The Road Ahead for Architectural Languages

Patricia Lago, VU University Amsterdam

Ivano Malavolta, Gran Sasso Science Institute

Henry Muccini, Università dell'Aquila

Patrizio Pelliccione, Chalmers University of Technology and University of Gothenburg

Antony Tang, Swinburne University of Technology

// An exploration into the usability requirements of architectural languages in terms of language definition, language features, and tool support can lead to the next generation of architectural languages. //

"IF YOU THINK GOOD architecture is expensive, try bad architecture."[1] With this reflection, Brian Foote and Joseph Yode convey the message that developing an architecture is complex and expensive. One of the most complex and expensive tasks in software architecture design is the precise specification and communication of that architecture. A badly specified architecture design causes design and implementation flaws in a system and can create misunderstanding. In this article, we build on empirical studies to examine architectural languages (ALs)[2] and model-driven engineering (MDE)[3,4] as a means to improve architecture design.

ALs provide a way to describe a software system's architecture. According to the ISO/IEC/IEEE 42010-2011 Systems and Software Engineering—Architecture Description standard,[5] an AL is "any form of expression for use in architecture descriptions." So, an AL can be a formal language such as Acme, Darwin, or Architecture Analysis and Design Language; a UML-based notation; or any other way to describe a software architecture. A plethora of ALs has been proposed since the late 1980s, starting with box-and-line notations to describe systems as sets of components and connectors. In the late '90s, researchers remarked on their limited usefulness to provide automated analysis and implementation,[6] which led to a thread of research on formal ALs (for example, Wright, Cham, and Darwin) and resulted in tens of different languages. Some ALs provide features for specific application domains (such as automotive or avionics), whereas some UML-based languages and UML profiles are general purpose.[7]

However, our previous work involving 48 practitioners from 40 IT companies revealed a number of AL needs, many of which have a practical orientation (see the sidebar "What Industry Needs from Architectural Languages").[2] Building on those results and an in-depth analysis, this article defines, classifies, and clusters the requirements into a well-organized framework for designing and developing new ALs. In addition, to better clarify AL requirements, we explore MDE as a technology to help realize next-generation ALs.[8] MDE refers to the systematic use of models as first-class entities for describing specific aspects of a software system (such as data persistence, security policies, or software architecture) and the use of suitable engines for defining, analyzing, and manipulating those models

 0740-7459/15/$31.00

WHAT INDUSTRY NEEDS FROM ARCHITECTURAL LANGUAGES

To understand what organizations using architecture descriptions really need, we conducted an empirical study with 48 practitioners from 40 different IT companies in 15 countries.[1] The main purposes of the study were to understand which and how architectural languages (ALs) are used in the software industry, why some ALs aren't used in practice, and what AL features are lacking according to practitioners' needs.

We interviewed industrial experts who have used different types of ALs in production (including formal, semiformal, and informal ones). The study participants' software development experience ranged from two to 40 years, and averaged 19 years. Organizations participating in the study included both small to medium-size companies (52 percent) and large companies (48 percent). These organizations develop systems pertaining to both critical domains (for example, automotive, avionics, industrial automation, business information, and finance) and noncritical ones (such as media and entertainment, education, and project management).

We found that 86 percent of the respondents' organizations use UML or a UML profile, whereas approximately 9 percent use ad hoc or in-house languages, the remaining 5 percent of respondents declared to not use any modeling language for representing the software architecture of the system. Apart from ad hoc languages, the most-used ALs are Architecture Analysis and Design Language (around 16 percent), ArchiMate (around 11 percent), Rapide (around 7 percent), and EAST-ADL (around 4 percent). Moreover, only around 12 percent of respondents use architecture description languages (ADLs) exclusively, around 35 percent mix an ADL and UML, and around 41 percent use UML exclusively.

At the core of the study is a reflection about the needs and perceived limitations about ALs in industry. On one side, the most important identified needs are (in order) design (around 66 percent of respondents), communication support (around 36 percent), and analysis support (around 30 percent). Surprisingly, the least important needs are for code generation and deployment support (around 12 percent) and development process and methods support (6 respondents, 18 percent). On the other side, the most recurrent identified limitations are related to the insufficient expressiveness for non-functional properties (12 respondents, around 37 percent), insufficient communication support for nonarchitects (8 respondents, around 25 percent), and the lack of formality resulting in languages with no precise semantics, usually with no clear workflow on how to use them (around 18 percent).

Furthermore, some participants also declared that they have not adopted any AL for the following main reasons:

- formal ALs' need for specialized competencies with insufficient perceived return on investment,
- overspecification as well as the inability to model design decisions explicitly in the AL, and
- lack of integration in the software life cycle, lack of mature tools, and usability issues.

Interestingly, the study showed that software architects have two dual and complementary roles that must be appropriately reflected in ALs:

- Analyst and quality auditor. The architect's skills are mostly oriented to the disciplined development of the architectural design model. Tasks involve the design, analysis, and capturing of design decisions that have an impact on quality attributes such as cost, safety, evolution, performance, and security.[2]
- Negotiator and communicator. The architect's skills are mostly oriented to the communication of architectural decisions and knowledge to other stakeholders. The architect communicates and collaborates with project teams, customers, and developers by sharing and coediting system designs and the underlying knowledge[3] using various media ranging from verbal discussions, email, and wikis to the documenting of architecture specification documents.

Readers can refer to our study[1] for more details about our results and a thorough discussion about the current use of ALs in industry.

References

1. I. Malavolta et al., "What Industry Needs from Architectural Languages: A Survey," *IEEE Trans. Software Eng.*, vol. 39, no. 6, 2013, pp. 869–891.
2. P. Kruchten, "What Do Software Architects Really Do?," *J. Systems and Software*, vol. 81, no. 12, 2008, pp. 2413–2416.
3. V. Clerc, P. Lago, and H. van Vliet, "Architectural Knowledge Management Practices in Agile Global Software Development," *Proc. 1st Workshop Architecting Global Software Development Systems* (AGSE 11), IEEE CS, 2011, pp. 1–8.

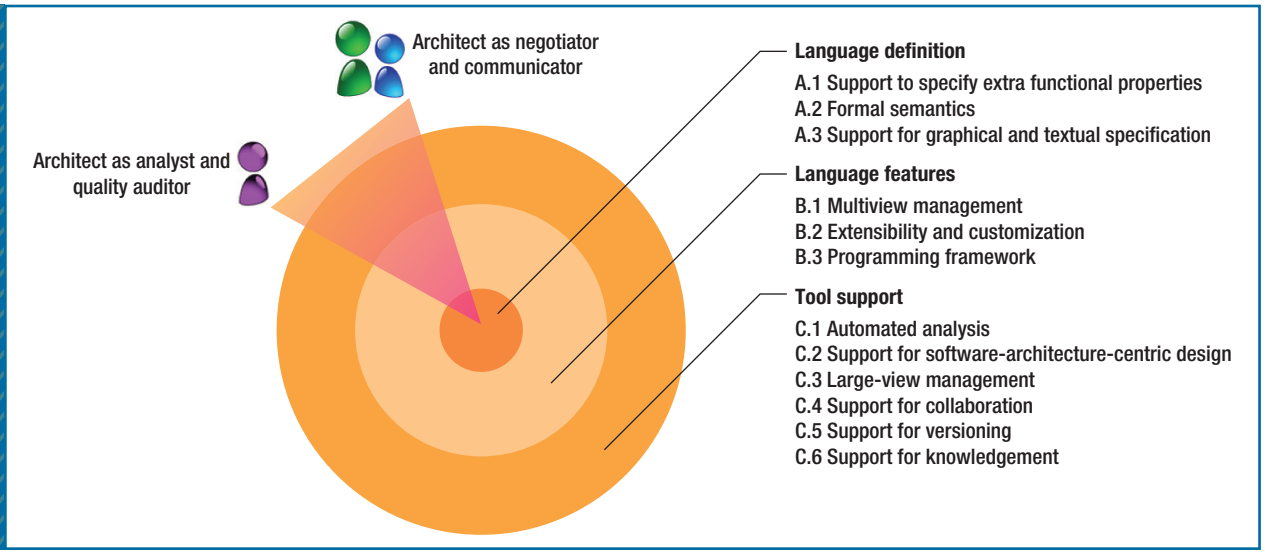

FIGURE 1. A framework of architectural-language requirements organized into three clusters: language definition, language features, and tool support.

throughout the system development life cycle. Such modeling activities include code generation, performance analysis, and so on. MDE has proven to be very effective,[3,4] and many well-established MDE techniques can satisfy AL requirements.

A Requirements Framework for Next-Generation ALs

To better understand how next-generation ALs will support various architecting activities, let's review a framework of AL requirements. A large number of requirements focusing on specific aspects of ALs, architecting activities, and methodologies emerged from our analysis.[2] To organize those requirements, we classified them into three clusters on the basis of the three standard elements a modeling language in software engineering has to consider:[9]

- language definition elements that make up the notation for modeling relevant concepts;
- language mechanisms that, built upon such concepts, offer mechanisms to change, refine, and organize the concepts in a certain context or perspective; and
- tool support that offers tools and applications for carrying out modeling activities for individual and collaborative modeling.

Figure 1 shows the requirements for next-generation ALs grouped into these three clusters.

Cluster A: Architectural Language Definition

The language definition cluster contains requirements about defining a language composed of abstract syntax, concrete syntaxes, and semantics. In this context, practitioners suggested many needs and concerns that we elaborate into three main requirements:

1. Support to specify nonfunctional properties. There's a growing need for non-functional analysis such as dataflow analysis, runtime dependency analysis, and analysis on performance, scalability, security, requirements, and change impact.
2. Formal semantics. Formal languages (such as process algebras, state charts, CSP (Communicating Sequential Processes), and π-calculus) give a precise and unambiguous semantics to the language. While practitioners consider formal semantics an obstacle to usability and dissemination of ALs, they also recognize it as an important enabler for analysis and other automatic tasks.
3. Support for graphical and textual specification. Practitioners report the need to use a combination of textual and graphical representations in the same project. For instance, graphical representations can be useful for knowledge sharing and discussion, whereas expert users might use textual representations for rapidly building a model.

These requirements underpin the needs for an AL language to express an architecture model.

Cluster B: Architectural Language Feature

The language mechanisms cluster contains three requirements that offer features to organize, change, and refine architecture descriptions:

1. Multiview management. There's an emerging need for multiview modeling, where each view delivers a different perspective on the architecture or addresses a different concern or stakeholder. In this context, ALs should be able to manage multiple views of the same architecture as well as maintain consistency across the various views.
2. Extensibility and customization. Practitioners need improved support for extending ALs to better express domain- and project-specific concepts, for specifying constraints, and for enabling additional analysis capabilities.
3. Programming framework. Practitioners need instruments—that is, suitably defined APIs—to programmatically access and operate on architecture descriptions. More specifically, a programming framework must expose facilities to manage, create, and modify different views of architecture descriptions in a coordinated way. These facilities will play a key role in the seamless integration of the software architecture description with all the other artifacts used and produced across the whole development process.

These requirements are the language features that software architects can use in different situations.

Cluster C: Architectural Tool Support

Building on language elements and mechanisms, an AL should provide tools to carry out individual and collaborative modeling activities. Our discussions with practitioners resulted in the definition of six main requirements:

1. Automated analysis. Practitioners need automated support for analyzing their systems, especially against non-functional properties. Automation must be able to mask the complexity of the analysis engine, thus reducing the demand for specific skills and competencies for performing these tasks.
2. Support for software architecture-centric (SA-centric) design. SAs should be used as high-level design blueprints during system development and later on for maintenance and reuse. Therefore, ALs should be integrated into development processes, specifically with system requirements, implementation, maintenance, and so on.
3. Large-view management. Architecture descriptions of complex systems can encompass several large, different, and interrelated views. In light of this, architectural information relevant for a specific stakeholder might be scattered across different views. Practitioners need new tools that create accessible architecture descriptions easily and pragmatically, despite this complexity and fragmentation.
4. Support for collaboration. Globalization of software development requires collaborative services across geographic areas. Collaborative services ought to support both synchronous and asynchronous collaboration. Synchronous collaboration tools enable real-time communication and collaboration in a "same time, different place" mode, but they require that all participants must be available at the same time. Asynchronous collaboration tools enable participants to connect at their own convenience and schedule, at the cost of possible delays of interaction.
5. Support for versioning. Versioning tools allow system stakeholders to maintain a repository of artifacts with special emphasis on keeping track of their various versions throughout the project duration. Because software architects produce and consume artifacts of a heterogeneous nature, scope, and interlocutor, AL tools should provide facilities to seamlessly version architectural artifacts in an easy, transparent, and homogeneous way.
6. Support for knowledge management. Typically, organizations rely on their employees to be proactive in capturing and promoting knowledge sharing across development sites. As knowledge

> An architectural language should provide tools to carry out individual and collaborative modeling activities.

is typically dispersed and ill organized, there are many challenges in sharing and retrieving. To overcome this issue, next-generation ALs ought to leverage knowledge-sharing tools such as wikis and semantic wikis to record and discuss architectural design decisions and their rationale. This requirement is strictly connected to the well-known problem of architectural knowledge vaporization, which leads to high maintenance costs.[10]

In order for AL to be workable, tools must be available to support these requirements. In order to build tools that are flexible enough to work in varying situations, we investigate the application of MDE.

Architects' Dual and Complementary Roles

Both the architecting activities presented in the "What Industry Needs" sidebar and the requirements for ALs we just described emphasize the need to further improve ALs in supporting the dual and complementary roles of software architects: analyst and quality auditor as well as negotiator and communicator. Whatever role architects play, their ALs should satisfy a combination of the requirements from the three clusters, regardless of the kind of system being developed, the type of involved organizations and people, and the various constraints and risks of the project being carried out.

A Technological Solution for Building Next-Generation ALs

MDE is a possible technological solution for successfully supporting the requirements of next-generation ALs. In MDE, architects use domain-specific modeling languages (DSMLs) to describe the system of interest. The concepts of a DSML—its first-class entities, relationships, and constraints—are defined by its metamodel. According to this, every model must conform to a specific metamodel, similar to how a program conforms to the grammar of its programming language. In MDE, it's common to have a set of transformation engines and generators that produce various types of artifacts. Practitioners can take advantage of transformation engines to obtain source code, alternative model descriptions, deployment configurations, inputs for analysis tools, and so on.

MDE Techniques in the Software Architecture Domain

An AL can be considered a DSML tailored to the software architecture domain. From this perspective, architecture models describe the software architecture of a system according to the structure and constraints dictated by the AL metamodel, and model transformation engines and generators (as well as other MDE techniques) can be used to accommodate the AL requirements discussed earlier. More specifically, in the following we present how MDE techniques might be successfully used for defining and managing domain-specific languages in the software architecture domain:

1. MDE can be used to define precise and unambiguous ALs that contain only the model elements that the domain requires, as well as UML profiles that extend the UML infrastructure. Empirical studies show that UML isn't universally accepted,[3] while DSLs are far more prevalent than expected. Metamodeling and profiling provide the techniques required to support the definition or extension of modeling languages with a focus on nonfunctional properties such as performance, scalability, and security.
2. MDE tools give behavioral semantics to an AL by means of constraint languages—for example, OCL (www.omg.org/spec/OCL)—by mapping the language's structure onto a semantic domain (for example, via model transformations).
3. Finally, MDE provides a set of engines for graphical, tree-based, and textual editors with various levels of automation, such as GMP (www.eclipse.org/modeling/gmp) and GME (www.isis.vanderbilt.edu/projects/gme).

In the following, we highlight the main MDE mechanisms and techniques for managing architecture descriptions with a focus on multiview modeling, language extensibility, and programmatic access to architecture model elements:

1. MDE promotes the use of multiple views linked by means of suitable relationships, which are fundamental to understanding the impact of design decisions. Each viewpoint and view can be described by a set of domain-specific languages (DSLs), and MDE can be used for developing DSLs and for tailoring them to the various needs of the architect. MDE provides model weaving, which establishes typed relationships between models. This technique stores the relationships in dedicated models called weaving models. MDE also provides model transforma-

tion techniques. Together, model weaving and transformation support the composition and reuse of viewpoints and help ensure consistency among different viewpoints and views.
2. MDE provides different techniques to manage language and tool extensibility. Among them, profiling is a well-known lightweight extension mechanism that has been a key enabler for UML's success and widespread use. Also, model weaving and model transformations are currently used for defining extension points and extension operators of modeling languages in a nonintrusive way (that is, without having to modify or adapt the language being extended). Examples of these technologies are AMMA (http://wiki.eclipse.org/AMMA) and Epsilon (www.eclipse.org/epsilon).
3. MDE offers various facilities for building programming frameworks based on the structure of DSML metamodels. These facilities have different degrees of automation, ranging from fully automatic programming-framework generation (for example, with support for model persistency, model validation, transactions, and so on) to simple Java class generation.

In the following we focus on tool support provided by already existing MDE engines, which can play a relevant role in architecture descriptions.

1. Practitioners can use model transformations such as QVT (www.omg.org/spec/QVT) and ATL (www.eclipse.org/atl) to automatically obtain analysis models from architectural models and to propagate analysis results back to architectural models. MDE researchers have proposed many model transformation languages, each with specific features such as directionality, incrementality, tracing support, and so on.[11]
2. Practitioners can also use model transformations to automatically obtain various types of artifacts spanning the development life cycle. They can also be able to use model weaving for similar purposes. Practitioners can use them to carry out traceability analysis (between SA elements and requirements, design decisions, generated skeleton code, financial prospects, and so on) and change impact analysis while maintaining the system.
3. When dealing with a large ecosystem of models (that is, a large set of models representing the same system), MDE provides a technique called megamodeling (http://wiki.eclipse.org/AM3), which lets practitioners keep an organized register of all the involved models and their relations. MDE also provides a complementary approach to megamodeling called virtual modeling (https://code.google.com/a/eclipselabs.org/p/virtual-emf). It promotes the management of a large amount of information with a single metamodel representing all the domain concepts and, when needed, projects specific information into smaller and more manageable models on demand. When dealing with a large number of models, MDE techniques and tools allow practitioners to orchestrate a set of transformations among them and to consider those transformations as a unique chain (http://atenea.lcc.uma.es/index.php/Main Page/Resources/Wires*).
4. MDE techniques, such as those provided by EMF Compare (www.eclipse.org/emf/compare) and CoDesign (http://softarch.usc.edu/?ronia/codesign), can automatically calculate model differences. This is an essential process to control model changes and evolutions made by geographically distributed users. MDE also provides techniques with various levels of automation to propagate model changes to geographically distributed users and to notify users about conflicts upon concurrent modifications.
5. MDE techniques provide ways to support the management of different versions of software architecture models, to effectively match and merge different versions of one model, and to identify and solve possible conflicts.
6. Knowledge-sharing tools such as wikis and semantic wikis can support architects in their decision-making process and rationale. MDE can enable

> Model-driven engineering provides different techniques to manage language and tool extensibility.

ABOUT THE AUTHORS

PATRICIA LAGO is an associate professor of computer science and leader of the software and services group at the VU University Amsterdam. Her research interests are in energy-aware and sustainable software engineering, software architecture, and service orientation. Lago received a PhD in control and computer engineering from Politecnico di Torino. She's chair of the IEEE/IFIP WICSA Steering Committee and a member of the IFIP 2.10 Working Group on Software Architecture, the IFIP 2.14 Working Group on Services-Based Systems, and the Dutch Knowledge Network on Green Software. Lago is a member of ACM and IEEE. Contact her at p.lago@vu.nl.

IVANO MALAVOLTA is a postdoctoral researcher at the Gran Sasso Science Institute. His research focuses on software architecture, model-driven engineering (MDE), and mobile-enabled systems, especially how MDE techniques can be exploited for architecting complex and mobile-enabled software systems at the right level of abstraction. Malavolta received a PhD in computer science from the University of L'Aquila. He's a member of ACM and IEEE. Contact him at ivano.malavolta@gssi.infn.it.

HENRY MUCCINI is an assistant professor of computer science at the University of L'Aquila. His research focuses on the role of software architectures in producing quality software, and specifically, how they can be used for the verification and validation of complex software systems. Recent research is on the architecting of wireless sensor networks and mobile systems. Muccini received a PhD in computer science from the University of Rome–La Sapienza. He cochaired the Program Committee of AST 2013, QSIC 2012, and Euromicro SEAA 2012 and is a member of ACM SIGSOFT and IEEE. Contact him at henry.muccini@univaq.it.

PATRIZIO PELLICCIONE is an associate professor of computer science at the Chalmers University of Technology and University of Gothenburg, where he was recently named a Docent of Software Engineering, and an assistant professor of computer science (on leave) from the University of L'Aquila. His research focuses on software engineering, software architecture modeling and verification, and formal methods. Pelliccione received a PhD in computer science from the University of L'Aquila. He's one of the founding members of the European Research Consortium for Informatics and Mathematics Working Group on Software Engineering for Resilient Systems. Contact him at patrizio.pelliccione@gu.se.

ANTONY TANG is an associate professor of computer science in the Swinburne University of Technology's Faculty of Science, Engineering, and Technology. His research interests include software architecture design reasoning, software development processes, software architecture, and knowledge engineering. Tang received a PhD in information technology from the same university. He's a member of ACM and IEEE. Contact him at atang@swin.edu.au.

seamless integration of all artifacts used throughout the project (for example, financial prospects, architecture models, and stakeholder concerns) and knowledge-sharing tools. From this perspective, architects can integrate artifacts by transforming them into (semantic) wiki pages and establishing tracing dependencies between the modeling artifacts and the content in a knowledge-sharing tool.

The above discussed MDE techniques can surely play the role of a technological solution for successfully supporting the requirements of next-generation ALs. Nevertheless, there are some known limitations of MDE techniques that may limit their usage in their current form; those limitations are discussed in the next section.

Limitations of MDE in Industrial Adoption

MDE techniques and tools address many concerns an AL designer might have, but empirical studies show that some barriers exist for its adoption. Using a combination of online questionnaires (449 responses) and interviews (22 MDE practitioners), one study showed that the barriers hampering the industrial adoption of MDE are not only technical and tool related but also social and organizational.[3] As examples of organizational change management, the study pointed out that the successful adoption of MDE techniques needs a progressive and iterative approach, integration with existing organizational commitments, and a clear business focus.

Another study reported a taxonomy of factors that play a role in MDE adoption.[4] This empirical study was based on 19 interviews

with MDE practitioners working in 18 different companies. The researchers used their analysis of the data to define the taxonomy that they then validated through another 20 interviews carried out in two companies. The study shows that "MDE can be very effective but it takes effort to make it work."[4] From the technological point of view, the main limitations are the immaturity of tool support as well as its complexity and lack of usability. Practitioners highlighted that MDE often lacks consideration for how people think and work. Moreover, MDE requires investment in training, process change, and cultural shift.

Moreover, the success of MDE technologies depends on the domain they are applied to. Lessons from the first of the two studies just mentioned show that MDE techniques are useful in creating well-defined software architectures.[3] In fact, the interviewees unanimously argued that MDE makes it easier to define explicit architectures, especially when MDE is a ground-up effort.

Our research provides practitioners, researchers, and tool vendors a practitioner-proven guide to focus on the requirements for designing and developing new ALs. In this context, this article offers a starting set of sources of MDE technologies, together with a thorough mapping of MDE techniques with respect to next-generation AL requirements.

The mapping between MDE techniques and AL requirements helps in understanding how an existing technological solution (MDE, in this case) can be leveraged to successfully support the requirements of next-generation ALs. More importantly, the mapping helps in elaborating and reusing the knowledge base about ALs accumulated over the years. Indeed, MDE offers more than a way to realize ALs; it further suggests how ALs can be integrated in the broader development process where architecture is the main driver of the development of a software system. Overall, this article is suggesting ways for practitioners to finally bring ALs to industry.

References

1. B. Foote, N. Harrison, and H. Rohnert, *Pattern Languages of Program Design*, Addison-Wesley, 1995.
2. I. Malavolta et al., "What Industry Needs from Architectural Languages: A Survey," *IEEE Trans. Software Eng.*, vol. 39, no. 6, 2013, pp. 869–891.
3. J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-Driven Engineering Practices in Industry: Social, Organizational and Managerial Factors That Lead to Success or Failure," *Science of Computer Programming*, vol. 89, part B, Sept. 2014, pp. 144–161.
4. J. Whittle et al., "Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?," *Model-Driven Engineering Languages and Systems*, LNCS 8107, 2013, pp. 1–17.
5. *ISO/IEC/IEEE 42010-2011, Systems and Software Engineering—Architecture Description*, IEEE, 2011.
6. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *SIGSOFT Software Eng. Notes*, vol. 17, no. 4, 1992, pp. 40–52.
7. N. Medvidović et al., "Modeling Software Architectures in the Unified Modeling Language," *ACM Trans. Software Eng. and Methodology*, vol. 11, no. 1, 2002, pp. 2–57.
8. D.C. Schmidt, "Model-Driven Engineering," *Computer*, vol. 39, no. 2, 2006, pp. 25–31.
9. D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of 'Semantics'?," *Computer*, vol. 37, no. 10, 2004, pp. 64–72.
10. D. Falessi et al., "The Value of Design Rationale Information," *ACM Trans. Software Eng. Methodology*, vol. 22, no. 3, 2013, pp. 21:1–21:32.
11. K. Czarnecki and S. Helsen, "Feature-Based Survey of Model Transformation Approaches," *IBM Systems J.*, vol. 45, no. 3, 2006, pp. 621–645.