Evaluation of MDE Tools from a Metamodeling Perspective

João de Sousa Saraiva, INESC-ID/Instituto Superior Técnico, Portugal

Alberto Rodrigues da Silva, INESC-ID/Instituto Superior Técnico, Portugal

ABSTRACT

150 words or less

Keywords: evaluation framework; metamodel; MDE; tool

INTRODUCTION

Ever since the appearance of computers, researchers have been trying to raise the abstraction level at which software developers write computer programs. Looking at the history of programming languages, we have witnessed this fact, with languages evolving from raw machine code to machine-level languages, afterward to procedural programming languages, and finally to object-oriented languages, which allow developers to write software by mapping real-world concepts into modular segments of code (called objects). Still, object-oriented languages are too "computing-oriented" (Schmidt, 2006), abstracting over the solution domain (computing technologies) instead of the problem domain.

Currently, the abstraction level is being raised into the model-driven engineering (MDE) paradigm (Schmidt, 2006). In this abstraction level, models are considered first-class entities and become the backbone of the entire MDE-oriented software development process; other important artifacts, such as code and documentation, can be produced automatically from these models, relieving developers from issues such as underlying platform complexity

or the inability of third-generation languages to express domain concepts.

MDE is not a new idea. Already in the 1980s and 1990s, computer-aided software engineering (CASE) tools were focused on supplying developers with methods and tools to express software systems using graphical general-purpose language representations. The developer would then be able to perform different tasks over those representations, such as correction analysis or transformations to and from code. However, these CASE tools failed due to issues such as (a) poor mapping of general-purpose languages onto the underlying platforms, which made generated code much harder to understand and maintain, (b) the inability to scale because the tools did not support concurrent engineering, and (c) code was still the first-class entity in the development process while models were seen as only being suited for documentation (Schmidt, 2006). Currently, there are better conditions for such modeling tools to appear. Software systems today are reaching such a high degree of complexity that third-generation languages simply are not sufficient anymore; another abstraction level over those languages is needed. This need, combined with the choices of IT development platforms currently available (Java, .NET, etc.), to which models can be somewhat easily mapped, is the motivation for the adoption of MDE. There are already a few MDE-related case studies available, such as Zhu et al. (2004) and Fong (2007), but since most MDE work is still in the research phase, there is still a lack of validation through a variety of real business case studies.

There are already multiple MDE initiatives, languages, and approaches, such as the unified modeling language (UML), the MetaObject Facility (MOF), the model-driven architecture (MDA), and domain-specific modeling (DSM; Kelly & Tolvanen, 2008). There are also other derivative approaches, such as software factories (http://msdn2.microsoft.com/en-us/teamsystem/aa718951.aspx) that follow the MDE paradigm. Nevertheless, it is important to note that these initiatives are not a part of MDE; rather, MDE itself is a paradigm that is independent of language or technology, and is addressed by these initiatives.

All these approaches share the same basic concepts. A model is an interpretation of a certain problem domain, a fragment of the real world over which modeling and system development tasks are focused, according to a determined structure of concepts (Silva & Videira, 2005). This structure of concepts is provided by a metamodel, which is an attempt at describing the world around us for a particular purpose through the precise definition of the constructs and rules needed for creating models (*Metamodel.com*, n.d.). These basic concepts are the core of metamodeling, the activity of specifying a metamodel that will be used to create models, which is the foundation of MDE.

From the developer's point of view, a key issue for acceptance of any approach is good tool support so that software programs can be created in an easy and efficient manner. There is a wide variety of modeling tools available today, covering most modeling standards and approaches in existence. For example, Rational Rose and Enterprise Architect (EA; SparxSystems, n.d.) are only two examples of a very long list of tools that support UML modeling. DSM has recently become popular with the developer community, with tools such as Microsoft's DSL Tools (MSDSLTools, n.d.) or MetaCase's (n.d.) MetaEdit+.

The aim of this article is to present our evaluation framework for tool support of the metamodeling activity, and to evaluate a small set of tools according to this framework; although these tools do not reflect everything that is currently available in MDE tools, they address the MDE-based approaches presented in this article by providing the features typically found in tools of their corresponding approach. The evaluation framework used in this article focuses on the following issues: (a) supported exchange formats, (b) support for model transformation and code generation, (c) tool extensibility techniques, (d) the logical levels that can be manipulated, (e) support for specifying metamodel syntax and semantics,

and (f) complexity of the meta-metamodel hard-coded into the tool. The final purpose of this evaluation is to determine the strengths and weaknesses of the support that each of these MDE tools offer to the developer's tasks.

This article is divided as follows. The second section presents a brief overview of MDE and some related concepts, standards, and approaches. Then the article describes the evaluation framework, the selected modeling tools, and the results of their evaluation. Next it discusses the current status of MDE-based software tools and some open research issues for metamodeling. The final section presents the conclusions of this work.

MODEL-DRIVEN ENGINEERING

Software systems are reaching such a high degree of complexity that the current third-generation programming languages (like Java or C#) are not sufficiently adequate to create such systems in an easy and efficient manner. One of the problems with current programming languages is that they are still too oriented toward specifying how the solution should work instead of what the solution should be. This leads to a need for mechanisms and techniques that allow the developer to abstract over current programming languages and focus on creating a good solution to a certain problem.

Model-driven engineering (sometimes called model-driven development, or MDD) is an emerging paradigm based on the systematic use of models as first-class entities of the solution specification (Schmidt, 2006). Unlike previous software development paradigms based on source code as a first-class entity, models become first-class entities, and artifacts such as source code or documentation can then be obtained from those models.

It is very important to note that, although MDE is often mentioned alongside MDA (which is explained further later), MDE does not depend on MDA, nor is MDA a subset of MDE. In fact, MDA is one of several initiatives that intend to address the MDE paradigm.

The OMG's Approach to MDE

The Object Management Group (OMG) has created its own MDE initiative based on a set of OMG standards that make use of techniques for metamodeling and model transformation.

Unified Modeling Language

UML (http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf), currently in Version 2.1.1, is a general-purpose modeling language originally designed to specify, visualize, construct, and document information systems. UML is traditionally used as a metamodel (i.e., developers create models using the language established by UML). However, the UML specification also defines the profile mechanism, which allows for new notations or terminologies, providing a way to extend metaclasses to adapt them for different purposes. Profiles are collections of stereotypes, tagged values, and constraints (Silva & Videira, 2005). A stereotype defines additional element properties, but these properties must not contradict the properties that are already associated with the model element; thus, a profile does not allow the user to edit the metamodel.

Although UML was definitely a step forward in setting a standard understood by the whole software engineering community and aligning it toward MDE, it is still criticized for reasons such as (a) being easy to use in software-specific domains (such as IT or telecom-style systems) but not for other substantially different domains, such as biology or finance (Thomas, 2004), (b) not being oriented to how it would be used in practice (Henderson-Sellers, 2005), and (c) being too complex (Siau & Cao, 2001). Nevertheless, UML is often the target of overzealous promotion, which raises user expectations to an unattainable level; the criticisms that follow afterward are usually influenced by this (France, Ghosh, Dinh-Trong, & Solberg, 2006). An example of such a criticism is the one regarding the difficulty in using UML to model non-software-related domains: Although UML is a general-purpose modeling language, it is oriented toward the modeling of

software systems and is not intended to model each and every domain.

MetaObject Facility

MOF (http://www.omg.org/cgi-bin/apps/doc?formal/06-01-01.pdf), currently in Version 2.0, is the foundation of OMG's approach to MDE. UML and MOF were designed to be themselves instances of MOF. This was accomplished by defining the UML Infrastructure Library (http://www.omg.org/cgi-bin/apps/doc?formal/05-07-05.pdf), which provides the modeling framework and notation for UML and MOF, and can also be used for other metamodels. Figure 1 illustrates the dependencies between UML and MOF; note that MOF can be described using itself, making it reflexive (Nobrega, Nunes, & Coelho, 2006). Besides UML, the OMG has also defined some other MOF-based standards, such as the XML (extensible markup language) metadata interchange (XMI) and query-views-transformations (QVT).

XMI allows the exchange of metadata information by using XML, and it can be used for any metadata whose metamodel can be specified in MOF. This allows the mapping of any MOF-based metamodel to XML, providing a portable way to serialize and exchange models between tools. Nevertheless, users often regard XMI as a last resort for exchanging models between tools because tools frequently use their own vendor-specific XMI extensions; thus they lose information when exchanging models between different tools. The QVT specification defines a standard way of transforming source models into target models by allowing the definition of the following operations: (a) queries on models, (b) views on metamodels, and (c) transformations of models. One of the most interesting ideas about QVT is that the transformation should itself be considered an MOF-based model, which means that QVT's syntax should conform to MOF. Figure 2 presents OMG's typical four-layer architecture: (a) MOF is the meta-metamodel in the M3 layer, (b) UML, an instance of MOF, is the metamodel in the M2 layer, (c) the user model contains model elements and snapshots of instances of these model elements in the M1 layer, and (d) the M0 layer contains the runtime instances of the model elements defined in the M1 layer.

Model-Driven Architecture

MDA is OMG's framework for the software development life cycle driven by the activity of modeling the software system (Kleppe, Warmer, & Bast, 2003). It is based on other OMG standards such as UML, MOF, QVT, and XMI, and places a greater emphasis on UML model transformation techniques (through QVT) than on metamodeling itself; however, it should be noted that QVT model transformations are made possible only because of the model-metamodel relationship between UML and MOF.

Figure 1. The dependencies between UML and MOF

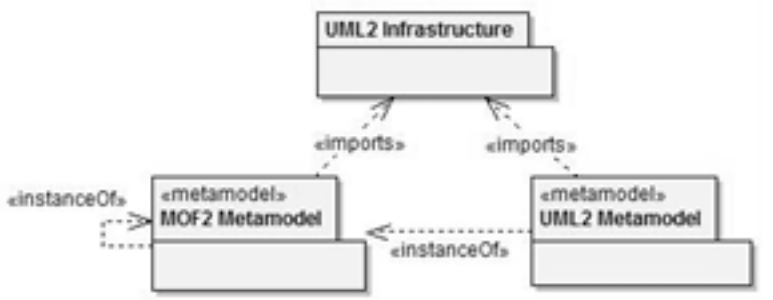
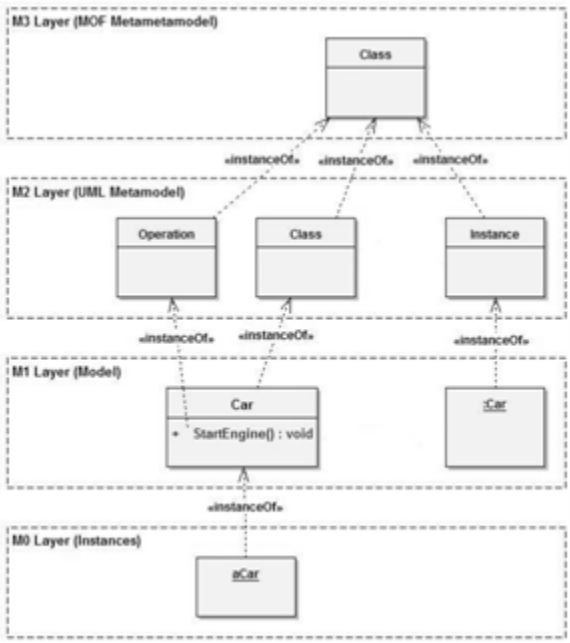


Figure 2. An example of OMG's four-layer metamodel architecture



MDA defines two types of models: (a) the platform-independent model (PIM) and (b) the platform-specific model (PSM; Kleppe et al., 2003). A PIM is a model with a high level of abstraction that makes it independent of any implementation technology, making it suitable to describe a software system that supports a certain business without paying attention to implementation details (like specific relational databases or application servers). A PSM also specifies the system, but in terms of the implementation technology. A PIM can be transformed into one or more PSMs, each of those PSMs targeting a specific technology because it is very common for software systems today to make use of several technologies. Figure 3 presents an overview of MDA; the solid lines connecting the boxes are transformations, which are defined by transformation rules. MDA prescribes the existence of transformation rules, but it does not define what those rules are; in some cases, the vendor may provide rules as part of a standard set of models and profiles.

MDA still faces some criticism in the software engineering community because of issues such as its usage of UML (Thomas, 2004) and the view that while current MDA generators are able to generate a significant portion of an application, they are not particularly good at building code that works within an existing code base.

Domain-Specific Modeling

DSM (Kelly & Tolvanen, 2008) uses problem-domain concepts as the basic building blocks of models unlike traditional CASE, which uses programming-language concepts. From a technological perspective, DSM is supported by a DSM system, which can be considered as an application for making domain-specific CASE tools (or as a tool-building environment to create CASE tools that can be used to produce applications). Thus, DSM adds an abstraction layer over traditional CASE, enabling the domain-specific configuration of the resulting modeling application as illustrated in

Figure 3. An overview of MDA

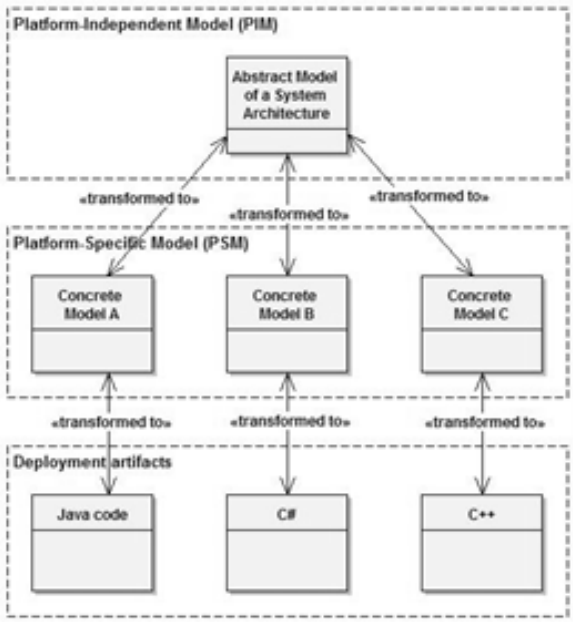


Figure 4. Because of this, DSM systems are also called meta-CASE tools. DSM is closely related to the concept of domain-specific language (DSL). A DSL is a language designed to be useful for a specific task (or a specific set of tasks) in a certain problem domain unlike a general-purpose language (Kelly & Tolvanen). As Figure 5 illustrates, due to a DSL's highly specialized nature, DSLs and corresponding generators are usually specified by experts (i.e., experienced developers) in the problem domain; other developers, less experienced with the mapping between domain concepts and source code, will invoke the DSL in their own code. A well-known example of a DSL is the standard query language (SQL), which is a standard computer language for accessing and manipulating databases (so, SQL's problem domain is the domain of database querying and manipulation).

Developers usually prefer DSLs to UML because of the set of used concepts: The latter uses programming concepts directly, which places models at the same abstraction level as source code; a DSL uses concepts from the problem domain, which means developers do not need to worry about how those concepts will map to code.

Figure 4. How CASE and DSM systems are related

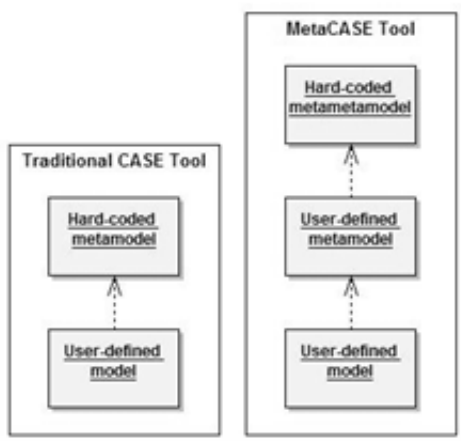
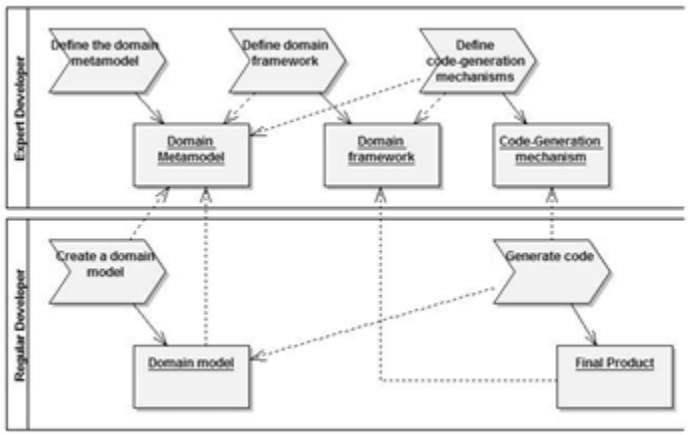


Figure 5. Using the expertise of some developers to orient other developers toward the problem domain



UML itself can be seen as a set of DSLs (corresponding to use-case diagrams, class diagrams, activity diagrams, etc.); however, these would be dependent on each other in a "DSL spaghetti" manner. UML can also be used to define DSLs using the profile mechanism, although this does bring some limitations that DSLs do not, such as the ability to ignore the semantic constraints already defined in UML.

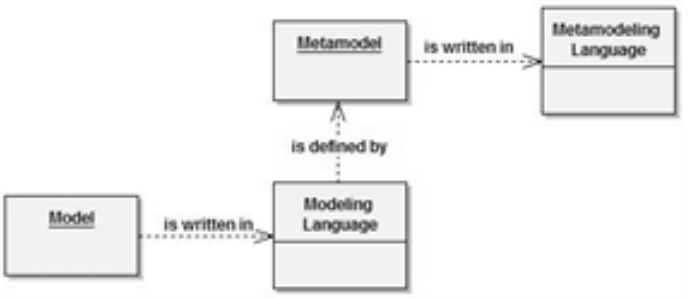
Metamodeling

The approaches presented lead us to the point where we can see that all concepts presented here are deeply related among themselves. We have a recurring pattern—the usage of metamodels and their instances of models—and the only real difference (in modeling terms) between all these approaches is in the number of layers each one uses. So, aside from a question of vocabulary, all these MDE-based variants have their foundation on the same topic: metamodeling.

But what is metamodeling? Metamodel.com (*Metamodel.com*, n.d.) provides the following definitions: "metamodeling is the activity that produces, among other things, metamodels" and "a metamodel is a precise definition of the constructs and rules needed for creating models." These definitions agree with other definitions that can be found in literature, such as the ones in Kleppe et al. (2003) and Söderstrom, Andersso, Johannesson, Perjons, & Wangler (2002). This means that a metamodel provides a language used to create a model, as Figure 6 illustrates; similarly, a metamodel that defines the language in which another metamodel is specified is called a meta-metamodel.

Similar in concept to DSM, metamodeling is about developing a language (a metamodel) adapted to the problem domain; for example, MOF is a language adapted to the domain of object-oriented approaches to modeling (Atkinson & Kühne, 2005), while UML is a language adapted to the domain of object-oriented programming languages (OOPLs). A possible example, in the context of an organization, of what could be done with metamodeling can be the following: (a) the specification of a new language or metamodel (with an existing language as its metamodel, e.g., MOF or UML) that reflects the concepts, syntax, and semantics of the corresponding problem domain, which is the organization, (b) after creating a tool that supports the metamodel, the modeling of a solution using the organization's terms (e.g., the organization specifies a certain role R1 that can perform activities A1 and A2), and (c)

Figure 6. A metamodel defines a language used to create a model



depending on the features provided by the tool, an application that implements the designed solution could be generated (either by model transformations, or by direct generation of source code). In fact, the PSMs for the MDA approach (oriented toward the implementation domain) can be obtained by using UML profiles tailored to an OOPL's concepts (such as C#'s class, struct, etc.). This would present an advantage over traditional development approaches as the solution would be created using the organization's terms instead of using implementation terms; we later present a more detailed view of how software development can be done combining metamodeling and model transformations. An example of the need of using metamodeling and metamodels can be found in Zhao and Siau (2007), which uses metamodels to handle the mediation of information sources.

The main difference (in modeling terms) between the presented modeling approaches is their number of modeling layers (i.e., model-metamodel relationships). Theoretically, the number of layers could be infinite, but any particular approach should have a specific number of layers; otherwise, its implementation would be impractical, if not impossible.

It is still rare to find a development tool that has explicit support for metamodel creation and/or configuration, which can be surprising if we consider that metamodeling is one of the founding principles of MDE. This means that, until recently, a developer who wanted to use a certain metamodel would probably have to either (a) create a new modeling tool, which is not reasonable at all (Nobrega et al., 2006) or (b) settle on a CASE tool (with a hard-coded metamodel) that allows the developer to perform the desired task with the least possible hassle. However, adding metamodeling support to a tool does bring some practical issues that should be mentioned, such as (a) separating the OOPL class- instance relation from the metamodel-model relation, (b) deciding whether the number of logical levels should be limited or potentially unbounded, and (c) deciding whether the tool should support model transformation and/or code generation.

In addition to these issues, it is also necessary to consider how to change a metamodel, which should be considered a very high-risk activity because models, consistent in the context of a certain metamodel, can become inconsistent with only some changes to that metamodel. Obviously, this introduces a potential element of disruption that should be avoided at all costs. One possible way of ensuring the validity of existing models when changing their metamodels is through the specification and application of model transformations (e.g., UML transformations, such as those presented in Selonen, Koskimies, & Sakkinen, 2003): For any change to a metamodel, a corresponding transformation must be defined that receives the previously

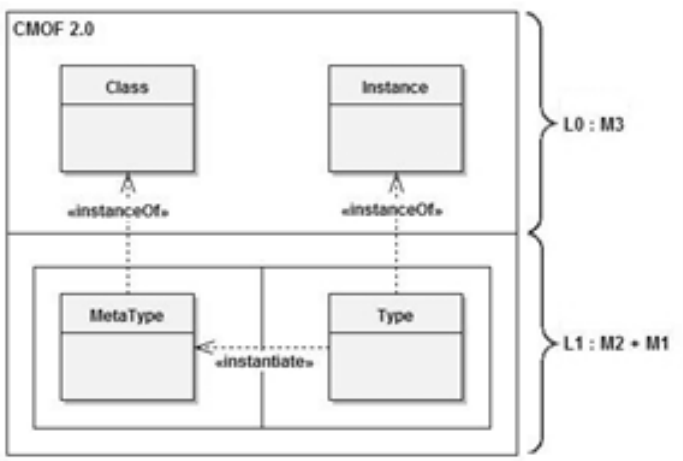
consistent models and produces new models consistent with the new metamodel.

However, in our research we have found no tool that addresses all of these metamodeling issues (although there are tools that address some of the presented issues).

Implementing a modeling tool with just one logical level (i.e., user model editing and a hard-coded metamodel) is easily done with current OOPLs using the class-instance relation: The logical level is implemented by the instance level. Metamodeling adds one (or more) logical level to the modeling tool, complicating the implementation as the instance level now has to hold two or more logical levels (Atkinson & Kühne, 2003). Level compaction (Atkinson & Kühne, 2005), an example of which is illustrated in Figure 7, is a technique that addresses this problem. Instead of the representation format for a level being defined by the level above, the format for a level is supplied by the modeling tool.

Although level compaction is essential for supporting multiple modeling levels, it is also important to determine whether the metamodel hard-coded into the tool allows such a number of levels. Atkinson and Kühne (2005) present the language and library metaphors, which allow tool creators to choose whether the number of layers in the tool's architecture should be restricted or potentially unbounded. When using the language metaphor, the basic elements of each layer (e.g., object, class, metaclass, etc.) are contained in the hard-coded metamodel itself; if the user wanted to add other basic elements, necessary for additional layers, it would be necessary to alter the hard-coded metamodel. This metaphor helps in supporting a standard (such as OMG's), but at the cost of not being able to edit the metamodel. On the other hand, in the library metaphor, the hard-coded metamodel consists only of a minimal core language, and the basic elements of each layer are available as predefined types in libraries to which the user can add elements (or remove them, if the tool allows it). With this metaphor, users can experiment with all metamodel layers because only the minimal core is hard-coded; the burden of syntax checking and language semantics is placed on the remaining metamodel layers. Note that if a tool does not use level compaction, then it obviously uses the language metaphor because the supported modeling levels are limited by the class-instance relation, which only allows one modeling level (in the instance level) besides the hard-coded metamodel (in the class level).

Figure 7. An example of using level compaction to compact three logical levels



Another important aspect to consider in metamodeling tools are model-to-model transformations. It would be natural that, after some time using such a tool, a developer has created or adopted some languages adjusted to relevant problem domains. However, after modeling a solution using the problem-domain language, the developer would then need to re-create the model in the language of the target domain. Obviously this would render the first model useless. So, if the tool also provided some kind of framework or language for specifying transformations between model languages, this would certainly benefit the developer.

EVALUATION OF MDE TOOLS

One of the key issues for the success of MDE is appropriate tool support as developers will only use a certain approach if it is supported by available tools. This section first presents the evaluation framework used through the rest of this article. Afterward, we present the tools that are evaluated. Finally, the evaluation's results are presented.

Evaluation Framework

This subsection presents the proposed evaluation framework used in this article. This framework focuses on a tool's support for metamodeling and involves the following dimensions, as illustrated in Figure 8:

1. supported exchange formats,
2. model transformation support,
3. usage of the level-compaction technique (Atkinson & Kühne, 2005),
4. usage of the language and library metaphors (Atkinson & Kühne, 2005),
5. the logical levels that the user can manipulate,
6. support for specifying metamodel syntax and semantics, and
7. the size of the hard-coded meta-metamodel.

The third and fourth dimensions were directly based on the conceptual framework defined in Atkinson and Kühne (2005); the other dimensions are derived from the issues described in the previous section ("Model-Driven Engineering") since this evaluation also tries to focus on the practical usage of these tools instead of exclusively considering architectural details. Note that we do not define a ranking system because the ultimate objective of this evaluation is not to determine the best tool but

Figure 8. An overview of the proposed evaluation framework

Dimension			Measurement range	Observations
Supported standard exchange formats		Metamodels	Set of standards (possibly None)	Measure can only be either a combination of standards, or None
		Models		
Model transformation framework			Yes, No	Values are mutually exclusive
Level Compaction			Yes, No	Values are mutually exclusive
Language and Library metaphors			Language metaphor, Library metaphor	Values are mutually exclusive; If *Level Compaction* is *No*, value is *Language metaphor*
Number of levels the user can manipulate			Set of natural numbers	Values are mutually exclusive
Support for metamodel specification	Syntax	Supports specification	Yes, No	Values are mutually exclusive.
		Languages used	Set of languages (possibly None)	Measure can only be either a combination of languages, or None; If "*Supports specification*" is *No*, this should be empty
	Semantics	Supports specification	Yes, No	Values are mutually exclusive
		Languages used	Set of languages (possibly None)	Measure can only be either a combination of languages, or None; If "*Supports specification*" is *No*, this should be empty
Hard-coded metametamodel size			Small, Average, Large	Values are mutually exclusive

rather if (and how) the industry is currently addressing metamodeling. In addition, we believe it is up to each developer to determine what approach and tool characteristics are required for development. However, we do believe that this framework provides a practical contribution through its generic set of guidelines that help determine whether a tool can appropriately address metamodeling (both as an activity in itself and as an activity in the context of software development). Moreover, metamodeling is still an active research topic that is not addressed by many tools, and we believed that ranking these tools would ultimately yield unfair results (as some of the tools were not created to address this issue in the first place).

We also highlight the fact that, although this evaluation framework has been empirically validated (in the context of our experience with various MDE-based tools), some of these criteria and measurement metrics are still subjective and can be refined by performing an explicit validation of the existing criteria and their measurements metrics, according to approaches such as Moore and Benbasat (1991), and by adding further (and more objective) criteria that address other issues regarding metamodeling.

Supported Standard Exchange Formats

With all the modeling tools now available, the ability to exchange models between tools is becoming a very important requirement; the lack of this ability can easily lead to a situation in which a developer is stuck with a certain tool. This would require that tools be able to export and import models to and from a standard format, such as XMI. Although each tool creator is free to create or choose his or her own exchange format, it should be taken into account that developers usually choose tools that can import or export to standard formats, allowing models to be independent of the tools in which they are manipulated.

This dimension is divided into two subdimensions: (a) metamodels, which involves determining whether metamodels can be imported or exported, and (b) models, which involves determining whether user models can be imported or exported. This division is useful because the formats used by a tool to import, or export metamodels and models may be different; also, a tool may only allow the import or export of models but not metamodels. The values for both dimensions are the set of standards used (possibly none).

Model Transformation Framework

This dimension measures whether the tool supports model transformations, and only allows a single value from its measurement range: *yes*, meaning that the tool additionally provides a framework or language based on the metamodel or the meta-metamodel for specifying transformations between user models (such as QVT), and *no*, meaning that the tool does not provide such a framework.

Level Compaction

This dimension measures whether the tool uses the level-compaction technique (Atkinson & Kühne, 2005) and only allows a single value from its measurement range: *yes*, meaning that the tool uses level compaction and can therefore easily be adjusted to support additional logical levels, and *no*, meaning that the tool does not employ level compaction.

Language and Library Metaphors

This dimension measures which of the two metaphors (language or library; Atkinson & Kühne, 2005) are used in the tool, and only allows a single value from its measurement range: language metaphor or library metaphor, according to the metaphor used. Note that if the dimension level compaction evaluates as *no*, then the value of this dimension will obviously be the language metaphor, as presented in the previous section ("Model-Driven Engineering").

Number of Logical Levels the User can Manipulate

Despite what architectural options are present in a tool, one of the aspects that directly affects a tool's user is the number of logical levels

that can actually be manipulated in the tool (by creating, editing, or deleting elements) as a limited number may force the user to compact two or more metamodel levels into a single layer (i.e., the user places elements from several logical levels in a single level).

This dimension measures how many metamodel-model relationships can be handled by the tool, and it only allows the usage of a single natural number (i.e., 1, 2, etc.). For example, a typical UML CASE tool only allows the manipulation of one logical level (M1) as the creation of instances is still performed in M1.

Support for Metamodel Specification

In the evaluation of the support that a tool provides for specifying metamodels, it is important to analyze what a tool supports.

This dimension is divided into two other dimensions, syntax and semantics, evaluating the support that the selected tools provide to the specification of the syntax and semantics of metamodels, respectively. The definitions of *metamodel syntax* and *metamodel semantics* are similar to the ones found at http://www.klasse.nl/research/uml-semantics.html and are described next.

- *Syntax.* A metamodel's syntax consists of the set of model elements (i.e., graphical representations of domain elements) and the relationships between those model elements; this is very similar to the definition of *syntax* in the context of linguistics, in which syntax is the study of the way words are combined together to form sentences. The syntax dimension is divided into two subdimensions: specification support and languages used.
- *Specification support.* This dimension evaluates whether the tool supports the specification of the syntactic component of a metamodel (i.e., the graphical representation of its elements). It only allows a single value from its measurement range: *yes*, meaning that the tool allows the specification of the metamodel's syntax, and *no*, meaning that the tool does not support this.
- *Languages used.* This dimension determines the set of languages used by the tool to specify the metamodel's syntax (including proprietary or standard languages). Note that this dimension can only have a meaningful value when the specification-support dimension's value is *yes*.
- *Semantics.* A metamodel's semantics can be seen from two perspectives: the semantic domain and the semantics of each model element. The semantic domain consists of the whole set of domain elements that the metamodel is supposed to represent (i.e., the concepts that were captured during the analysis of the problem domain). On the other hand, the semantics of a certain model element is determined by the relation(s) between that model element and one or more domain elements.

This dimension is divided into two subdimensions, specification support and languages used, which evaluate some aspects of the mechanisms provided for defining metamodel semantics.

- *Specification support.* This dimension measures whether the tool supports the specification of the semantic component of a metamodel. It only allows a single value from its measurement range: *yes*, meaning that the tool allows specification of a metamodel's semantic constraints, and *no*, meaning that the tool does not support this.
- *Languages used.* This dimension, like the languages-used dimension of syntax, determines the set of languages used by the tool to define a metamodel's semantic constraints (such as OCL for MOF-based models, available at http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf). Note that this dimension can only have a meaningful value if the specification-support value is *yes*.

Hard-Coded Meta-Metamodel Size

An important aspect to consider is the size of the meta-metamodel hard-coded into the tool (or metamodel if the tool only allows creating user models) because it reflects how wide the range of metamodel primitives is. In this evaluation, we consider the size of a model (or a meta-metamodel, in this case) to be defined by the quantity of information involved in the formal specification of the model (i.e., how many objects, relationships, and constraints are used to specify the model); the explanation for this lies in the amount of information that the user should be aware of when creating a metamodel in order to take full advantage of the language provided by the meta-metamodel.

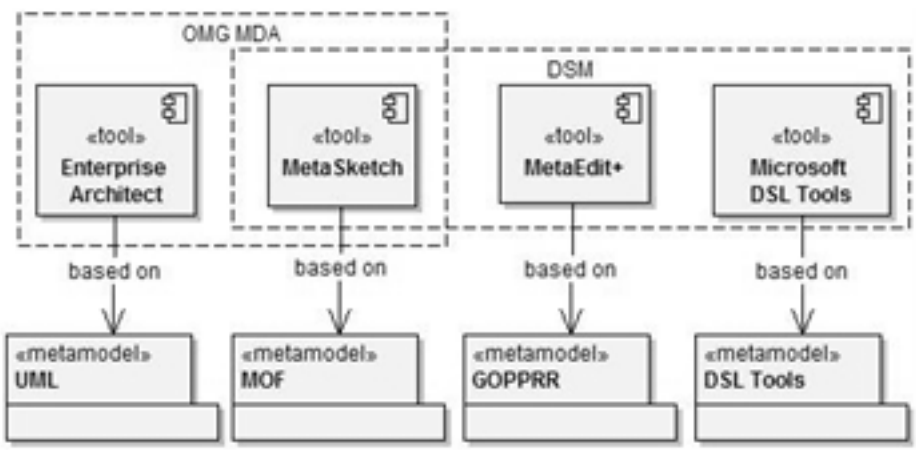
This dimension only allows a single value from its measurement range: (a) *small*, meaning that the tool's hard-coded meta-metamodel consists of 15 elements or less (in this article, we consider an element to be either an object, a relationship between objects, or a constraint), (b) *average*, meaning that it consists of 16 to 30 elements, and (c) *large*, meaning that it consists of more than 30 elements. It is important to note that this measurement is highly subjective since we know of no framework to objectively classify a model's size or complexity; ultimately, it is up to the reader to make his or her own definition of how large a meta-metamodel must be before it can be considered large.

MDE Tools

Figure 9 presents an overview of the small set of MDE tools used in this evaluation: Enterprise Architect (SparxSystems, n.d.), MetaSketch (Nobrega et al., 2006), MetaEdit+ (MetaCase, n.d.), and Microsoft's DSL Tools (MSDSLTools, n.d.).

The initial criteria used for the selection of MDE tools to evaluate were the following: (a) The tool must be recent (or still be under development) to ensure it addresses current MDE approaches, (b) each tool must address one of the MDE initiatives presented in the previous section, and (c) the tool must have a relatively smooth learning curve as developers are usually more inclined to choose tools that they find to be user friendly and that facilitate their activities. We searched the Internet for candidate tools that fit these criteria; however, we found many candidate tools, so we limited this evaluation to popular tools in order to keep the evaluation (and this article) simple. We also included MetaSketch in this evaluation because, although it is not yet popular, it explicitly addresses the metamodeling activity, so we believed that including it in the evalu-

Figure 9. The selected MDE tools



ation could yield some interesting results. We did not consider any of our own tools (i.e., developed in-house) as candidates for this evaluation in order to maintain an independent perspective over this tool evaluation and prevent us from inadvertently specifying dimensions that would favor any one of the tools being evaluated.

These tools were chosen because we consider that this set is a good representative of the current status of MDE-supporting tools currently available (e.g., Enterprise Architect can do most of what can be done with ArgoUML, http://argouml.tigris.org; Poseidon for UML, http://www.gentleware.com; Rational Rose 2003, http://www-306.ibm.com/software/awdtools/developer/datamodeler; or other UML modeling tools); they also presented enough differences amongst themselves to justify their inclusion in this evaluation. Although these tools do not reflect everything that is currently available in MDE tools, they address the MDE-based approaches defined earlier by providing the features that can often be found in typical tools of their corresponding approach.

The reason we evaluate only a small number of tools is article simplicity and size. However, it is important to reiterate that there are a great number of other tools available, such as the Generic Modeling Environment (GME; http://www.isis.vanderbilt.edu/projects/gme) or the Eclipse Graphical Modeling Framework (GMF; http://www.eclipse.org/gmf). Although in this article we only evaluate this small set of tools, we believe that an evaluation of a greater number of tools, including a wider range of areas such as ontology modeling or enterprise architecture modeling, would yield some very interesting results to complement those obtained here. An added advantage of such an evaluation would also be the diverse set of metamodels used by the evaluated tools (e.g., enterprise modeling tools tend to use enterprise-oriented metamodels, such as the TOGAF or Zachman framework).

Traditional CASE Tools

Although traditional CASE tools may be adequate for the development of small and simple software systems, they clearly do not support the development tasks that come with larger, complex systems. One of the main problems of such tools is that they only support a specific metamodel, usually UML, and do not offer support for altering that metamodel (although UML does provide the profile mechanism, supported by some UML modeling tools).

This type of tools is included in this evaluation to determine whether current typical CASE tools could easily be adapted to allow the creation of models based on a user-specified language. For the evaluation purposes of this work, we chose Enterprise Architect (Sparx-Systems, n.d.) to represent traditional CASE tools as it is quite easy to use, provides good support for UML and its profile mechanism (in fact, EA makes the definition of a UML profile a simple and easy task), and seems to be one of the best representatives of the current status of CASE tools.

(For this evaluation, we used Enterprise Architect 6.5, which was the latest version of this tool at the time this work was written.)

MetaSketch

MetaSketch (Nobrega et al., 2006) is a MOF-based editor, unlike most editors, which are usually based on UML. It is based on the following ideas: (a) A metamodel is a model that conforms to MOF 2.0, not to UML 2.0, (b) the UML profile mechanism is not powerful enough to support the definition of new modeling languages, (c) a metamodel should be the primary artifact of a modeling language definition and developers should not need to code metamodels, and (d) a metamodel is not the final goal but the means used to produce models, so it is not reasonable to create another modeling tool each time another metamodel is specified. These ideas lead to MetaSketch, an editor that is MOF compliant, allowing the definition of any language that can be specified using MOF (i.e., a MOF-based metamodel). Thus, MetaSketch is best defined as a metamodeling tool.

MetaSketch does not offer code generation capabilities by itself, but it can import or export defined models and metamodels to XMI; the tool adheres strictly (with no vendor-specific extensions) to XMI 2.1 (Nobrega et al., 2006), so code generation could easily be handled by any code generator that can understand XMI. The tool also supports the definition of models conforming to metamodels specified in XMI (e.g., MOF or UML). Three metalevels, M3, M2, and M1, are supported by using level compaction. Figure 10 illustrates two interesting scenarios that are made possible by MetaSketch: the definition of a MOF metamodel by using itself (top), and the definition of the UML and CWM metamodels (bottom). In the first scenario, the user takes advantage of MOF's reflexive property in order to define a metamodel consisting of MOF itself (note the hard-coded MOF and the user-defined MOF); UML and CWM can then be defined as user models from that metamodel. In the second scenario, the user defines the UML metamodel by using the hard-coded MOF meta-metamodel. UML user models can then be created based on that metamodel (note that this second scenario is very similar to the typical OMG architecture, illustrated in Figure 2).

MetaEdit+

MetaEdit+, available at http://www.metacase.com, is a DSM-oriented environment (i.e., a meta-CASE tool) that allows the creation of modeling tools and generators fitting to application domains without having to write code (Tolvanen & Rossi, 2003). It uses a meta-metamodel called GOPPRR (graph, object, property, port, relationship, and role), named after the metatypes that are used when specifying the metamodel.

In MetaEdit+, an expert creates a modeling method by (a) defining a domain-specific language containing the problem domain's concepts and rules (in this article, we will treat a DSL in MetaEdit+ as a metamodel since the tool does treat DSLs as metamodels), and (b) specifying the mapping from that language to source code in a domain-specific code generator. Once the expert creates the modeling method (or even a prototype), the development team can start using it in MetaEdit+ to define models, and the corresponding code will be automatically generated from those models. The code generator itself uses a DSL that allows the developer to specify how to navigate through models and output its contents along with additional text. The tool also provides a repository for all modeling method information, allowing the storage and modification of modeling method definitions; any modifications to definitions are also reflected in their corresponding tools, models, and generators.

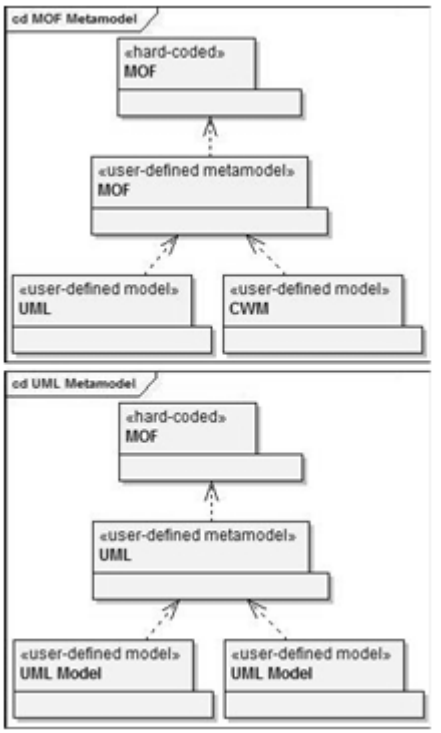
(For this evaluation, we used MetaEdit+ 4.5, which was the latest version of this tool at the time this work was written.)

Microsoft DSL Tools

Microsoft's DSL Tools, available at http://msdn.microsoft.com/vstudio/dsltools, is a suite of tools for creating, editing, visualizing, and using domain-specific data for automating the enterprise software development process. DSL Tools allow developers to design graphical modeling languages and to generate artifacts (such as code or documentation) from those languages; the visual language tools are based on Microsoft Visual Studio.

The process of creating a new DSL begins with the DSL Designer Wizard, which provides some metamodel templates (such as class diagrams or use-case diagrams) and guides the developer through specifying the features of the desired DSL. As a result of executing the wizard, a Visual Studio solution is created, containing a DSL project with the language's domain model (classes and relationships), its visual representation (diagram elements), and the mappings between domain elements and visual elements. The source code that will support the DSL tool is generated by using text templates, which process the DSL's specification and output the corresponding code. Developers can provide additional code to refine aspects of the model designer, define constraints over the language, and/or even alter the text templates (which can have substantial effects on the generated

Figure 10. MOF is used as a meta-metamodel and as a metamodel



source code). Testing is done within Visual Studio by launching another instance of the environment with the specified DSL tool. After ensuring that the tool is working correctly, the final step is creating a deployment package that allows its distribution.

(For this evaluation, we used the DSL Tools' Version 1 release, which was the latest version of this tool at the time this article was written.)

Applying the Framework

This subsection describes the small case study used to support this evaluation and the results obtained by applying the evaluation framework to each of the selected tools.

A Small Case Study: Social Network Metamodel

An essential part of the evaluation of a tool is determining how that tool actually supports the activities necessary toward the resolution of a certain problem. Thus, we use the facilities provided by each tool to specify and implement (when possible) a simple metamodel that supports the specification for simple social networks. This metamodel can be textually described by the following statements:

- A social network is composed of people and relationships between people.
- A person's participation in a relationship is defined by the role they play in it.
- A role must have a corresponding relationship.
- A role must have a corresponding person.
- A social relationship must involve at least two different people.

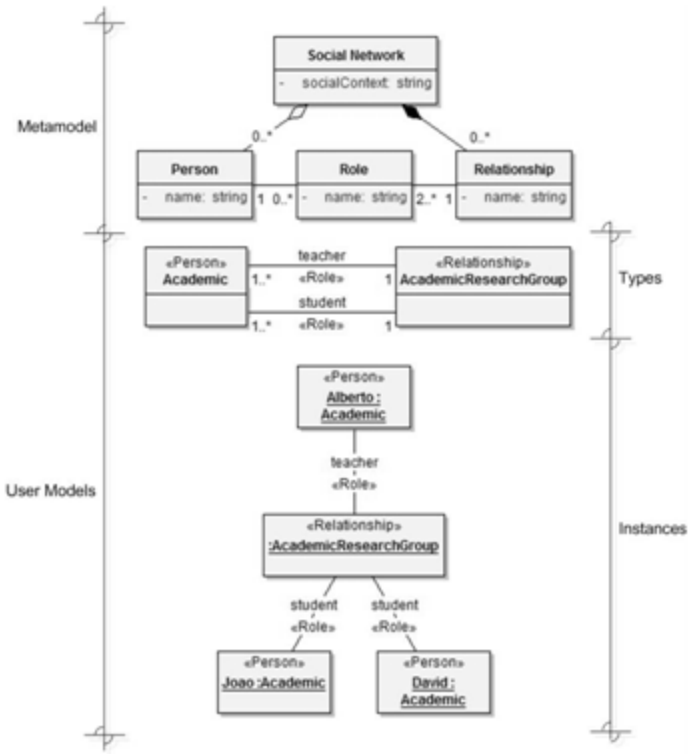
Figure 11 presents this metamodel (and two user models, for illustrative purposes) modeled in Enterprise Architect.

Note that this case study, because of its simplicity, could also be addressed with typical CASE tools (in fact, this is done in Enterprise Architect). However, the main objective of this article is to evaluate how the selected tools behave in specifying the Social Network metamodel and afterward producing and adapting a tool that can be used to create user models (i.e., with types and instances) using the language defined by that metamodel.

Evaluating the Tools

The evaluation framework's application to the presented tools was performed by us, so we did not need to resort to agreement measures, such as Cohen's Kappa coefficient. To compensate for the lack of a greater number of test participants, we tried not to define any dimensions that depended on the user's previous familiarity with one (or more) of the tools. Thus, the usage of each tool to define the Social Networks metamodel case study was

Figure 11. The Social Network metamodel and two user models



accompanied by thorough reading of available tool documentation and previous tests of the tool in order to gain a reasonable amount of experience with each of the selected tools. Nevertheless, we acknowledge that such dimensions are important to measure usability and the tool's learning curve (and can be a good indicator of whether the tool will be accepted by the community).

- *Enterprise Architect.* Enterprise Architect is an easy-to-use tool with a minimal learning curve. However, its traditional CASE-tool roots make it extremely limited when it comes to metamodeling. Since EA is a UML modeling tool, the only mechanism that it provides for metamodeling support is the UML profile mechanism, which only allows adding elements and semantics to the metamodel, but not altering it (i.e., editing or removing elements and constraints).

The definition of a profile in EA is limited to specifying the generic syntax of the profile (i.e., defining stereotypes and what metaclasses they extend, enumerations, etc.). Other semantic and syntactic relationships and constraints entered in the profile definition (using a text-based notation such as OCL) are not enforced when the user creates a model using that profile; the only validation that EA does enforce is the application of a stereotype to an instance of a metaclass (e.g., a stereotype that extends the metaclass Association cannot be applied to an instance of the metaclass Class). EA does present the advantage of not requiring the creation of a new tool adapted to the problem domain as it supports both the definition and application

of a UML profile (as is typically the case with profile-supporting CASE tools).

Like other CASE tools, EA does not appear to use level compaction or any similar technique because modeling is limited to one logical level; in this case, adapting the tool to support more logical levels (by using level compaction) would require an extra effort in order to separate the metamodel-model and class-instance relationships. The tool offers code generation capabilities and some predefined basic model transformations to support MDA, such as PIM to PSM. However, they require that PIMs and PSMs be specified in UML as it is the tool's hard-coded metamodel.

Figure 12 shows the definition of a profile representing the Social Networks metamodel previously presented; additionally, Figure 11 presents two user models (obtained through the application of the profile) modeled in EA.

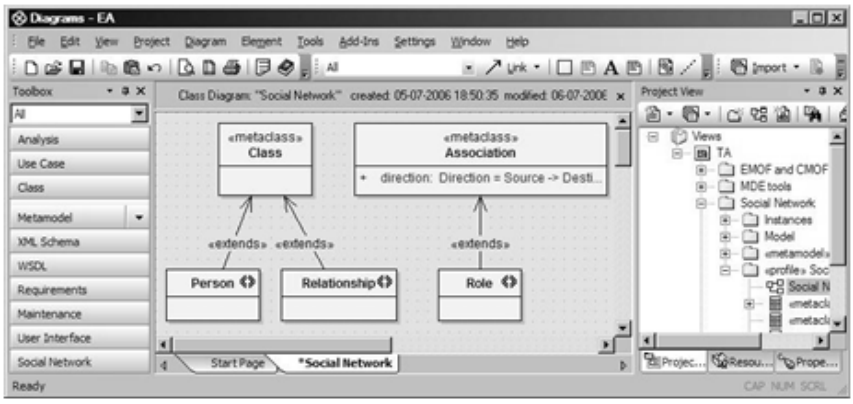
It is important to reiterate that the reason why EA is used in this evaluation is to show that typical CASE tools are not adequate for the metamodeling needs that are currently surfacing, even though EA (as other CASE tools) is not designed to support metamodeling; this evaluation is not meant in any way to diminish EA as a tool, and these results should not be interpreted as such.

- *MetaSketch.* From the set of evaluated tools, only MetaSketch supported metamodeling based on the MOF standard. The tool supports the XMI import and export of models and metamodels, so a user-defined model can become a metamodel simply by exporting it to XMI and then importing it from XMI as a metamodel. In fact, the tool can easily handle the XMI-based specifications of MOF and UML available on the OMG Web site.

MetaSketch uses the language metaphor (Nobrega et al., 2006), which in this case limits the user to manipulating two logical levels: the metamodel and the user model. However, MetaSketch uses level compaction, so it could be adapted to use the library metaphor with relatively little effort. Although MetaSketch does not support model transformations (to either source code or other models), this can be remedied because of the tool's XMI import and export capabilities; the user could export the model to XMI, and then process it with a code generator (such as the Eclipse Modeling Framework, available at http://www.eclipse.org/emf) or a model transformation tool (likely based on QVT).

The syntax of the metamodel is specified in XML (outside the tool's environment) by composing simple shapes (rectangles, ellipses,

Figure 12. A screenshot of Enterprise Architect with a profile definition

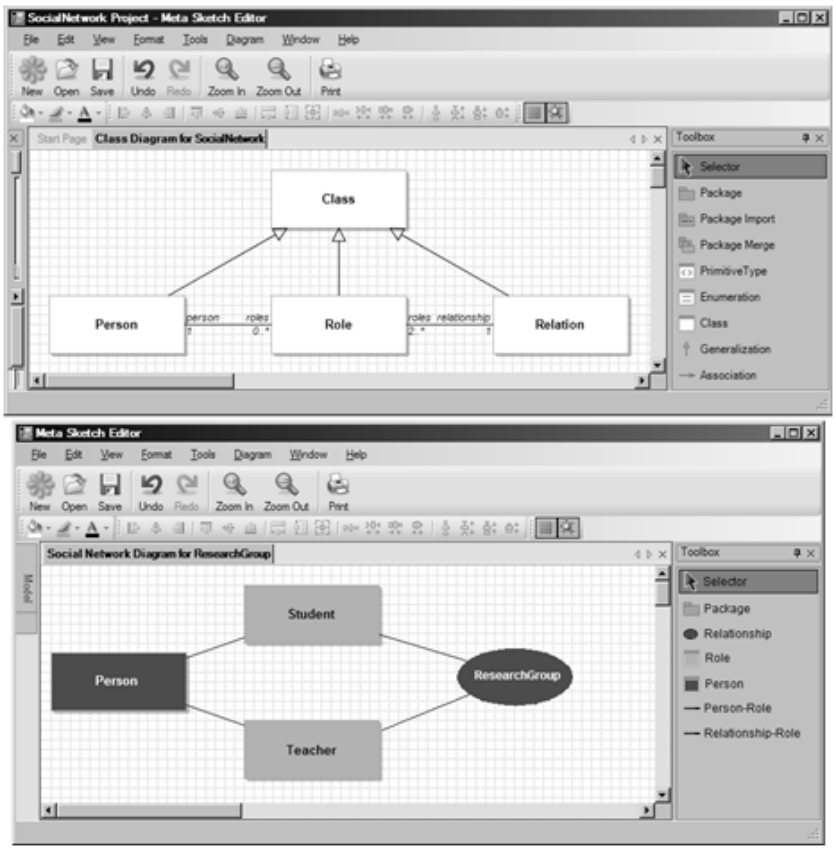


etc.) and using the tool's geometry management mechanism (Nobrega et al., 2006), which dynamically adjusts the spatial arrangement of those shapes. The semantics of the metamodel is specified in the tool itself when modeling the user model that later becomes the metamodel; however, there is no support yet for constraint specification. Nevertheless, it is important to note that the tool is still a prototype under active development, so it can be expected that such issues will be corrected in the future. Thus, the results obtained in this evaluation do not reflect the full potential of this tool.

- *MetaEdit+*. MetaEdit+ is based on a very simple and flexible meta-metamodel, GOPPRR; however, this meta-metamodel does not include behavioral features (only structural features), which can impact the possible set of metamodels that can be defined by the tool.

MetaEdit+ apparently uses the language metaphor, limiting the number of logical levels the user can edit to the metamodel and the user model. However, this metaphor is used not because of programming-language restrictions, but by choice of the tool creators, so the tool could be adapted to use the library metaphor with relatively little effort. Although the tool does not offer support for model transformations, it does provide a report mechanism that allows the generation of text-based artifacts (such as source code, HTML [hypertext markup

Figure 13. Social Network metamodel and user model in MetaSketch



language], or XML) based on the information available in the model's repository.

Syntax specification is done by creating instances of the meta-metamodel's elements and, eventually, creating vectorial images to represent those instances. Semantic specification is done when creating an instance of a graph (which corresponds to a type of model, like UML's class diagram or use-case diagram); constraints are then entered in the graph's corresponding form (e.g., "Objects of a certain type may be in, at most, a certain number of relationships"), which is designed to avoid as much manual text entering as possible (since it is prone to errors).

This tool did present a few important usability problems, such as the fact that it does not allow the altering of the superclass-subclass relationships between object types: Once the user chooses an object type's superclass (when creating the object type), it cannot be changed; the user should first draw the metamodel on a piece of paper or another modeling tool in order to obtain the definitive metamodel, and then re-create it within MetaEdit+.

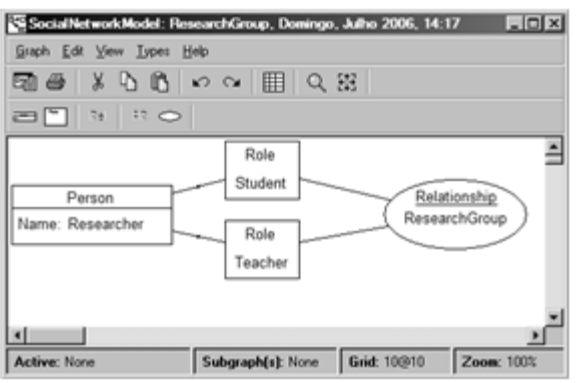
Figure 14 presents a model derived from the Social Network metamodel presented earlier.

- *Microsoft DSL Tools for Visual Studio.* This tool's meta-metamodel consists of the following elements: (a) class, (b) domain property, (c) embedding, (d) reference, and (d) inheritance. Like MetaEdit+, this meta-metamodel is highly object oriented but does not include behavioral features.

The tool's architecture is based on the language metaphor and limits the possible logical levels editable by the user to the metamodel and the user model. However, this limitation is because DSL Tools are based on the class-instance relation, so the adaptation of DSL Tools to use the library metaphor would likely require a great deal of effort.

The DSL designer itself is divided into two panes: Domain Model and Diagram Elements. In Domain Model, the developer identifies the relevant concepts of the problem domain and expresses them in the domain-model section of the designer along with model details like cardinality and source-code-specific details such as whether an association end should generate a property. Validations and constraints can also be specified by typing source code in additional validation classes. In Diagram Elements, aspects relating to the graphical layout of the model elements are specified, such as shapes used, association line styles, the shapes that can be on either end of an association, and how value properties are graphically displayed. Thus, the specification of the syntax and se-

Figure 14. Models of the Social Network metamodel in MetaEdit+



mantics of the metamodel is done entirely in the DSL designer (except for validations and constraints, which are expressed in source code) as the DSL Tools are highly focused on the graphical specification of user models and subsequent generation of text-based artifacts. Figure 15 presents the Social Network user model specified in DSL Tools.

Results

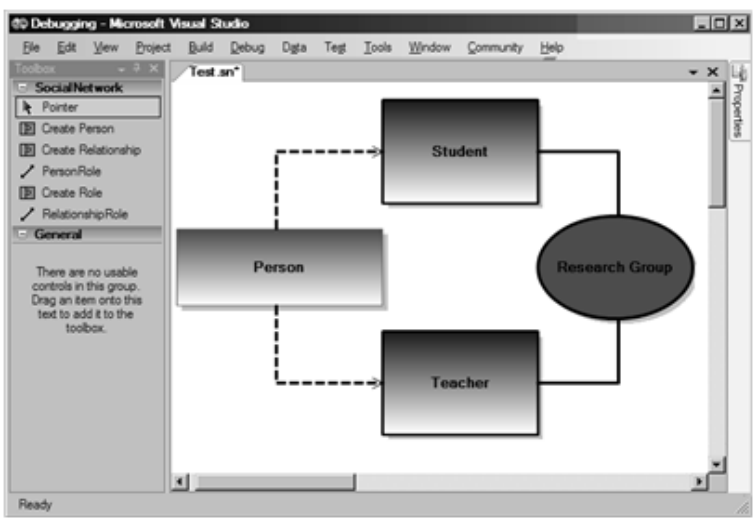
The results of this evaluation are shown in Figure 16. From these results, we can see that some tools already treat metamodel exchange as an important issue as only Enterprise Architect and MetaEdit+ do not export their metamodel definition. However, in Enterprise Architect's case, this is understandable since the metamodel (UML) is hard-coded into the tool and never changes. We find noteworthy the fact that MetaSketch is the only tool allowing metamodel import and export using a well-defined standard (XMI). User-model exchange, however, is supported by all tools, using either XMI or XML.

Model transformation does not currently seem to be a major concern as most tools do not provide any kind of support for it (only Enterprise Architect provides a framework for model-to-model or model-to-code transformations in the context of the MDA initiative), likely because of the immature state of the area.

Another interesting conclusion is that each of the evaluated tools uses the language metaphor, although most of them support the specification of two logical levels. This is likely due to the fact that a tool that supports more than two logical levels is likely to reveal itself as confusing since it can usually be assumed that developers will not need more than two logical levels (one to specify a language that represents the problem domain—and perhaps another language that represents the solution domain—and another to specify a solution to the problem). However, both MetaSketch and MetaEdit+ use level compaction, and thus could be adapted to use the library metaphor (therefore supporting additional logical levels) with little effort; DSL Tools would require a much more extensive effort to support such additional logical levels.

Most tools support the specification of a metamodel's syntax to some degree (either by associating elements with external image sources or by providing internal facilities to

Figure 15. Models of the Social Network metamodel in Microsoft's DSL Tools



create such graphical representations). However, the specification of a metamodel's semantic constraints seems to be sketchy at best, with only MetaEdit+ and DSL Tools supporting such constraint specifications. (MetaEdit+ uses its meta-metamodel concepts to establish constraints in the metamodel, while DSL Tools requires that developers use source code to specify constraints.)

Finally, the tools that do not follow a standard meta-metamodel (e.g., MetaEdit+ and MS DSL Tools) seem to prefer using a meta-metamodel that is as simple as possible: MetaEdit+'s consists of six elements, while DSL Tools' consists of five.

Discussion

Although CASE tools failed on their first appearance some years ago (Booch, Brown, Iyengar, Rumbaugh, & Selic, 2004), they brought the idea that development processes could be supported by such tools as long as those tools were adjusted to the development process. Early CASE tools were too inflexible, usually forcing the development process to be adjusted to the CASE tool instead of having the CASE tool support the development process. This led to the area of meta-CASE systems, which allow the automatic creation of development tools tailored to specific design processes, determined by organizational requirements.

The core problem with traditional CASE tools is that they only support specifying the solution; the identification of the problem-domain requirements is often done apart from these tools (usually in a word processor or similar). Hence, developers do not have a problem-domain-oriented language in which they can express the solution to the problem, forcing them to think of the solution in computational terms (toward which traditional CASE tool metamodels are especially oriented) rather than in problem-domain terms. The solution inevitably becomes misaligned with the problem domain and, therefore, with the problem itself. The consequences of this can be seen over the entire development process, but become especially critical during the product maintenance phase, when the product must be adapted to additional problem conditions and requirements, usually requiring extensive developer effort because of the difficulty of assuring that the product still solves the old problems while also solving additional problems.

Figure 16. The evaluation results

Dimension \ Tool			Enterprise Architect	MetaSketch	MetaEdit+	Microsoft DSL Tools
Supported standard exchange formats		Metamodels	None	XMI	None	Other (DSL definition format)
		Models	XMI	XMI	XML	XML
Model transformation framework			Yes	No	No	No
Level Compaction			No	Yes	Yes	No
Language metaphor or Library metaphor			Language metaphor	Language metaphor	Language metaphor	Language metaphor
Number of logical levels the user can manipulate			1	2	2	2
Support for metamodel specification	Syntax	Supports specification	Yes	Yes	Yes	Yes
		Languages used	None (Association between stereotypes and icons)	XML	None (Graphical vectorial drawing within the tool)	XML
	Semantics	Supports specification	No	No	Yes	Yes
		Languages used	-	-	GOPPRR (uses metamodel concepts to establish constraints)	.NET source-code (e.g., C#)
Hard-coded metametamodel size			Large (MOF + UML metamodel)	Large (MOF metamodel)	Small (GOPPRR - 6 elements)	Small (5 elements)

However, when considering metamodeling and meta-CASE tools, we need to be careful because of possible meta-metamodel fragmentation: In this evaluation, we can see an example of this as Microsoft DSL Tools uses its own meta-metamodel and so does MetaEdit+. This could lead to a panorama much like the one from a few years ago, in which there was a myriad of modeling languages (i.e., metamodels) all doing the same and yet all different among themselves. Now that the community (and the industry) is beginning to focus on metamodeling and meta-metamodels (i.e., metamodel languages), we need to start considering meta-metamodel standards as they help eliminate gratuitous diversity (Booch et al., 2004). Otherwise, the diversity of languages that would be defined would very likely lead to the fragmentation that UML was designed to eliminate in the first place.

All this is theory that must be put into practice in tools that developers can use. For such tools to be of help to the developer, they must support the whole software development life cycle, from requirements specification to deployment and maintenance. This also requires that tools allow developers to specify solutions in problem-domain terms, which of course requires that tools support some form of metamodeling. However, as the results of this evaluation show, the current tool support is primarily directed toward DSM, and issues such as model-to-model transformations (upon which MDA is based) are being left out in all but a few tools (such as the Eclipse Modeling Project, available at http://www.eclipse.org/modeling).

We believe MDA and UML have the potential to adequately cover the development phases more directly related to software itself, like implementation design and coding. However, MDA does not address the requirements phase, leading to the known gap between what the client wants the system to do and what the system actually does; in part, this is because UML is not adequate for requirements modeling. On the other hand, DSM's strength over MDA comes from the fact that it is more than adequate for requirements specification. Using a DSM system, a developer experienced in the problem domain creates a metamodel reflecting that domain and specifies how domain concepts are mapped to code (or any other artifact type). Requirements are then specified as models (oriented toward not the implementation but what the client wants the system to do) using the defined metamodel. These models are then mapped into code using the mappings initially defined. However, DSM as it is used today has a weak point: the transformation between models and code (or even between models of different languages). If the DSM system user wishes to switch target platforms (for example, from Java to .NET), the mappings will have to be re-created by the expert developer, unlike what happens with MDA, as PIMs and PSMs provide the ability to exchange target platforms with minimal extra effort. This is not unlike what is said in Schmidt (2006), which states that MDE is evolving toward DSLs combined with transformation engines and generators; in other words, MDE seems to be evolving toward MDA and DSM working together.

This is why we consider tools such as MetaSketch to be of utter importance to the industry, as MetaSketch reveals a genuine concern with adhering to OMG standards (which opens the door for its usage in MDA-oriented development scenarios) while also trying to address the metamodeling problem that we are facing today.

Another issue that we consider important to the success of metamodeling is complexity. The usage of standards is always conditioned by their complexity and how well adapted they are to the domain of interest. These points can be decisive factors over the difficulty of creating a model that correctly represents the problem (from the perspectives of syntax and semantics), which is where DSM differentiates itself. The fundamental issue is that developers and clients need to identify themselves with the metamodels they use; otherwise, they will look upon those metamodels as nuisances. An example can be seen in MOF, sometimes considered too complex for defining user

metamodels, because it includes concepts that would only be useful in the context of OMG-defined metamodels. This is why tools such as MetaEdit+ (with simple meta-metamodels) are gaining popularity throughout the developer community, and MOF/UML CASE tools (with complex meta-metamodels) are typically considered as only good for documentation and a last resort for code generation.

Finally, we consider that the evaluation framework defined in this article is quite relevant because it provides a good insight into the main problems that metamodeling tools would face: Its dimensions include support for language specification (syntax and semantics) and model transformations, which are essential to the creation of metamodels and models, as well as to obtaining new models in an automatic, MDE-oriented fashion.

CONCLUSION

Just as development paradigms changed and evolved over the last decades from assembly code to subsequent generations of programming languages, the development paradigm is changing from our current third-generation programming languages to a higher abstraction level. This shift is gradually happening as MDE is gaining importance as an abstraction mechanism over traditional programming activity.

However, tools need to follow and support this paradigm change. The only way that a modeling tool can effectively support the software developer's complex tasks is by providing metamodeling support: Such a tool should allow a software developer or architect to specify a language or metamodel and be able to automatically create tools that enable the creation of models based on that metamodel.

This article presented a framework for evaluating a tool's adequacy in the metamodeling activity. This framework defines some criteria that address both theoretical and practical issues in metamodeling and in modeling tools; nevertheless, it is still subjective and open to further refinement by adding more important criteria and by defining measurement metrics that can establish a higher degree of consensus regarding metamodeling issues.

After presenting the framework, we applied it to a small set of current modeling tools that we believe to be representative of the status of the mainstream MDE area. Finally, this article discussed some open research issues for metamodeling-based software development tools.

ACKNOWLEDGMENT

We would like to thank Leonel Nobrega for his promptness in supplying the latest version of his MetaSketch tool as well as all the documentation that was available at the time. We would also like to thank the reviewers of this article for all their excellent constructive suggestions to improve its quality.

REFERENCES

Atkinson, C., & Kühne, T. (2003, September-October). Model-driven development: A metamodeling foundation. *IEEE Software, 20*(5), 36-41. Retrieved June 5, 2006, from http://doi.ieeecomputersociety.org/10.1109/MS.2003.1231149

Atkinson, C., & Kühne, T. (2005, October). Concepts for comparing modeling tool architectures. In L. Briand & C. Williams (Eds.), *Model Driven Engineering Languages and Systems: Eighth International Conference, MoDELS 2005* (pp. 398-413). Springer. Retrieved June 23, 2006, from http://dx.doi.org/10.1007/11557432 30

Booch, G., Brown, A., Iyengar, S., Rumbaugh, J., & Selic, B. (2004, May). An MDA manifesto. *Business Process Trends/MDA Journal.* Retrieved June 15, 2006, from http://www.bptrends.com/publication-files/05-04COLIBMManifesto-Frankel-3.pdf

Fong, C. K. (2007, June). *Successful implementation of model driven architecture: A case study of how Borland Together MDA technologies were successfully implemented in a large commercial bank.* Retrieved November 23, 2007, from http://www.borland.com/resources/en/pdf/products/together/together-successful-implementation-mda.pdf

France, R. B., Ghosh, S., Dinh-Trong, T., & Solberg, A. (2006, February). Model-driven development using UML 2.0: Promises and pitfalls. *Computer,*

39(2), 59-66. Retrieved June 5, 2006, from http://doi.ieeecomputersociety.org/10.1109/MC.2006.65

Henderson-Sellers, B. (2005, February). UML the good, the bad or the ugly? Perspectives from a panel of experts. *Software and Systems Modeling, 4*(1), 4-13. Retrieved June 5, 2006, from http://dx.doi.org/10.1007/s10270-004-0076-8

Kelly, S., & Tolvanen, J.-P. (2008). *Domain-specific modeling*. Hoboken, NJ: John Wiley & Sons.

Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA explained: The model driven architecture. Practice and promise.* Reading, MA: Addison-Wesley.

MetaCase. (n.d.). *MetaCase: Domain-specific modeling with MetaEdit+*. Retrieved June 5, 2006, from http://www.metacase.com

Metamodel.com: Community site for meta-modeling and semantic modeling. (n.d.). Retrieved June 5, 2006, from http://www.metamodel.com

Moore, G. C., & Benbasat, I. (1991, September). Development of an instrument to measure the perceptions of adopting an information technology innovation. *Information Systems Research, 2*(3), 192-222.

Nobrega, L., Nunes, N. J., & Coelho, H. (2006, June). The meta sketch editor: A reflexive modeling editor. In G. Calvary, C. Pribeanu, G. Santucci, & J. Vanderdonckt (Eds.), *Computer-Aided Design of User Interfaces V: Proceedings of the Sixth International Conference on Computer-Aided Design of User Interfaces (CADUI 2006)* (pp. 199-212). Berlin, Germany: Springer-Verlag.

Schmidt, D. C. (2006, February). Guest editor's introduction: Model-driven engineering. *Computer, 39*(2), 25-31. Retrieved June 5, 2006, from http://doi.ieeecomputersociety.org/10.1109/MC.2006.58

Selonen, P., Koskimies, K., & Sakkinen, M. (2003). Transformations between UML diagrams. *Journal of Database Management, 14*(3), 37-55.

Siau, K., & Cao, Q. (2001). Unified modeling language: A complexity analysis. *Journal of Database Management, 12*(1), 26-34.

Silva, A., & Videira, C. (2005). *UML, metodologias e ferramentas CASE* (Vol. 2, 2nd ed.). Portugal: Centro Atlântico.

Söderstrom, E., Andersso, B., Johannesson, P., Perjons, E., & Wangler, B. (2002, May). Towards a framework for comparing process modelling languages. In *CAiSE '02: Proceedings of the 14th International Conference on Advanced Information Systems Engineering* (pp. 600-611). London: Springer-Verlag. Retrieved June 21, 2006, from http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=680389#

SparxSystems. (n.d.). *Enterprise architect: UML design tools and UML CASE tools for software development.* Retrieved June 5, 2006, from http://www.sparxsystems.com/ products/ea.html

Thomas, D. (2004, May-June). MDA: Revenge of the modelers or UML utopia? *IEEE Software, 21*(3), 15-17. Retrieved June 5, 2006, from http://doi.ieeecomputersociety.org/10.1109/MS.2004.1293067

Tolvanen, J.-P., & Rossi, M. (2003, October). MetaEdit+: Defining and using domain-specific modeling languages and code generators. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (pp. 92-93). New York: ACM Press. Retrieved June 5, 2006, from http://doi.acm.org/10.1145/949344.949365

Visual Studio 2005: Domain-specific language tools. (n.d.). Retrieved June 5, 2006, from http://msdn.microsoft.com/vstudio/dsltools

Zhao, L., & Siau, K. (2007, November). Information mediation using metamodels: An approach using XML and common warehouse metamodel. *Journal of Database Management, 18*(3), 69-82.

Zhu, J., Tian, Z., Li, T., Sun, W., Ye, S., Ding, W., et al. (2004). Model-driven business process integration and management: A case study with the Bank SinoPac regional service platform. *IBM Journal of Research and Development, 48*(5/6), 649-669. Retrieved November 23, 2007, from http://www.research.ibm.com/journal/rd/485/zhu.pdf

Jo˜ao de Sousa Saraiva was born in Macau, in September 1982. In September 2004 he joined the Information Systems Group (GSI) of INESC-ID, where he has since been developing skills in web application development and graphical specification of information software systems. In 2005, he graduated in Computer Science at Instituto Superior T´ecnico (IST), in Lisbon, Portugal. In 2007, he obtained an MSc in Computer Science from the same institution. He is currently a PhD student at IST, and a researcher at INESC-ID.

Alberto Rodrigues da Silva is professor of information systems at the Department of Computer Science and Engineering at Technical University of Lisbon (IST/UTL) Portugal. He is also a senior researcher at INESC-ID and director at the SIQuant Company. Rodrigues da Silva's professional and research interests are in modeling and metamodeling, model-driven engineering, requirement engineering, enterprise knowledge-based platforms, and CSCW and CASE Tools