

Improving CBS Tool Development with Technological Spaces

Jonathan Sprinkle
University of California, Berkeley
sprinkle@eecs.berkeley.edu

Abstract

The complexity of Computer Based Systems (CBSs) requires that multiple levels of abstraction be available to a designer in order to facilitate their formal specification. Generating final executable code from the model of the system is preferred to hand-coding the implementation, but this is seldom done in one step—usually there are several cascading transformations that eventually result in the executable system. This paper explains how the concept of the Technological Space (TS) can be used to define and describe the layers between cascading transformations, and the transformations themselves. TSs are also shown as a categorization that better distinguishes between a domain and the technology used to store information in a domain.

1. Introduction

The software crisis of the 1960s [11] led to the development of *software engineering*. However, software engineering has proved to be insufficient for the complex nature and enormous size of today’s Computer Based Systems (CBSs). Today’s software crisis is not due to a shortage of programmers, but the complex nature of a software solution needed to describe a complex system. This has led to the development of *systems engineering*, the methodology for creating system solutions, rather than just the solutions. The increasing popularity of domain-specific languages (DSLs) [15–17] and the UML’s Model Driven Architecture (MDA) [12] exemplifies this trend.

Software engineering addresses the architecture, specification, performance, and implementation of software. Systems engineering addresses these aspects from the perspective of the system, and seeks to generate the underlying software for the computer systems that the overall system contains. Note that systems engineering still requires that software engineering be performed: the importance is that it is not necessary to encode the system in terms of its underlying software—rather, it may be possible to provide some

mapping from the system to the software. Using models to describe software [9] can ease the software development process, but does nothing to create the system: this encoding into software is done by the software modeler, who must coordinate closely with the system expert.

To accurately map system behavior into software, system and software experts must work together (or—less likely—the software and system experts must be the same persons) in order to understand the idiosyncrasies of the system and its software implementation. It is here that the systems engineering science is found: the formalization of the mapping between the abstract arenas of system specification and software specification. This paper examines how to improve this mapping arena through the use and understanding of Technological Spaces (TSs).

Throughout the remainder of this paper we will explain how a TS can aid CBS designers and software engineers. First we provide some motivation as to why the TS is an important concept in the realm of domain-specific modeling. After this we provide the definition of a TS. Next, we give insight into how bridges between TSs can aid in artifact generation. After an example of how using the TS concepts can lead to a more formal definition of a DSME, we present some ideas for the future integration of TS concepts with CBS design, and future research requirements for TSs as a design concept.

2. Motivation

The discipline of software engineering provides insight into best practices and methodologies for software development. One aspect in which software engineering was always soft, though, was in the automation or generation of code from software requirements. Languages designed with formal requirements specifications in mind, such as Z [13], provided means in which to capture requirements, but eventually failed in two ways. 1) It was difficult (or impractical) to encode systems into the Z language, and 2) the formal methods did not naturally scale up to meet the demands of

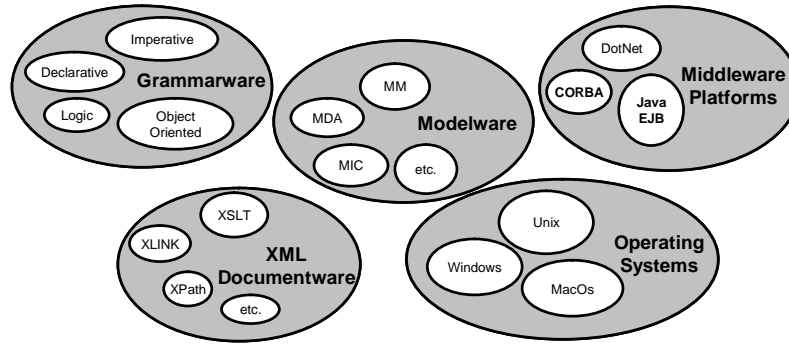


Figure 1. Technologies can be structured in similar families or composite structures.

large-scale systems. Although there exist numerous examples of the success of formal methods (even in large scale systems, e.g., [5]) each of these techniques either operates on existing software (verification only) or serves as a companion to the actual source code rather than a formal model of it. The formal (and efficient) specification of requirements, coupled with code generation, is quite simply the Holy Grail of software development.

It would be a mistake to say that software engineering failed in its objectives; more appropriately, software engineering failed to perform in arenas where it was never designed to compete. That is to say, software engineering is good at creating software, but bad at designing systems that require a software implementation. Even with the high-level modeling techniques of UML [10] it is not possible to create large-scale systems because UML was designed to model software, not systems.

A Domain-Specific Language (DSL) is one way in which to raise the level of abstraction to that of a system. Modeling using DSLs has proved to be an effective way to formally specify a system in a well-defined domain [15–17]. The structure and/or behavior of the system may be relatively easily specified by a domain expert, which immediately translates into increased productivity by the domain expert, fewer misunderstandings in system architecture by the software expert, and fewer mistakes in the low-level implementation (if that implementation is generated from the system model). When a DSL is included in a development environment designed to do formal modeling of a system, it is called a Domain-Specific Modeling Environment (DSME).

One reason for the success of DSMEs can be traced back to how the syntax and semantics of its DSL are defined [6]. The denotational semantics of the DSL are usually familiar to the domain expert that is expected to use the language. This is accomplished by crafting the concrete syntax such that it reflects domain concepts, and the abstract syntax such that creating programs (or models) is less like programming

and more like specifying the existence of the system. The operational semantics of the DSL are not normally important to the DSL programmer because they are considered to be an implementation detail rather than a design-critical element. When the operational semantics become design critical, they are absorbed as part of the DSL, meaning that the operational semantics become either an output choice (e.g., generating C++ or Java code from a CASE tool) or are governed by generation options (e.g., optimize for speed, or small size, or no optimizations for debugging).

The DSME is a good solution for the problem of system engineering, but the development of the DSME is not a trivial task. Even though there exist metaprogrammable tools [1–3] that provide excellent user interfaces and a quick turnaround time for developing rapid prototypes of a DSME syntax and denotational semantics, none of them have an equally efficient manner in which to develop the generation of the operational semantics. This is not a failing of the tools as much as it is a tribute to the difficulty of formally specifying the semantics of a programming language. During the development of the DSME crucial design decisions are made that determine what the DSME will do; will it generate assembly code for multiple platforms, or a low-level language with well-defined compilers for multiple platforms? Even when the DSL is created, however, it can sometimes be difficult to explain to someone what the DSME “does” to make systems specification easier. The description of how a DSME “does” what it does can be greatly enhanced through the use of Technological Spaces.

3. Technological Space

A *Technological Space (TS)* is a working context with a set of associated concepts, body of knowledge, tools, required skills, and goals (or possibilities). TSs are sometimes associated with a user community of shared experience, educational support, common literature set, and sometimes implementation [7]. A TS can be seen as an implementa-

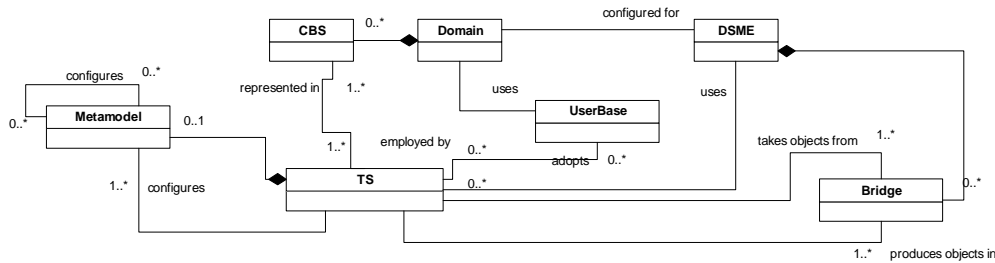


Figure 2. TSs and their relation to CBSs and domains

tion method of a domain. Just as category theory can be used to define set theory [8], the concepts behind TSs can be used to define the concepts of domain-specific modeling (or programming).

TSs are capable of interoperation, meaning that they *should* be able to exchange data, concepts, or an execution space with other members of the same TS. Of course, the amount of interoperability is an implementation detail for the TS members (e.g., although there is an operating system TS not all users of all OSs should be allowed to login to any OS). Members of a TS share an explicit (or implicit) meta-model or structure. This is required since TS members must, by definition, be operationally interoperable.

When a DSME is used to formally model a CBS, it is the domain expert who usually builds the system model. One beneficial side effect of this has already been mentioned: namely that the domain expert spends time specifying the system, rather than using e-mail, whiteboards, or other informal methods to describe the system to software engineers. The role of a TS in specifying a system is not in this final system specification, but in the specification of the behavior and role of the DSL and its artifact chain during DSME design and implementation. However, the TS classification serves as a categorical aid in describing the artifacts, behavior, and use cases of the DSL itself.

Figure 2 shows how TSs relate to domains and CBSs. The CBS is an existing system in a domain, but it is expressed using some technology—that is, its implementation is in some TS. A TS is configured to represent objects in some domain, and is adopted by a user base that uses that domain. This TS may use other TSs to create objects using common practices, and also has a metamodel that is used to configure the TS which implements the domain models.

There are many examples of TSs in software and CBS design and implementation. The TS itself is a particular slice of technology that proves useful in the description or implementation of something. A TS is not a domain, but the concept of domain-specific programming is a TS. A TS is not a language, but languages are found in a TS. Simply, the TS is used not to say *what* you are doing, but *how* you are doing it.

Some examples of TSs are XML, model management, graph transformations, grammars, textual programming, and assembly programming (see [7] for descriptions of these, and more). There are also composite sets of TSs that group together TSs that are similar in scope or purpose (see Figure 1). Although it is outside the scope of this paper, TSs exist not only in software, but hardware and other kinds of engineering, and even further beyond. In this paper, though, we are concerned mainly with how TSs can be used to describe the functionality of tools that design and implement CBSs.

4. Bridges between TSs

Using only one TS for all system design and implementation is infeasible. Moreover, some TSs are useful when solving certain problems but against other problems perform poorly. For example, [7] shows that the XML TS is good at solving the problem of transformations (using XSL) but when it comes to execution—a necessary performance metric for real-time systems—XML performs poorly. Once again, this is an example of the fact that TSs have certain niches where their use makes sense, and in others, it makes sense to utilize another TS.

The data that was used in one TS, though, should be available to another TS. *Bridges* are mechanisms for transforms between TSs. In modeling parlance, bridges are called model transformations, and in the textual programming TS, a C++ compiler/linker is a two-level bridge between textual programs and assembly code. Some bridges are bi-directional (e.g., properly implemented round-trips between code and models) but not all of them (e.g., hierarchy flattening).

Bridges can be used to take advantage of transformation technologies available in other TSs. Consider domain-specific models whose DSME has recently evolved [14]. The stored models (let us say we are using GME [2] and they are stored in the proprietary MGA format) will require some reformatting to be used in the new version of the DSME. There are several ways in which this might be

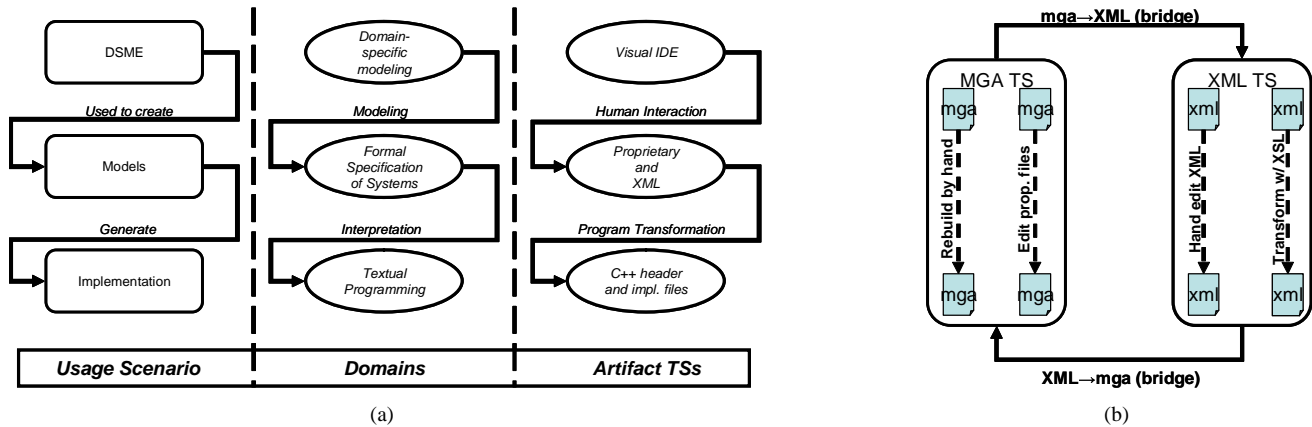


Figure 3. (a)Layers of a DSME model (left), along with the domains for each of those layers (center) and a TS in which the artifact of that layer exists (right). (b)Bridges between the mga and XML TSs allow the models to be transformed using XSL scripts.

done.

- Rebuild some (or all) of the models by hand
- Edit the model storage file by hand
- Use model transformations

Although the beginning and end artifacts are in the same TS in this example (proprietary MGA) the ways in which the artifact can be created are numerous. By using the XML serialization feature of GME, we can use the XSL language to transform the models into the appropriate format. In this case, we are using the MGA→XML bridge to take advantage of XSL, and the XML→MGA bridge to get the end artifact. Each of the above options is presented in graphical form in Figure 3(b). Within the mga TS, there are no well-defined ways to do the model transforms, but within the XML TS, XSL can modify the models using the concepts of the XML TS, making the transformation more reliable, and less subject to human error through data “hacking.”

5. Using TSs to Describe DSMEs

A DSME is canonically described as a tool with several layers of specification. Figure 3(a) shows the DSME, which is used to create formal models of a system, and from those formal models an implementation is generated. Usually omitted from the description of a DSME is that each layer of specification is domain-specific itself. In Figure 3(a) the domains are shown to suggest a formal CBS modeling domain that generates a C++ implementation.

Conspicuously present in Figure 3(a), but usually omitted (or not conceived) is the TS in which each layer of speci-

fication exists. During the development of the DSME, however, the TS for the generated artifacts is likely the most important high-level design decision. For instance, choosing the textual programming TS in this example shows that the final implementation is compiled, and is probably linked with some other libraries for final execution. Of course, the same end could have been achieved by generating the final assembly code from the models, rather than the intermediate C++ code.

Of course this is a design decision, and should be based on the final usage of the artifacts. If the objective is to distribute the files to many different users who will be using many different machine architectures, then using the assembly code TS results in unnecessary work that is actually counter productive. If the domain expert requires an executable immediately after interpreting the system model, though, then it makes more sense to absorb the compilation process into the DSME rather than require the domain expert to perform it.

Using this line of reasoning, the “how” of a DSME can be described by showing the TS transformations *enclosed* in the DSME, and by showing the TS interfaces it exposes. For example, the GME tool [2] allows for models to be read if they are in MGA format, or in an XML file that conforms to the MGA.dtd. These two TSs (the MGA and XML) are the *input* TSs for a DSME that uses GME as its interface. In our example, the *output* TS is textual programming.

Thinking even more abstractly about our example, it becomes clear that a DSME that uses GME as its interface exposes the GME TS as its input interface, and its output TS interface is that of a GME interpreter: the *model management* TS. By creating the GME interpreter, we absorb the model management composite TS into the DSME, and

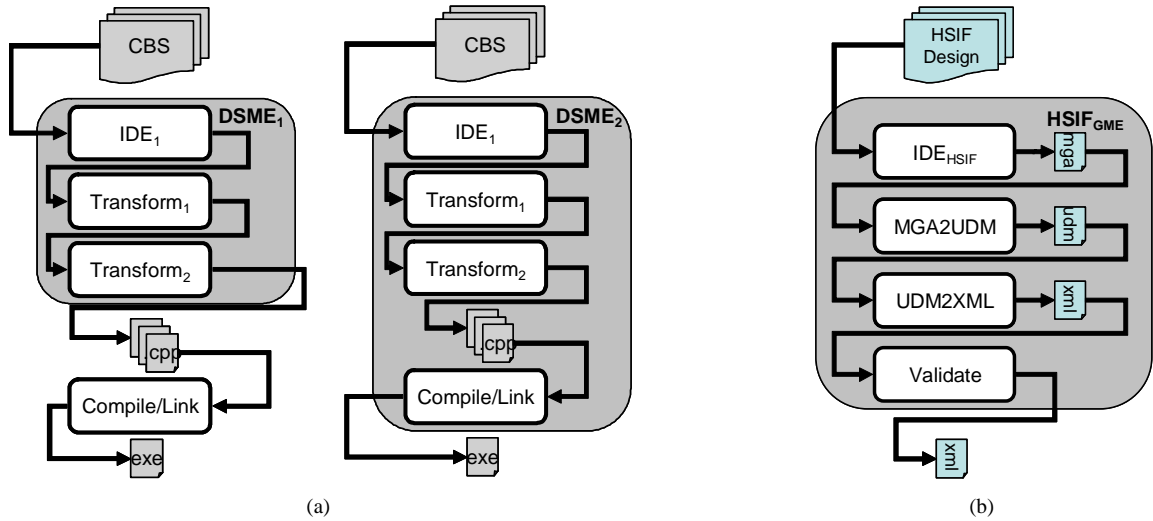


Figure 4. (a) The concept of TS absorption into a DSME. The exposed interfaces of DSME₁ and DSME₂ differ in their output. DSME₂ absorbs the compilation bridge, to create an output artifact in the machine code TS. DSME₁ produces an artifact in the textual programming TS. (b) Representation of the HSIF_{GME} DSME in the same notation as (a). Note that each translator in (b) produces an artifact in a TS that is shown in (a).

instead expose the TS that model management bridges to in this case: textual programming. This is shown in Figure 4(a). The decision of which bridges should be absorbed into the DSME should be based on the expertise level of the domain expert that will be using the tool. The more experienced the domain expert, the more likely that exposing a technical TS as an interface will be acceptable, which means that absorbing TS bridges that would expose a different interface may not be required.

6. Concrete Example: HSIF

An excellent example of using the notion of the TS to clarify the architecture of a CBS design tool is the Hybrid Systems Interchange Format (HSIF) design tool. The HSIF standard grew out of the DARPA MoBIES program as a common syntax and semantics with which to describe models of systems in the hybrid systems domain. The syntax of HSIF is formally described with an XML schema document, to which all HSIF models must conform in order to be considered valid. In addition, the semantics of the model of computation used by the HSIF standard, as well as any constraints on the representation of the system.

XML is a useful format for rapidly determining whether documents are syntactically valid, and there exists a bevy of freeware and professional toolsuites and binary executables capable of performing this validation. The verbose na-

$$\dot{x}_3 = x_1(1 - x_2/2)$$

(a)

```

- <DiffEquation>
- <AExpr> <MExpr>
- <MExprR mulOp="*">
- <MExpr> <ParExpr unOp="NOP">
- <Expr> <LExpr> <RExpr> <AExpr>
- <AExprR addOp="-">
- <AExpr> <MExpr> <MExprR mulOp="/">
- <MExpr>
  <Const unOp="NOP" value="2" />
  </MExpr>
</MExprR>
<VarRef _id="id1d" var="id3" unOp="NOP" />
</MExpr> </AExpr> </AExprR>
- <MExpr>
  <Const unOp="NOP" value="1" />
  </MExpr>
</AExpr> </RExpr> </LExpr> </Expr>
</ParExpr> </MExpr> </MExprR>
<VarRef _id="id20" var="id8" unOp="NOP" />
</MExpr> </AExpr>
<VarRef _id="id1b" var="id6" unOp="NOP" />
</DiffEquation>

```

(b)

Figure 5. (a) A simple differential equation (b) The same equation encoded in the HSIF.xsd expression syntax.

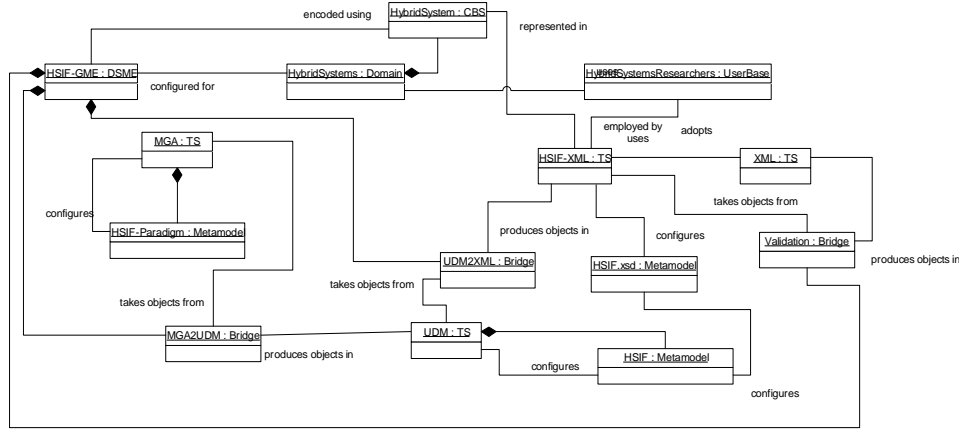


Figure 6. Object diagram that describes where the TS can be used to clarify the architecture of the HSIF CBS design tool.

ture of XML and the syntax of the HSIF schema make it relatively inconvenient to specify an HSIF model *by hand* when compared to a mathematical specification of the same model. For example, in order to ensure the guarantee of order of operations of a complicated differential equation a structured hierarchy of expressions is used in the HSIF schema to exactly encode the equation. As shown in Figure 5 the encoding of a simple equation can be cumbersome; the encoding of a complicated equation, frankly, is impractical.

The designers of HSIF chose the syntax to be precise, *not necessarily* to be convenient to enter in a text editor. They envisioned that many different modeling tools would be interchanging HSIF models via automatic translators that converted to/from HSIF and proprietary storage formats. However, there was still a need in research to create HSIF models for the sake of having them in HSIF (e.g., interface and regression testing), so a tool was designed for the hybrid systems domain. In order to demonstrate the role of the TS moniker in describing the behavior of CBSs, two alternative descriptions are given of the HSIF modeling tool—one using standard domain, artifact, metamodel, terms, and the other incorporating the TS term.

1. The HSIF modeling environment (named $HSIF_{GME}$ because of its configuration in the GME domain) is a DSME specially configured for the hybrid systems domain. Users can create hybrid systems models with $HSIF_{GME}$ and generate XML files of those HSIF models to interchange with other HSIF tools. $HSIF_{GME}$ generates artifacts (via UDM objects created according to another HSIF metamodel) in the XML domain (XML files), which conform to the HSIF.xsd schema document as specified by the HSIF standard. Incidentally, the UDM objects created dur-

ing the generation phase are guaranteed to produce valid $HSIF_{XML}$ files, because the UDM HSIF metamodel is used to generate the HSIF.xsd schema.

2. The HSIF modeling environment ($HSIF_{GME}$) is a DSME in the GME TS, and is configured for the hybrid systems domain. $HSIF_{GME}$ uses model generators to generate objects in the UDM TS, which provides a convenient interface to serialize those objects in the $HSIF_{XML}$ TS, which uses XML as a storage medium, and is configured by the HSIF.xsd schema. Incidentally, the metamodel of the $HSIF_{UDM}$ TS is actually used to generate the HSIF.xsd schema, which ensures the validity of the final $HSIF_{XML}$ artifact(s).

Now, the second description, which uses this notion of the TS, is not only shorter to write, but is more formal (and therefore less ambiguous) than the first. While formality should not be the sole metric for determining the effectiveness of a high-level behavioral description, it does permit a formal visual representation of the behavior, as shown in Figure 4(b).

This visual representation (shown as a UML object diagram) provides the unambiguous description of the role of the technologies in use for the HSIF, and how those technologies are related to the domain, and to the domain experts expected to use the modeling language. In the case of the $HSIF_{GME}$ DSME, generation of artifacts in the $HSIF_{XML}$ TS is absorbed into the DSME in order to hide complexity from the end user.

7. Recommendations and Conclusion

The recent increase in interest for domain-specific languages and the Model Driven Architecture shows that the specification of CBSs is moving away from the software layer and toward the system layer. Software designers are not obsolete: rather they must turn their attention to generative techniques and language transformations for the implementation of complex CBSs. The TS is a useful abstraction with which to view the software architecture and integration aspects of a design tool.

A DSME designer should carefully examine the artifact chain and transformation technologies used in the DSME from the perspective of the TS in which those artifacts and transformations reside. By doing so, the designer will not only have more confidence in the efficiency of the design, but will enable the CBS implementation to be performed in a piecewise manner, by rigidly defining the interfaces within the DSME and those exposed to the domain expert.

The TS is not yet a formal design concept but is quite useful in semi-formally explaining (or evaluating) the design of a DSME. DSMEs can be projected onto certain TSs in order to better understand their underlying semantics, and in order to ensure that their implementation is appropriate. Future research and input in this area is required, but the notion of the TS promises to be a useful one to DSME designers and domain experts.

8. Acknowledgment

Special thanks and acknowledgment are given to Jean Bézivin, whose keynote address at «UML» 2003 [4] and rich conversations contributed heavily to the content and ideas presented in this work.

References

- [1] *The Domain Modeling Environment (DoME)*. <http://www.htc.honeywell.com/dome/>.
- [2] *The Generic Modeling Environment*. <http://www.isis.vanderbilt.edu/>.
- [3] *MetaEdit+*. <http://www.metacase.com/>.
- [4] J. Bézivin. MDA™: From hype to hope, and reality. Keynote address, «UML»2003, October 2003.
- [5] E. Clark, J. Wing, and et al. Formal methods: State of the art and future directions. *ACM Comp. Surveys*, 28(4):626–643, December 1996.
- [6] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics, and all that stuff. part i: The basic stuff. Technical Report MCS00-16, Mathematics & Computer Science, Weizmann Institute Of Science, Rehovot, Isreal, 2000.
- [7] I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: An initial appraisal. In *International Federated Conferences (OTM'02), Industry Program*, 2002.
- [8] F. W. Lawvere and S. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, Cambridge, UK, 1997.
- [9] J. Ludwig. Models in software engineering – an introduction. *J. Software and Systems Modeling*, 2(1):5–14, March 2003.
- [10] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language*. Upper Saddle River, NJ, 2001.
- [11] S. Shapiro. Splitting the difference: The historical necessity of synthesis in software engineering. *IEEE Annals of the History of Computing*, 19(1):20–54, 1997.
- [12] R. Soley and T. O. Staff. *The Model Driven Architecture*. The Object Management Group, 2001.
- [13] J. M. Spivey. Understanding Z, a specification language and its formal semantics. *Tracts in Theoretical Computer Science*, 3, 1988. Cambridge University Press.
- [14] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *J. Visual Lang. and Computing*, April 2004.
- [15] J.-P. Tolvanen, J. Gray, and S. Kelly, editors. *ACM OOPSLA Workshop on Domain-Specific Visual Languages*, ISBN: 951-39-1056-3/ISSN: 0359-8470, October 2001. University Printing House, University of Jyväskylä, Finland.
- [16] J.-P. Tolvanen, J. Gray, and M. Rossi, editors. *Second ACM OOPSLA Workshop on Domain-Specific Visual Languages*, ISBN: 951-791-726-0/ISSN: 1235-5674, November 2002. Helsinki School of Economics Printing, Helsinki, Finland.
- [17] J.-P. Tolvanen, J. Gray, and M. Rossi, editors. *ACM OOPSLA Workshop on Domain-Specific Modeling*, ISBN: 951-39-1582-4/ISSN: 1239-291X, October 2003. University Printing House, University of Jyväskylä, Finland.