

Juha-Pekka Tolvanen
MetaCase
Jyväskylä, Finland
jpt@metacase.com

ABSTRACT

A resurging interest in domain-specific languages (DSLs) has identified the benefits to be realized from customized languages that provide a high-level of abstraction for specifying a problem concept in a particular domain. Although there has been much success and interest reported by industry practitioners and academic researchers, there is much more work that is needed to enable further adoption of DSLs.

The goal of this panel is to separate the hype from the true advantages that DSLs provide. The panel discussion will offer insight into the nature of DSL design, implementation, and application and summarize the collective experience of the panel in successful deployment of DSLs. As a counterpoint to the current benefits of DSLs, the panel will strive to provide a fair and balanced assessment of the current state of the art of DSLs by pointing to the existing limitations and future work that is needed to take the concept of DSLs to further heights.

The assembled panelists are experts in the research and practice of DSLs and represent diverse views and backgrounds. The panel is made up of industrial researchers, commercial tool vendors, and academic researchers. The panelists have different perspectives on the technical concerns of DSLs; e.g., half of the panelists are proponents of textual DSLs and the other half has experience in graphical notations representing visual languages.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Classification – *Very high-level languages.* D.2.6 [**Software Engineering**]: Programming Environments – *Programmer workbench.*

General Terms

Design, Languages

Keywords

Domain-specific languages, metamodeling, Grammarware

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

1. Jeff Gray (Moderator)

Biography: Jeff Gray is an Associate Professor in the Department of Computer and Information Sciences at the University of Alabama at Birmingham (UAB) where he co-directs the research in the Software Composition and Modeling (SoftCom) laboratory. His research interests are in aspect-oriented software development, model-driven engineering, and generative programming. He is a 2007 NSF CAREER award winner in the area of evolution of domain-specific models. He also was awarded an IBM Eclipse Innovation grant for research supporting testing tools for DSLs. Jeff is the 2008 Program co-Chair of the International Conference on Model Transformation (ICMT) and the 2009 Organizing Chair of AOSD.

Panel Position: For over three decades, DSLs have assisted programmers and end-users by improving productivity through automation of common tasks. DSLs allow a programmer to concisely state a problem using abstractions and notations that closely fit the needs of a specific domain. Early examples include the languages from the 1970s for the Unix 'make' tool and yacc.

Despite the benefits offered by DSLs, there are several limitations that hamper widespread adoption. In particular, the state of the art for DSL tools, in general, is several generations behind tool support provided by general-purpose languages like Java or C++. Many DSLs are missing even basic tools such as debuggers, testing engines, and profilers. The lack of tool support can lead to leaky abstractions and frustration on the part of the DSL user.

[1] The title for this panel was inspired by Brent Hailpern and Perri Tarr's introduction to a special issue on model-driven development in *IBM Systems Journal*, 45(3), 2006, pp. 451-461.

Some of the questions that I would like to explore with the panel include the following: What is missing in current DSL tooling that is needed to push adoption of DSLs more heavily in general practice? Are the problems purely technical, or are there social and political forces at play? In what domains do DSLs have the greatest success, and where do they fail? Where do graphical languages offer benefits over textual DSLs, and vice versa? How is evolution supported with respect to changes in the various definitions and tools?

2. Kathleen Fisher

Biography: Kathleen Fisher is a Principal Member of the Technical Staff at AT&T Labs Research, where she has worked since receiving her Ph.D. in Computer Science from Stanford University in 1996. Her early work on the foundations of object-oriented languages led to the design of the class mechanism in Moby. The main thrust of her recent work has been in DSLs to facilitate programming with massive amounts of ad hoc data. Kathleen is Chair of SIGPLAN, on the steering committee of CRA-W, an editor of the Journal of Functional Programming, and chair of the steering committee for the Commercial Uses of Functional Programming Workshop.

Panel Position: In my years at AT&T, I have helped to design and build two DSLs: Hancock, a procedural language that facilitates processing massive transaction streams to build customer profiles, and PADS, a declarative language for describing ad hoc data formats from which a number of supporting tools can be automatically generated. These two languages exhibit a number of the claimed benefits for DSLs. They raise the level of abstraction, making it easier for domain-experts with little coding experience to write correct and efficient programs quickly. Code in these languages is concise and readable, making it easier to read, understand, and maintain. PADS has the additional benefit of being declarative, giving the compiler the freedom to generate multiple artifacts from a single description: a parser, a pretty-printer, a statistical analyzer, a converter to XML, etc.

Although these advantages are real, there are also difficulties in working with DSLs. People are generally reluctant to learn new languages, tool support for DSLs is poor, and it can be difficult to get DSLs to interoperate with mainstream languages. These disadvantages mean that the motivation for a DSL has to be compelling for it to be successfully adopted. Challenges for the future of DSLs include improving education in programming languages so that people are more comfortable learning new languages, developing meta-tool infrastructure so that tools for DSLs can be created more easily, and improving facilities for language interoperability so programmers can shift freely between domain-appropriate languages.

3. Charles Consel

Biography: Charles Consel is a professor of Computer Science at University of Bordeaux and leads the Phoenix research group at INRIA. His research interests are in programming language semantics and implementation, software engineering and operating systems. His experience with DSLs spans over two decades, in the contexts of industry-sponsored projects and Ph.D. supervision. He has designed and implemented DSLs in a variety of areas, including device drivers, programmable routers, and stream processing. Building on this experience, he has worked towards defining methodologies to design DSLs, assist in their development, and assess their practicality. His latest DSL, named VisuCom, is dedicated to the creation of telephony services and has been successfully transferred to the INRIA spin-off Siderion Technologies.

Panel Position: The DSL approach has long been used with great success in both historical domains, such as telephony, and recent ones, such as Web application development. And yet, from software engineering to programming languages, there is a shared feeling that there is still much work to do to make the DSL approach successful.

Unlike General-purpose Programming Languages (GPLs) that target trained programmers, a DSL revolves around a domain: it originates from a domain and targets members of this domain. Thus, a successful DSL should be some kind of a *disappearing language*; that is, one that is blended with some domain process. Achieving such a goal critically relies on the domain analysis and the language design. In my experience, these two phases are time consuming, human intensive and high risk. How can these two phases be tooled? How much improvement can we expect?

A successful DSL is above all one that is being used. To achieve this goal, the designer may need to downgrade, simplify and customize a language. In doing so, DSL development contrasts with programming language research where generality, expressivity and power should characterize any new language. As a consequence, programming language experts may not be the perfect match for developing a DSL. Does this mean that, for a given domain, its members should be developing their own DSL? Or, should there be a new community of *language engineers* that bridge the gap between programming language experts and members of a domain?

Finally, in many respects, a DSL is often an over-simplified version of a GPL: customized syntactic constructs, simple semantics, and by-design verifiable properties. These key differences may raise concerns about a lack of tool support for DSL development. Yet, there are many program manipulation tools (parser generators, editors, IDEs) that can be easily customized for new languages, whether textual or graphical. Furthermore, for a large class of DSLs, compilation amounts to producing code over a domain-

specific programming framework, and enabling the use of high-level transformation tools. Lastly, properties can often be checked by generic verification tools. Then, what is missing to develop DSLs? Do we need to have an integrated environment for DSL development, orchestrating a library of tools? Should there be a new breed of compiler and verification generators matching the requirements of DSLs?

4. Gabor Karsai

Biography: Gabor Karsai is a Professor of Electrical Engineering and Computer Science at Vanderbilt University, and Senior Research Scientist at the Institute for Software-Integrated Systems. He has over twenty-five years of experience in software engineering. He conducts research in the model-based design and implementation of embedded systems, in programming tools for visual programming environments, in the theory and practice of model-integrated computing, and in resource management and scheduling systems. He has worked on several research projects in the recent past: on model-based integration of embedded systems whose resulting tools are being used in various embedded software development tool chains, on advanced scheduling and resource management algorithms, and on fault-adaptive control technology that has been transitioned into the aerospace industry.

Panel Position: DSLs are great because they could potentially increase the productivity of engineers. Every problem domain has a 'language' in which it is easy, natural, and self-evident to express problems (and possibly solutions) of the domain. Software designers are building custom abstractions in any case, and if there is an explicit form for using and applying those abstractions, and one does not have to continually transcribe those into an implementation language, then solutions require less effort. Such productivity increases have been observed since the days of Lisp macros (which could be used to define DSLs), and novel developments (e.g., the use of Simulink/Stateflow by control software engineers) just reinforce the point.

There are some downsides of DSLs, however. One is that if they are too easy to define, then there is a danger of their unmanaged and uncontrollable proliferation on a project. We need efficient and effective techniques for specifying and implementing DSLs rapidly, but in such a way that their users can understand their semantics. The generative programming (program generation) community has come up with very nice solutions (in a functional language context) for adding DSLs to a base language, but it is not clear how this carries over to more traditional languages. Using a multitude of DSLs on a project unavoidably causes problems with DSL to DSL integration (i.e., composition), as well as managing the DSL's evolution. That is, if the DSL changes how can we ensure that old 'code' written using the 'old' DSL remains (re-)usable with the new DSL?

5. Marjan Mernik

Biography: Marjan Mernik is an Associate Professor at the Faculty of Electrical Engineering and Computer Science, University of Maribor, where he leads the Programming Methodologies laboratory. He has worked on several research projects on programming languages, grammarware, and evolutionary computation. Recently, his research has focused on DSLs. He is the co-author of the paper entitled "When and how to develop domain-specific languages" (with Jan Heering and Tony Sloane, *ACM Computing Surveys*, 37(4), December 2005, pp. 316-344).

Panel Position: By providing notations and constructs tailored toward a particular application domain, DSLs offer substantial gains in expressiveness and ease of use compared with general-purpose languages for the domain in question, with corresponding gains in productivity and reduced maintenance costs. By reducing the amount of domain and software development expertise needed, DSLs open up their application to a larger group of software developers compared to general-purpose languages. These benefits have often been observed in practice and are supported by quantitative studies, although perhaps not as many as one would expect. The advantages of specialization are equally valid for programming, modeling, and specification languages. However, DSLs are not a panacea for all software engineering problems, but their application is currently unduly limited. Below are my top-10 reasons why DSLs are not used more in industry:

1. Cost-benefit analysis for a particular domain is hard to perform. Is it worth the effort to develop a DSL? Often, this decision is postponed or never answered.
2. Lack of proper tool support (e.g., debuggers, test engines, profilers). Such tools are costly to build. Just building a DSL compiler is not enough.
3. Inadequate DSL support by existing Integrated Development Environments (IDEs). Programmers want to work with existing IDEs that they are familiar with.
4. Poor interoperability with other languages. In industrial projects, multiple languages are the norm rather than an exception.
5. Personal/social barriers: "I know Java, why should I learn something else?"
6. Instability of design and implementation of DSLs (i.e., frequent changes to DSL definition).
7. Improper education to general developer community about DSL benefits (e.g., gains in productivity, expressiveness, better possibility for analysis, verification, optimization, parallelization, and transformation)
8. Lack of proper semantic definition of DSLs.
9. Poor documentation and training.
10. Limited knowledge and expertise on how to perform domain analysis and how the results of domain analysis can be used in DSL design and implementation.

6. Juha-Pekka Tolvanen

Biography: Juha-Pekka Tolvanen is the CEO of MetaCase. He has been involved in model-driven approaches, metamodeling, and domain-specific modeling languages and tools since 1991. He has acted as a consultant worldwide on modeling language and code generator development. Juha-Pekka has authored a book on Domain-Specific Modeling and written over sixty articles for software development magazines and conferences. He holds a Ph.D. in computer science and he is an adjunct professor at the University of Jyvaskyla, Finland.

Panel Position: I have seen many times in industrial cases how productivity has improved 500-1000% when companies have moved to Domain-Specific Modeling. In this panel I would like to discuss the key mechanisms that make domain-specific (graphical, textual, matrix) languages successful in practice. These include raising the level of abstraction away from implementation (rather than mixing levels with embedded DSLs), providing full code generation in one step from integrated languages (rather than having CIM-PIM-PSM mappings of MDA), defining languages incrementally, and focusing on one company situation only. The last is often the most crucial since trying to build languages and generators for everybody is far more difficult. In the modeling language area it has led to creation of languages like UML and SysML that are tied to the code level, do not raise the abstraction significantly, nor provide possibilities for adequate code generation.

Obstacles to DSLs are often related to tooling and the language design process. Many of the tools require man-months, if not man-years, to implement, test, and share the languages to developers. More importantly, tools often totally ignore language evolution: too often models made earlier can no longer be used when the metamodel changes. Because this is naturally unacceptable in industry use it is not surprising that most of the tools have not been applied on a large scale. The second issue deals with the language design process. Because most developers are creating their first language, it is relevant to define and test languages incrementally to reduce the risks, support evolution and prepare the organization to work at a higher level of abstraction. Recently, more industry cases and proven practices for language design process have been reported, which I will summarize at the panel.