Software ENgineering

Domain-specific languages in perspective

J. Heering, M. Mernik

REPORT SEN-E0702 SEPYEMBER 2007

ISSN 1386-369X

Domain-specific languages in perspective

ABSTRACT
Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose languages in their domain of application. Although the use of DSLs is by no means new, it is receiving increased attention in the context of model-driven engineering and development of parallel software for multicore processors. We discuss these trends from the perspective of the roles DSLs have traditionally played.

2000 Mathematics Subject Classification: 68N19
1998 ACM Computing Classification System: D.3.2; D.2.1; D.2.13
Keywords and Phrases: domain-specific language; model-driven engineering; domain-specific modeling; chip multiprocessing
Note: Submitted to IEEE Software

Domain-Specific Languages in Perspective

Jan Heering[1], Marjan Mernik[2]
[1] CWI Amsterdam, The Netherlands, Jan.Heering@cwi.nl
[2] University of Maribor, Slovenia, marjan.mernik@uni-mb.si

Abstract

Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose languages in their domain of application. Although the use of DSLs is by no means new, it is receiving increased attention in the context of model-driven engineering and development of parallel software for multicore processors. We discuss these trends from the perspective of the roles DSLs have traditionally played.

General-purpose versus domain-specific languages

There is a plethora of computer languages. Some of them, such as Java and C#, are general-purpose programming languages. These are large languages with many features used to write programs for a wide range of applications. They are the primary tools of the programmer. Not all general-purpose computer languages are programming languages. UML is a general-purpose modeling language for the abstract specification and documentation of many different kinds of software, while Z is a general-purpose formal specification language.

Generality is a mixed blessing. Broad applicability is paid for by suboptimal expressiveness in any particular application domain. This is where domain-specific languages (DSLs) step in. They sacrifice generality and trade it for expressiveness in a particular domain, obeying a kind of Boyle's law, where *expressiveness* $\times$ *domain size* = *constant.* Of course, neither expressiveness nor domain size can be measured very well, so this identity is approximate at best.

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

DSL	Application domain
BPEL	Business processes
(E)BNF	Syntax specification
DiSTiL	Container data structures
ESML	Embedded systems modeling
Excel	Spreadsheets
HTML	Hypertext web pages
Lex/Yacc	Scanner/parser generation
Make	Software building
Maple	Computer algebra
SQL	Database queries
VHDL	Hardware design

Table 1: Some representative domain-specific languages.

as one would expect. The advantages of specialization are equally valid for programming, modeling, and specification languages.

Some representative DSLs with their application domains are listed in Table 1. We give examples for two of them, VHDL and BNF, in the following sections.

VHDL

An outgrowth of the VHSIC (Very High Speed Integrated Circuits) initiative in the early 1980s, VHDL (VHSIC Hardware Description Language) is a standard DSL for describing digital circuit designs. It describes circuit structure in terms of submodules with inputs and outputs (entities with ports in VHDL terminology) and uses concurrent and sequential programming to specify circuit behavior. By offering an appropriate hardware-oriented vocabulary and constructs for using standard libraries and predefined packages, the language yields a substantial reduction in circuit design effort. By allowing circuit designs to be tested by simulation and verified by model checkers or other methods, it minimizes the need for expensive hardware prototyping. Finally, it serves as a basis for hardware synthesis. Widely used in industry and academia, it is among the most successful DSLs.

Figure 1 shows the VHDL specification of an auxiliary clock generator. It uses package `STD_LOGIC_1164` for the standard multivalue logic. The single entity `clock2` has an interface consisting of two input and three output ports. Its behavior is given in the body of the `architecture` declaration. `Process` declarations are used for sequentialization. Simulation output is shown in Figure 2.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.STD_LOGIC_ARITH.ALL,
    IEEE.STD_LOGIC_UNSIGNED.ALL;

entity clock2 is
    Port ( clk : in std_logic;
           res : in std_logic;
           pon, pon_des: out std_logic := '0';
           res_des : out std_logic := '1');
end clock2;

architecture Behavioral of clock2 is
   signal tmp : std_logic_vector(7 downto 0) := x"00";
   signal tmp1 : std_logic_vector(7 downto 0) := x"04";
   constant N : std_logic_vector(7 downto 0) := x"07";

begin
   process(clk,res)
   begin
      if res = '1' then
         tmp <= x"01";
      elsif (clk'event and clk='0') then
         if tmp <= N then
            tmp <= tmp+1;
         else
            tmp <= x"01";
         end if;
      end if;
   end process;

   with tmp select
      pon <= '0'when x"01",
             '1'when others;
   [ ... ]

end Behavioral;
```

Figure 1: Partial VHDL specification of auxiliary clock generator.

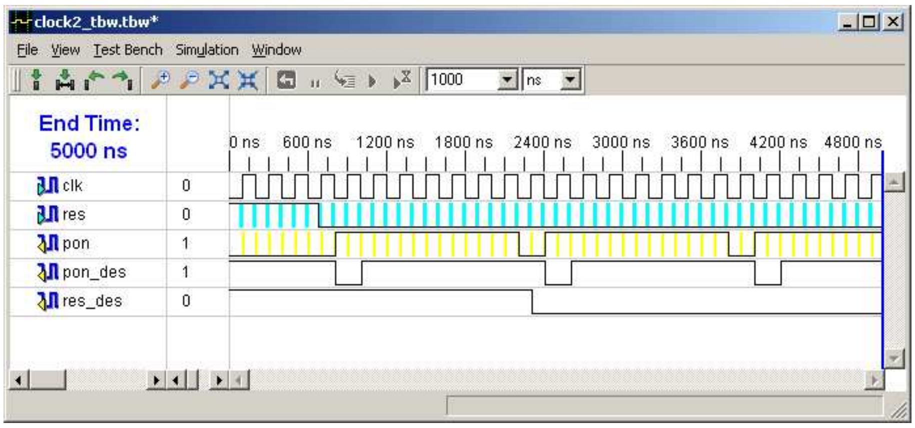
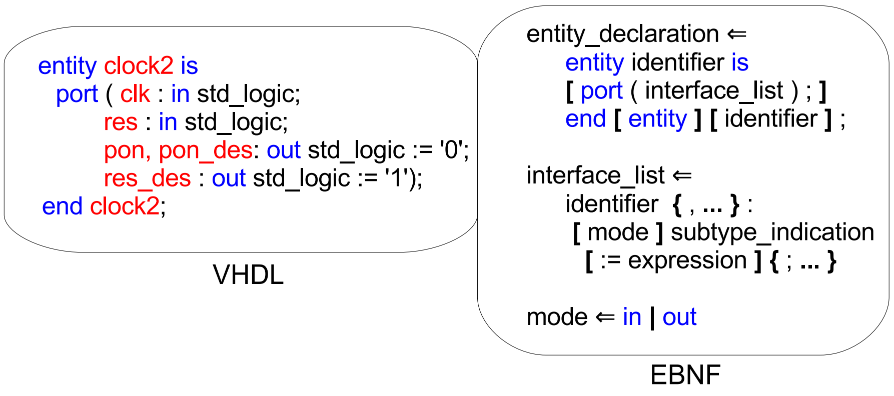

clock2_tbw.tbw*
File View Test Bench Simulation Window
1000 ns
End Time: 5000 ns
0 ns 600 ns 1200 ns 1800 ns 2400 ns 3000 ns 3600 ns 4200 ns 4800 ns
clk 0
res 0
pon 1
pon_des 1
res_des 0

Figure 2: Simulation of the circuit of Figure 1.



```
entity clock2 is
  port ( clk : in std_logic;
         res : in std_logic;
         pon, pon_des: out std_logic := '0';
         res_des : out std_logic := '1');
end clock2;
```

VHDL

```
entity_declaration ⇐
    entity identifier is
    [ port ( interface_list ) ; ]
    end [ entity ] [ identifier ] ;

interface_list ⇐
    identifier { , ... } :
     [ mode ] subtype_indication
      [ := expression ] { ; ... }

mode ⇐ in | out
```

EBNF

Figure 3: EBNF rules for a small VHDL fragment.

BNF

BNF is a DSL for specifying the syntax of computer languages. A BNF syntax for a language is a set of production rules that generate the sentences of the language and no others.

While not directly executable in a meaningful way, such a syntax can be used as input to a parser generator such as Yacc to obtain a parser for the language defined by it. In practice, the expressive power of BNF is inadequate and some form of Extended BNF is used, with virtually each language reference manual inventing its own slightly different EBNF dialect.

Figure 3 shows an EBNF syntax of a small VHDL fragment used in the previous example. A production rule defines the syntactic variable on the left-hand side of a left arrow in terms of the string of syntactic variables and language symbols on the right-hand side. Items enclosed in square brackets are optional and curly braces indicate lists.

Extending general-purpose languages

In combination with an application library, any general-purpose programming language can act as a DSL. The library's application programming interface (API) constitutes a domain-specific vocabulary of class, method, and function names that becomes available by object creation and method invocation to any program using the library. The general-purpose modeling language UML offers profiles and some other features for domain-specific extension. This being the case, why were DSLs developed in the first place? Simply because they can offer domain-specificity in better ways. Appropriate or established domain-specific notations are usually beyond the limited user-definable notation offered by general-purpose programming and modeling languages. Java has no user-definable operator notation. UML's notational extensibility is very limited. The new parallel language Fortress promises to be an exception. One of its key design goals is to facilitate embedding of (simple) domain-specific languages both notationally and semantically. The importance of good notations should not be underestimated as they are directly related to the productivity improvement associated with the use of DSLs.

Another problem is that appropriate domain-specific constructs and abstractions cannot always be mapped in a straightforward way on functions or objects that can be put in a library. Traversals and transaction processing are typical examples. By allowing implementation in terms of interpretation and code generation, DSLs transcend the limitations inherent in linking to application library code.

Furthermore, use of a DSL offers possibilities for analysis, verification, optimization, parallelization, and transformation (AVOPT) in terms of DSL constructs that would be much harder or unfeasible if a general-purpose language is used. In the general-purpose case the software artifacts (UML diagrams or source code) involved would be too complex for domain-specific patterns to be recognized in a reliable way.

Despite their limitations, domain-specific extensions of general-purpose languages are formidable competitors to DSLs. Even with improved DSL toolkits, such extensions will remain the most cost-effective solution in many cases.

Degrees of domain-specificity and executability

Domain-specificity is a matter of degree. Some consider Cobol to be a DSL for business applications, but others would argue this is pushing the notion of domain-specificity too far. It is natural to think of DSLs in terms of a scale such as the one shown in Figure 4, with highly specialized DSLs such as the syntactic specification language EBNF on the left and general-purpose languages such as Java on the right. On this scale, Cobol is somewhere between EBNF and Java, but much closer to the latter.

Many DSLs have a strongly declarative character allowing their users to concentrate on the "what" (problem space) rather than the "how" (solution space) of the software under construction. Accordingly, depending on the character of the DSL in question, the corresponding programs are often more properly called specifications, definitions, or descriptions. By raising the abstraction level of software development, DSLs move implementation closer to design. Using a DSL in the design phase to specify a system at a high level of abstraction is called domain-specific modeling (DSM) [1]. DSLs for modeling (DSMLs) potentially offer all the benefits of DSLs in the way of notation, automatic generation of software from models, and AVOPT. These observations give rise to a DSL executability scale similar to that shown in Figure 5. We will come back to modeling issues in the DSM section.

Two notions of application domain

DSLs derive their power from the opportunities for specialization provided by their application domain. We distinguish two notions of application domain. In the UML glossary, a domain is defined as *a field of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that field*. We call this a type A domain. Computer algebra is an an excellent example of a very rich type A domain. Maple, for example, not only supports mathematical notation but is actually a powerful system for doing numerical and symbolic mathematics, providing support for linear algebra, calculus, partial differential equations, Fourier series, and much more.

In software engineering a domain is often understood to be a *system family*, that is, a set of software systems that exhibit similar functionality. We call this a type B domain. Computer algebra, hardware design, and many other application domains of DSLs are not domains in this sense. By viewing a software system as encoded or crystallized knowledge about the system's functionality, a system family is seen to correspond to a field of knowledge. From this per-

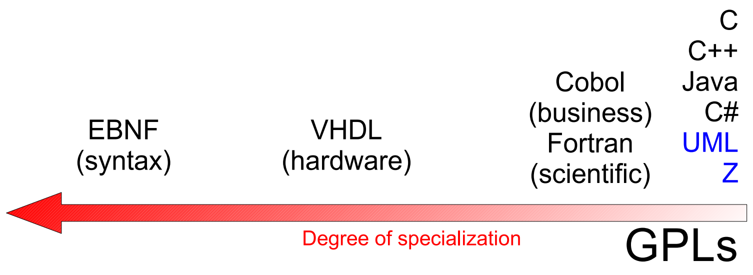


Figure 4: Domain-specificity scale.

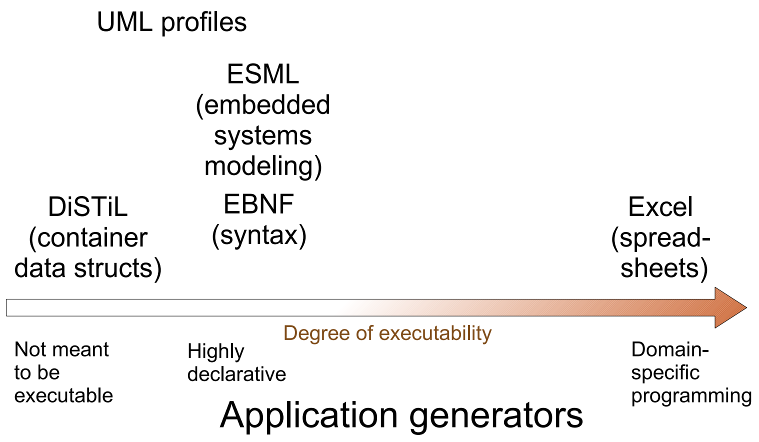


Figure 5: DSL executability scale.

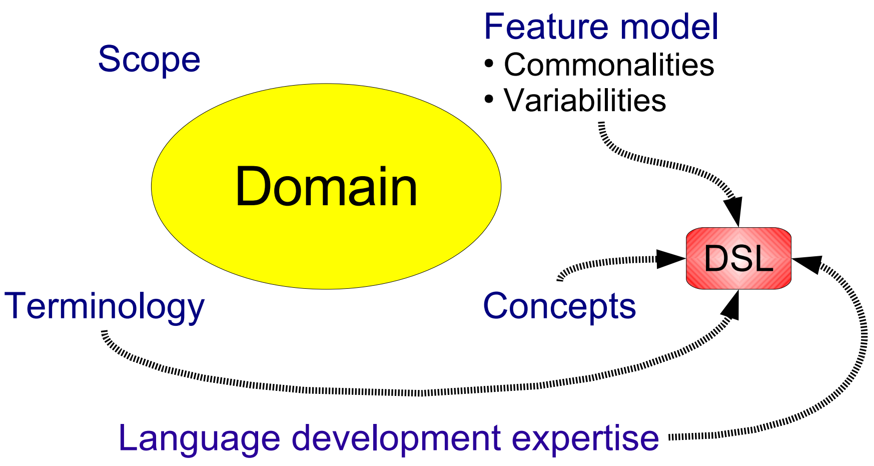


Figure 6: From formal domain analysis to DSL.

spective, a system family is an important special kind of application domain, that is, a type B domain is a special kind of type A domain. By virtue of their high degree of specialization, type B domains offer excellent opportunities for software reuse and formal domain analysis in terms of feature models (Figure 6).

Some would-be application domains are not sufficiently specialized to be classified as such. Scripting, for instance, is a way of using a language rather than an application domain. Scripting languages may be general-purpose or domain-specific. Similarly, modeling languages and parallel languages may be general-purpose or domain-specific.

The knowledge-oriented notion of domain is sufficiently general, fits the DSL context well, and subsumes the notion of system family as an important special case. It suggests the possibility of using techniques from knowledge engineering in DSL development. Knowledge capture, knowledge representation, and ontology development are potentially useful in the early stages of DSL development. This link is only beginning to be explored.

DSL development

Weighing the pros and cons of developing a new DSL in a specific case is rarely easy. DSL development requires both domain and language development expertise (Figure 6). Few people have both. Initially, it is often far from evident that a DSL might be useful or that developing a new one might be worthwhile. The concepts underlying a suitable DSL may emerge only after a lot of programming or modeling in a general-purpose language has been done. In such cases, DSL development may be a key step in software reengineering or software evolution.

The full picture is often surprisingly complex. Fourth-generation languages (4GLs) are a case in point. Despite often significant productivity advantages in database applications compared with Cobol [2], many 4GLs have not survived the transition to today's environments, having succumbed to a combination of insufficiently flexible, often proprietary, generator technology, eroding language expertise, and increasing environmental mismatch. A tentative conclusion might be that proprietary DSLs are particularly vulnerable.

To complicate matters further, DSL development techniques are more varied than those for general-purpose languages, requiring careful consideration of the factors involved. Choosing the right technique may make all the difference. Interoperability with other general-purpose or domain-specific languages used in a software project may be problematic, and, depending on the size of the user community, development of training material, language-specific tool support, standardization, and maintenance may become serious and time-consuming issues.

To aid the DSL developer, we provide a survey of guidelines, patterns, and toolkits for DSL development in [3]. The input to DSL toolkits is a description of various aspects of the DSL to be developed in terms of specialized metalanguages. Depending on the type of DSL, some important aspects are syntax, consistency checking, analysis, code generation, translation, transformation, and debugging. The metalanguages supported are often rule based. For instance, the syntax for textual DSLs is usually described in some kind of EBNF, while consistency checking of programs or specifications is often described in terms of attribute grammars or rewrite rules. DSMLs are typically described in UML/OCL rather than EBNF.

The main strength of current DSL toolkits lies in the implementation phase. Their main assets are the metalanguages they support, and in some cases a meta-environment to aid in constructing and debugging language descriptions, but they have little or no built-in knowledge of language concepts or language design. Furthermore, they offer no support for domain analysis.

Spurred on by an increasing need for DSL, and in particular DSML, tooling, DSL toolkits are becoming part of popular IDEs. Domain-Specific Language Tools [4] is a toolkit for Visual Studio aiming at graphical DSMLs. The Eclipse Modeling Project [5] provides components for abstract and graphical syntax development, model transformation, and code generation. SAFARI, an Eclipse-based meta-tooling framework for generating language-specific IDEs, is currently under development [6].

We now take a closer look at two promising new fields for DSLs.

Opportunity #1: Domain-specific modeling

DSM is a particular case of model-driven engineering (MDE) [7] advocating the use of DSMLs rather than general-purpose UML with its limited extensibility and overwhelming size. The potential of DSMLs is not limited to notation or specialized software generation, but includes the full range of AVOPT benefits mentioned before. For example, provided their semantics is sufficiently well-defined, DSMLs can act as a firm foundation for transforming, optimizing, comparing, and refactoring models. The Embedded Systems Modeling Language (ESML) is a good example [8]. Parts of ESML models are transformed into Finite State Processes (FSP) and model-checked for both generic properties such as deadlock freedom and domain-specific properties specified with FSP.

Like UML, DSMLs used for modeling often have a graphical syntax, which offers benefits when deep hierarchical structures are involved. Apart from this difference, there is little to distinguish DSMLs from other declarative DSLs, whether graphical or textual. Building textual DSLs for MDE by embedding in Ruby is discussed in [9].

The semantics of modeling languages is a moot point [10]. Usually it is given by model compilers or generators implemented in general-purpose programming languages. Such a semantics is inadequate from the viewpoint of language documentation and cannot serve as a basis for automatic generation of language-specific tools such as test engines or debuggers. Also, model processing tools may interpret a model differently if DSML semantics is underdefined. In practice, multiple domains may be involved, and the corresponding DSMLs and language-specific tools may have to be composed in some way. The Domain Workbench [11] supports definition, creation, editing, transformation, and integration of multiple domains during software creation.

The software factories methodology [12] advocates the development and use of DSMLs as essential to the construction of system families (type B domains). Where do these DSMLs come from? In many cases, they do not yet exist and will have to be developed. This makes DSML development a key element in the software factories approach. A recent book on domain-driven design [13] devotes less than two pages to DSLs, considering, among other drawbacks, their development and maintenance as beyond the skills of most development and maintenance teams. This is certainly a source of serious concern. The wider availability of toolkits may help to reduce the DSML development barrier, but the proper design of DSMLs (and DSLs in general) does not suddenly become a simple matter when a toolkit is used. How should the results of domain analysis be used in DSML design? Which DSML design rules should be followed? These are important questions that need to be addressed. Also, especially when the user group is relatively small, the maintenance and evolution of DSMLs should not be taken lightly.

Opportunity #2: Parallel software development

With the rise of multicore processors and chip multiprocessing (CMP), mainstream software development will have to adopt parallel programming techniques. In response, general-purpose programming languages will evolve into parallel languages, and more powerful parallelization tools will come into use. Nevertheless, it is to be expected that parallel programming will remain much harder than sequential programming. Because Moore's law is still in effect, the number of cores available to applications will grow steadily, taking over from clock frequency as the driver of computing performance. To keep up with an increasing number of cores, scalable parallelism is desirable, making software development even harder.

As a consequence, shielding software developers from having to bother about parallelism is becoming a major concern. This is where DSLs may offer a solution. Recall that the AVOPT benefits mentioned before include domain-specific parallelization as a key reason for using a DSL. Code generators for DSLs have a much better chance of achieving a high degree of parallelism without any special effort on the part of the developer than general-purpose code generators. As before, application libraries are an important special case.

There are two factors contributing to the potential of DSLs for harnessing parallelism. First, code for domain-specific constructs can be generated in terms of specially developed, perhaps highly intricate, parallel algorithms that are beyond the capabilities of automatic parallelization tools in the general-purpose case. DSLs can use such algorithms transparently in their implementation. Second, optimization (including further parallelization) of the generated code using domain-specific rules available to the code generator is more effective than general-purpose optimization.

There is no lack of potential domains. Perhaps computer games constitute the first domain (or set of domains) coming to mind. For instance, the real-time physical simulations needed by games for increased realism of moving clothing or ocean waves are natural candidates for scalable parallelism. Currently, such simulations would need supercomputers, which are not available to game software. Future multicore processors will be able to perform them on the desktop, an instance of a long-time trend of high-end applications becoming part of the mainstream. This trend can be used to extrapolate future desktop workloads partially from current high-end workloads. In addition, there will be many novel compute intensive desktop applications, such as video and music mining [14].

Outlook

Also called application-oriented or special-purpose languages, DSLs have been around for a long time. BNF, for instance, dates back to the late 1950s. They are firmly established in important roles, acting as enablers of the use and reuse of domain knowledge, as tools for large-scale software development and software generation, and as tools for end-user development and human-computer interaction. The final verdict on their wider application in MDE and in development of parallel software, which is the focus of current attention, is not yet in. The pros and cons are many and varied. With new challenges in ultra-large-scale systems on the horizon [15], the DSL field is in full swing and it will take some time for the dust to settle.

Acknowledgments

We would like to thank Jeff Gray, Tijs van der Storm, and Magiel Bruntink for valuable comments and discussions during the preparation of this article. Also, we thank Bojan Jarc and Tomaž Dogša for their assistance with the VHDL example.

References

[1] J. Gray, J.-P. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle. Domain-specific modeling. In P. A. Fishwick, editor, *CRC Handbook of Dynamic System Modeling*, chapter 7. CRC Press, 2007.

[2] R. L. Glass. The realities of software technology payoffs. *Communications of the ACM*, 42(2):74–79, February 1999.

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

[4] S. Cook, G. Jones, S. Kent, and A. Cameron Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison Wesley, 2007.

[5] Eclipse Modeling Project. `http://www.eclipse.org/modeling/`, 2007.

[6] P. Charles, J. Dolby, R. M. Fuhrer, S. M. Sutton Jr., and M. Vaziri. SAFARI: A meta-tooling framework for generating language-specific IDE's. In *Companion to the 21st ACM SIGPLAN conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA 2006)*, pages 722–723. ACM, 2006.

[7] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, February 2006.

[8] G. Karsai, S. Neema, A. Bakay, A. Ledeczi, F. Shi, and A. Gokhale. A model-based front-end to TAO/ACE. In *Proceedings of the 2nd Workshop on TAO*, 2002.

[9] J. S. Cuadrado and J. G. Molina. Building domain-specific languages for model-driven development. *IEEE Software*, 24(5):48–55, September/October 2007.

[10] D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics". *IEEE Computer*, 37(10):64–72, October 2004.

[11] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *Proceedings of the 21st ACM SIGPLAN conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA 2006)*, pages 451–463. ACM, 2006.

[12] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

[13] E. Evans. *Domain-Driven Design.* Addison-Wesley, 2004.

[14] P. Dubey. A Platform 2015 workload model. Technical report, Microprocessor Technology Lab, Intel Corporation, April 2007. `http://softwarecommunity.intel.com/articles/eng/1224.htm`.

[15] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute, Carnegie Mellon University, 2006. `http://www.sei.cmu.edu/uls/`.