Appeared in the Proceedings of the 8th International Conference MoDELS / UML 2005

Concepts for Comparing Modeling Tool Architectures

Colin Atkinson

University of Mannheim
atkinson@informatik.uni-mannheim.de

Thomas Kühne

Darmstadt University of Technology
kuehne@informatik.tu-darmstadt.de

Abstract. As model-driven development techniques grow in importance so do the capabilities and features of the tools that support them, especially tools that allow users to customize their modeling language. Superficially, many modeling tools seem to offer similar functionality, but under the surface there are important differences that can have an impact on tool builders and users depending on the tool architecture chosen. At present, however, there is no established conceptual framework for characterizing and comparing different tool architectures. In this paper we address this problem by first introducing a conceptual framework for capturing tool architectures, and then—using this framework—discuss the choices available to designers of tools. We then compare and contrast the main canonical architectures in use today.

1 Introduction

Given the growing interest in Model Driven Development (MDD), modeling tools are becoming an increasingly central and important element of software development environments. As a result, software project managers are increasingly faced with the issue of deciding what modeling tool(s) to use in a project and what role the chosen tool(s) should play. Until recently this was not an issue of great import because modeling has traditionally played a secondary, supportive role in software engineering. The primary artifact of software development has until recently always been code, leaving models, if used at all, to play the role of supporting, non-essential documentation. Even when models are used to generate code skeletons, as is often the case today, they are essentially viewed as accelerators of the coding process rather than as a part of the critical path of software development. However, if the vision of model driven development is even partially successful this situation will change and modeling will become the dominant, critical path activity in software development.

At present however there is no established way of characterizing and comparing the capabilities of modeling tools beyond a superficial comparison of feature lists. This makes it difficult to select a tool for a specific project on a serious technical basis. Without the availability of concrete comparison concepts and evaluation criteria, decisions for modeling tools will be more or less random and at best based on irrelevant or secondary properties.

The lack of a tool evaluation framework not only affects tool users but also tool builders. Unless tool builders are aware of all the architectural options available to them and are able to evaluate and compare their tool architecture against other alter-

natives, they will make their choices in a restricted design space, usually heavily influenced by tradition rather than by objective criteria. The problem is not that there is a lack of different (meta-) modeling infrastructure models or metaphors. On the contrary, quite a number of different approaches exist, such as the famous OMG four-layer architecture [1], powertype-based approaches [2], two-level approaches [3], Domain Specific Languages [4], and the orthogonal classification approach [5]. The problem is that each of these on its own is not a suitable basis for a tool evaluation framework. While each approach has certain advantages in its own right, none provides a general perspective for capturing the properties of a particular tool architecture. In fact, the very number of different notations and modeling metaphors compounds the problem of enabling an objective tool architecture comparison.

Unfortunately, even the venerable OMG four-layer architecture cannot serve as a reference architecture against which to compare the design of modeling tools. Not only is it the subject of much debate on what its different levels actually mean and how they are related to one another, it is also difficult to map it to other modeling metaphors. Furthermore, it is a high-level architecture and therefore does not lend itself to explaining or discriminating between architectures used in current tools.

Consequently, in this paper we provide a conceptual basis for describing and distinguishing different tool architectures. These concepts allow us to compare the main realization approaches in use today and to provide a reinterpretation of the OMG four-layer architecture which more precisely characterizes how it is implemented in most modeling tools. One of the main contributions of the paper is an enumeration and trade-off analysis of the architectural options tool designers should consider when developing a tool. These can be thought of as tool architecture patterns for tool developers. Finally, we analyze the advantages and disadvantages of some existing architectures in use today.

2 Conceptual Foundations

Before discussing the various architectures that can be used to realize modeling tools we first need to establish ways to precisely and exhaustively capture the associated design space.

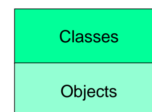

Fig. 1. Classification.

2.1 Types and Instances

The basic building block for constructing modeling tool architectures is the relationship between a type and its instances. This is not only the foundation for many metamodeling infrastructures, but also the foundation for the object-oriented implementation technology most widely used in mainstream software development today, e.g., that of Java.

Fig. 1 shows how we depict the relationship between types and instances. We use the concept of a classification frame split into two compartments—a type compartment and an instance compartment. To distinguish the type compartment from the instance compartment we draw the former with a darker shade of color than the latter and typically on the top or to the left. Note that in general one frame may have more

than two compartments, in which case the additional ones simply extend the classification hierarchy linearly. Between any two adjacent compartments we always have type / instance relationship.

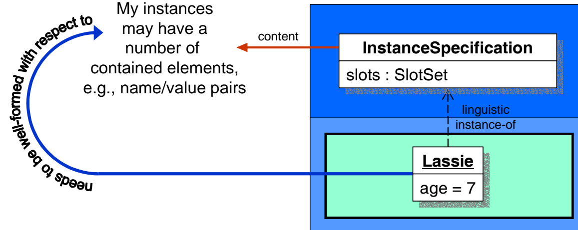

Fig. 2. Form Classification.

2.2 Form versus Content

To fully capture an architectural design it is insufficient to use just one general notion of "instance-of". An architecture presented in this way will admit many different interpretations and thus possibly allow consensus where there should be none. We therefore need to be more precise in order to explicitly distinguish between two fundamentally different kinds of instance-of relationships (see also [6] for a similar discussion).

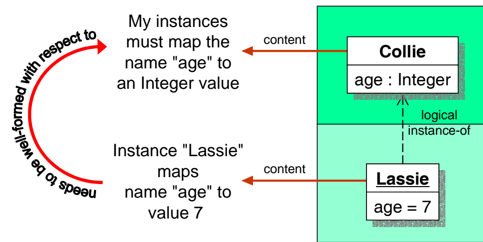

Fig. 3. Content Classification.

We refer to the kind of instance-of relationship used in Fig. 2 as "linguistic" instance-of. When it is used, the type (e.g., "Instance Specification") is part of a language definition and the instance (e.g., "Lassie") constitutes a language usage. Hence, we can check whether an element, (e.g., "Lassie") can be regarded as an instance of a *form*, e.g., "InstanceSpecification". The elements "age" and "7" of "Lassie" only need to be representable by the "Slot" classifier, i.e., be in a required *form* (e.g., "string" and "string" respectively). Whether e.g., "7" is an integer or not is irrelevant at this stage. Fig. 2 shows how we depict form-classification by *embedding* a frame within an instance compartment, and starting a new color scheme for the embedded frame. In the following, we will use form-classification to denote the representation format used to store elements, e.g., in a repository. The word "form" is used deliberately in the previous definition to distinguish this kind of "instance-of" relationship from the second kind of instance-of relationship which we refer to as "logical" instance-of (see Fig. 3).

Whether "7" needs to be of type "Integer" or an alternative type must be specified along the logical classification dimension (see Fig. 3). Here we may check whether the *content* (i.e., information expressed by "Lassie") can be regarded as an instance of the *content* expressed by "Collie". In other words, the type "Collie" contains information that is intended to define well-formedness rules which the content of instance "Lassie" must obey. We depict logical-classification by *stacking* compartments on top of each other. Thus, in summary, "form" and "content" are about the difference between *how* information is stored (form) and *what* information is stored (content). From now on, in contrast to Figs. 2 & 3, we will not use labels "linguistic" / "logical" for classification arrows anymore, because it will be clear from the frame notation (e.g., embedding) which kind is implicitly applicable.

2.3 Level Spanning

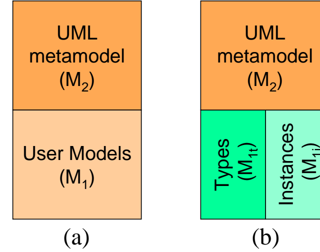

Fig. 4. Stacking versus Spanning.

Figs. 2 & 3 show two different ways of combining frames which we refer to as *embedding* and stacking respectively. In order to effectively capture all the level relationships that may occur in tool architectures, we need a third frame combination concept which we refer to as *spanning*. Fig. 4 shows an example of level-*spanning*, in terms of the OMG's classic four-layer architecture.

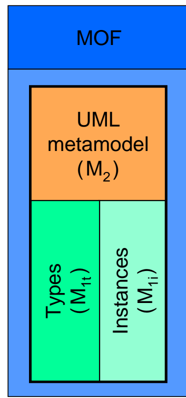

Fig. 5. Embedding & Spanning.

Fig. 4 (a) shows the usual depiction of the M_2 and M_1 levels in this architecture, where level M_1 is regarded as a monolithic level, even though it contains user instances (e.g., objects) and user types (e.g., classes), which are in a logical instance-of relationship to each other[1]. Fig. 4(b) makes this explicit by dividing level M_1 into two sublevels M_{1t} and M_{1i}. The reason for not *embedding* levels M_{1t} and M_{1i} within the instance compartment of frame M_2, is that we assume the contents of both M_2 and M_1 to be represented as MOF-data. Hence, we have only one representation format (MOF) and all three frames shown in Fig. 4(b) contain data that must be well-formed logically with respect to each other. However, none is the other's representation format.

The complete picture is depicted by Fig. 5, using embedding, spanning and stacking[2] to reinterpret the linear OMG four-layer design as an architecture in which the MOF is the common representation format for all other levels, the latter just establishing logical instance-of relationship with each other[3]. Note that the logical instance-of relationship from M_{1i} to M_{1t} is defined within M_2. In other words, level M_2 *spans* both levels M_{1t} and M_{1i}, meaning that elements from both levels must be well-formed with respect to the rules expressed in M_2.

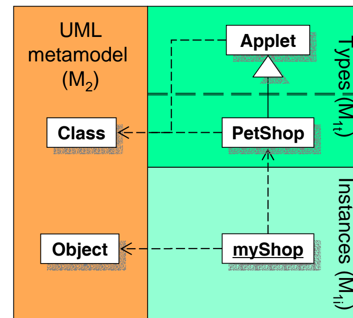

Fig. 6. Type Specialization.

2.4. Generalization

Classification is not the only way of deriving new elements from existing elements. Instead of differentiating an element by *instantiating* it from a *type* of another metalevel, it is sometimes more appropriate to *specialize* it using a *supertype*

Fig. 6 shows an example, where a "PetShop" class is defined to have "Applet" instances, by deriving it from superclass "Applet", as opposed to giving it a special

[1] We are referring to the corrected four-layer architecture, in which level M_0 is no longer part of the modeling stack, but represents the modeled system.
[2] One can think of Fig. 5 as a flat projection of a three-dimensional diagram.
[3] In section 4 we will further discuss possible interpretations of the four-layer architecture.

"Applet" property through instantiation. Depending on the purpose of the model, one of these alternatives might be more appropriate than the other, but both are available and a detailed architectural description technique must be able to distinguish and express both cases.

As we are typically not interested in individual element relationships when describing tool architectures, the typical use of a generalization layer to specialize from will be depicted in the way shown in Fig. 7. The generalization dimension is orthogonal to all other types of instance-of relationship kinds and may be used in any combination within a frame.

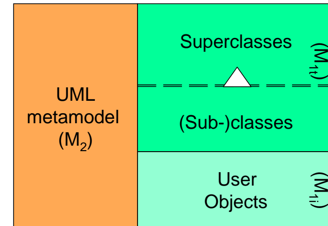

Fig. 7. Generalization Layer.

Note that the orientation of the frames carries no semantics and can thus be used to emphasize certain perspectives, such as the linguistic, logic or generalization dimension.

3 Architectural Options

Any tool architecture embodies a number of design decisions which directly or indirectly influence the challenge faced by the tool builders as well as the functionality available to tool users. The purpose of the following subsections is to make the respective design decisions more explicit and to provide a checklist to compare tool architectures against each other.

3.1 Number of Levels

One of the most basic choices to be made when designing a tool architecture is to decide how many type/instance levels it directly supports. A very common approach is to support two user modeling levels only. This is probably a vestige of traditional technologies such as databases (schema / data distinction) and object-oriented languages (type / instance distinction).

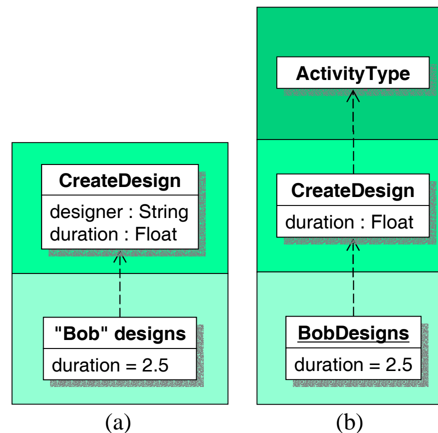

Fig. 8. DSL vs Domain Metamodel.

Even tools referred to as *meta*-modeling tools (e.g., MetaEdit+ [7]) often only support two user levels. Such tools allow users to first define a domain specific language (see Fig. 8(a), top part) and then build models using that new language (Fig. 8(a), bottom part). By allowing users to define their own languages they justify their name as "meta"-modeling tools, since the language definition is regarded as a model for the domain models. In other words, the language definition represents a (linguistic) model for models.

However, the existence of just two logical levels (Fig. 8(a)) already causes a problem for a tool based on a two-level implementation technology because it means that two logical levels have to be implemented within just one instance level. In general, one may even desire more than two logical levels: Fig. 8(b) demonstrates how users might want to model at three domain levels using the UML with a domain metalevel added on top of the usual instance and type levels. Such an additional metalevel is useful for making the class level dynamic as it is able to support the creation and deletion of classes even while the (modeled) system is running. Moreover, it lets one easily assign information to classes (e.g., whether an activity type appears in a certain workflow plan or not) by declaring corresponding attributes at the metalevel-types[4].

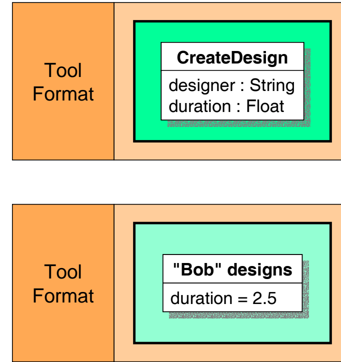

Fig. 9. Two Level Implementation.

Fig. 9 illustrates the above mentioned problem by showing an object-oriented (two-level) implementation in which the class level has to be used for defining the tool format in which the user data is modeled. The remaining object level then needs to represent the user models. Fig. 9 shows that both the user's domain specific language (DSL) *and* the corresponding user models must be represented within the tool. As none of the mainstream programming languages natively support more than two levels one cannot simply represent the user models in terms of the user language. However, creating user models only makes sense if the corresponding well-formedness rules are available at the same time. One way of having the models and rules available as data in the tool format is described in the next subsection.

The other approach uses code generation techniques to cast the information of the top part of Fig. 9 into a hard-coded, domain specific metamodel of a generated modeling tool. Fig. 10 depicts this process. Fig. 10(a) corresponds to Fig. 9's top part. Fig. 10(b) shows the architecture of the generated tool which is specifically tailored to deal with the user's DSL. Note that user models are directly represented in the format defined by the DSL definition.

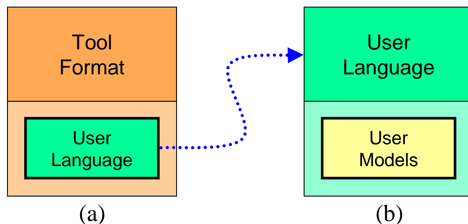

Fig. 10. Tool Generation.

The advantage of this generative approach is that any well-formedness rules governing the creation of user models are directly enforced by the underlying data structure. It is not necessary to write a generic checking algorithm which needs to be parameterized with the definition of the user's DSL. Also, such a tool offers an API for accessing and manipulating user models that is specifically tailored to the DSL used. An access method might thus be called "`getDuration()`" yielding a result of type "Float"

[4] Similar to the tagged value concept in the UML, but in more uniform way that simply extends the principles of the lower two levels.

instead of some generic access like "`getFeatureWithName('duration')`" yielding a result with a generic type, e.g., of type "String". A generated tool will also be very efficient in dealing with user models, as all the generated code will be specific to the DSL defined and will have been compiled.

The disadvantage of this generative approach is that it is not possible to use a single tool to work on several levels (language definition + language usage) at once. Especially in early phases, when the DSL is still being defined, it is very convenient to switch back and forth between the levels without going through a change-generate-compile-validate cycle every time. An interpreted language, such as Java, which allows compilation of new code to be done in the background and supports reloading of the new code into the running tool, blurs the boundaries between a generative and an integrative approach from the point of view of the user of the tool.

Another disadvantage of two-level based tools is that they potentially cannot use a user domain model directly as input for a new DSL. In other words, it may not be possible to conveniently use such a tool repeatedly in order to create a cascade of definition-usage pairs, thus creating a (meta-) modeling stack (e.g., MOF ← UML ← Classes ← Objects). The only way for tools with such a limitation to support more than two levels is "level compaction".

3.2 Level Compaction

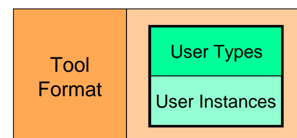

Fig. 11. Logical Stacking.

An alternative way to support multiple modeling levels with just one instance level is to abandon the idea that one modeling level (e.g., user classes) automatically defines the representation *format* for the level below (e.g., user objects). Instead, the native tool representation format is used for both user modeling levels. Fig. 11 shows how the situation of Fig. 9 can be resolved by keeping both datasets in the same tool, *stacking* them on top of each other.

With *stacking* we express the fact that one level ("User Types" in Fig. 11) controls another level ("User Instances" in Fig. 11) but not by being its format definition but by specifying the rules that it's controlled level must obey. In other words, the tool needs to look up data in the controlling level in order to check the data in the controlled level with respect to well-formedness. The scheme in Fig. 11 can easily be extended to include another level (above "User Types") in order to support a user domain metalevel and hence enable modeling as illustrated in Fig. 8(b).

The architecture shown in Fig. 11 can be extended not only by increasing the innermost stack, but also by using *spanning*. In this way, a tool can be promoted from being specialized for one language (e.g., UML) only, to supporting many user-definable languages. Fig. 5 shows how spanning can be used to build such a (MOF-based) UML tool.

The advantages of an integrative, level-compaction approach are manifold: User instance data can be manipulated independently of user type data. This allows for unlimited freedom in experimentation with what e.g., user domain models should look like. Note that a generative approach (Fig. 10) only allows domain models that adhere to the rules of the user DSL. When the DSL is changed the models formerly created with it are in an outdated format. In contrast, in an integrative approach—although the

user instances will no longer conform to the DSL—there will be no need to migrate them to the new format. No representation change is ever needed as long as all levels that may change are in a logical content-controlling relationship with each other.

From the point of view of tool builders, levels belonging to the same level stack can be treated in a uniform way. Multiple-level support only needs to be provided once and can then simply be scaled up to support any number of levels. Levels belonging to the same representation format can be treated uniformly with respect to many operations, such as serialization to output formats.

Another important difference introduced by level compaction is the fact that a tool builder no longer has to replicate model data. Fig. 10 makes it clear that a cascading approach necessitates model data to be stored twice: Once as instance data (e.g., “User Language” in Fig. 10(a)) and another time as type data (“User Language” in Fig. 10(b)). Level compaction uses the same set of data for both purposes at the same time (see, e.g., “User Types” in Fig. 11).

A potential disadvantage of level compaction is that access and modification of the supported levels has to occur in a generic manner, i.e., all levels are treated the same and thus the advantages of level-specific APIs are lost. Yet, this need not necessarily be the case. It is of course possible to provide special views onto each of the levels, by using adapters, for example, so that APIs can be made available that are identical to those of a two-level cascading approach.

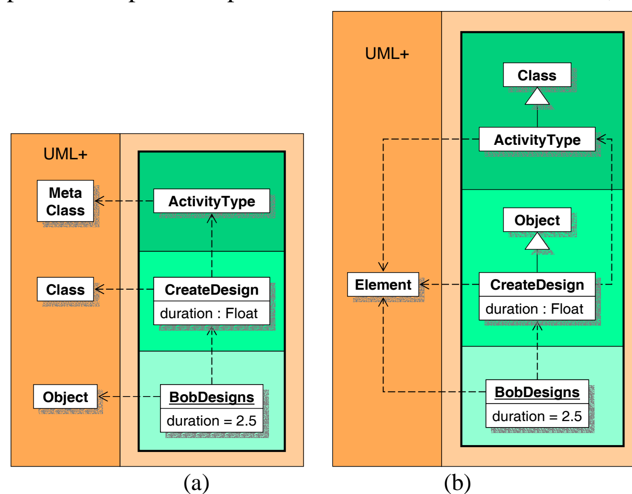

Fig. 12. Language vs Library Metaphor.

3.3 Language versus Library Metaphor

The previous section demonstrated how level compaction can be used to move control from the format-language to a logically controlling language. However, the issue of whether one supports multiple levels within one instance level (level compaction) is orthogonal to whether one uses a very liberal format language or not. A modeling tool with a built-in UML metamodel (see Fig. 12(a)) is an example of the simultaneous use of both level compaction (for user types & instances) and a confined language space. In order to make our next point more clearly we have added a domain metalevel and hence named the corresponding metamodel “UML+” instead of just “UML”. As can be observed from Fig. 12(a), user elements are controlled by two dimensions: First, their *form* must conform to the UML+ metamodel (through linguistic instantiation). Second, their *content* must conform to the next logical level higher up in the stack. The top level of the stack is

not content controlled in any way and just needs to obey the format rules imposed by the UML+ metamodel.
As discussed in the previous sections the approach shown in Fig. 12(a) has some trade-offs: Users may only model within the limits of the language defined at the UML+ level. This may be regarded as an advantage (in order to enforce a standard) or a disadvantage (since it is then impossible to use any kind of concept). Hence, any language extension will have to be accomplished by altering the built-in language metamodel "UML+". This is a direct consequence of using, what we call the *language metaphor* for defining valid syntax for user models. Even if the UML+ metamodel were kept as modifiable data, one still needed to perform language metamodeling and, thus, alter the modeling language standard when trying to create more domain specific models. This is of course the reason why the UML language designers chose to introduce stereotypes as a "lightweight" way of metamodeling. Hence, stereotypes represent another, third way of supporting one more level, in addition to "two-level cascading" (section 3.1) and "level compaction" (section 3.2). Note however the difference in providing a domain metamodel (as in Fig. 8(b)) versus allowing (strictly limited) extensions to the language definition (using stereotypes).

Fig. 12(b) demonstrates an alternative to the *language metaphor* which we refer to as the *library metaphor*. In comparison to Fig. 12(a), the language definition has been reduced to a bare minimum. User elements are not distinguished by their form classifier anymore (e.g., Class or Object), but by an assigned level number. They are not differentiated by creating them from special form-classifiers (e.g., UML+ element "Object"), but by controlling them with a special content-classifier (e.g., user type-level element "Object"). Typically, this control will occur indirectly, as the example in Fig. 12(b) demonstrates: Element "BobDesigns" is only indirectly controlled via "Object", being much more tightly controlled by "CreateDesign".

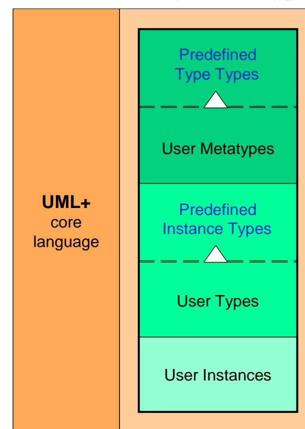

Fig. 13. Generalization Layer.

Fig. 13 gives an architectural view of this approach where the original language definition is split into a minimal *core* part and a number of predefined elements located at logical levels. The *library* part of the control over user models is hence distributed over the logical levels, depending on what user-model level the respective elements control.

Note that the two elements labeled "Object" (or "Class" respectively) in Figs. 12(a) and 12(b) are not identical. They not only differ with respect to their location in the architecture but also with respect to the way in which they control elements. Element "Object" in Fig. 12(a) enables its instances (e.g., "BobDesigns") to have a certain *form*. Element "Object" in Fig. 12(b) does not need to do that as this is already accomplished by "Element" at level "UML+", albeit in a much more generic way. Element "Object" in Fig. 12(b) restricts this genericity by exerting content-control over, e.g., "BobDesigns" yet this control is considerably strengthened by element "CreateDesign". The latter will be much more specific about the allowed properties of "BobDesigns" as any of the "Object" elements of Fig. 12(a) or 12(b) could ever be.

Note that element "Object" in Fig. 12(a) represents a tool builder's perspective and will support operations for model management. In contrast element "Object" in Fig. 12(b) may contain operations of relevance to the modeling tool user, such as "`equals()`" for comparing objects based on domain principles, instead of model management principles.

The advantages of using the library metaphor to controlling user models are:

- a simplified core language definition allowing experimentation with model concepts at all logical modeling levels,
- a stable core language definition even in the event of users wishing to extend their "language", and, hence,
- maximum flexibility for users with respect to domain specific modeling.

If the predefined libraries (see Fig. 13) are made immutable and fixed, this flexibility is even reconciled with the desire to retain a common core standard modeling approach, which may only be extended but not completely redefined.

The reduction of the core language to a minimal set of features can be compared to reducing the BNF definition of a programming language's syntax to a bare minimum and letting all removed rules (such as the difference between arithmetic and Boolean expressions) be enforced by static semantics checking. This makes the syntax definition more immune to changes to the language definition at the cost of shifting the change-burden to the definition of the static semantics (the library in our example). The library metaphor has indeed proven to be very successful for languages such as Smalltalk and Java which have a rather small language definition and provide the bulk of their utility through the availability of standardized libraries.

The disadvantages of the library metaphor is the unfamiliarity of the approach to most users and the need for creating machinery that deals with all possible logical levels generically. In particular one needs to implement a generic well-formedness checking algorithm to be applied to a level by parameterizing it with the content of the level above. However, tool builders then only need to define the basic principles of modeling, such as instantiation, specialization, and association once in the core language. These will work uniformly for all levels and there is no need for tool builders to use different checking algorithms for different level crossings or replicate the basic mechanisms time and again so that they are available to the next level. This replication is typically unnecessary, unless one specifically desires these features to work differently for each language level incarnation[5].

The next question to address with respect to tool architectures is therefore the choice of the appropriate number of linguistic levels.

3.4 Language Definition Stack Depth

The use of specialization, rather than instantiation, can also be put to use in the core language definition (in the linguistic dimension). Fig. 14 shows a very rough conceptual sketch of how the Fujaba [8] metamodel is composed of several specialization layers. Instead of creating a language definition stack in the sense of

[5] The MOF and the UML represent a typical counter-example. Here, one desires as much as uniformity between the UML core and the MOF as possible.

"MOF ← UML ← UserModels", the Fujaba developers opted to have a number of languages which refine each other, as opposed to being instantiations of each other. In this way, they have built up the resulting metamodel, step by step, and have alternative views (e.g., as AbstractSyntaxGraph elements or UML elements) on the same set of user data.

The design shown in Fig. 14 of course begs the question as to why the OMG has not opted to cast MOF as a *super*-model, i.e., use generalization rather than a classification, on top of languages such as UML or CWM?

The purpose of the MOF is to provide a common basis for defining all other OMG languages. One way to provide such a common basis is to define a language that *classifies* all the languages one is interested in, as is done by the MOF in its M_3-level role. The more diverse the set of languages to be captured under a common umbrella, the more linguistic levels are useful. At each language definition level, more languages fitting into the paradigm currently addressed can be properly described, a process that continues to the very top of the language stack. In the OMG's case we just have a language describing all user models (the UML metamodel) and another language on top of this (the MOF), describing object-oriented approaches to modeling. This makes sense if one is interested in a standardized meta-meta-*language* for creating metamodels (such as the UML) and providing the corresponding tools along with this capability.

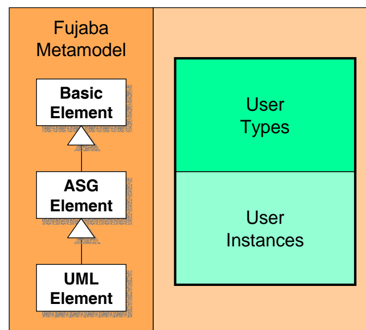

Fig. 14. Language Layers.

However the same effect, and more in this example, can be achieved by using a standardized *library* of metamodeling superclasses. Instead of specifying the element "Component" to be a "Class" (in contrast to, e.g., a "Data Type") by assuming it to be an *instance* of a M_3-level MOF-element "Class", it could also be differentiated as such by letting it *subclass* from an M_2-level element "M-Class". In this way, "Component"-usages would still be different to other UML concept usages, and they would immediately be accessible through this "M-Class" interface. We use the prefix "M-" (for MOF) in order to distinguish this element from the ordinary M_2-level UML element called "Class". In other words, the desired repository access to elements in user models can directly be achieved through corresponding metamodel superclasses. A double role as played by the MOF (as a M_3-level meta-metamodel & as a general repository format for all levels) would therefore not be necessary.

Note, however, that the above described library approach in the linguistic dimension only works if one is able to find a (MOF-)*super*-model for all the language defining metamodels (such as UML and CWM) that one would like to include. It is the distinguishing advantage of using a classifying language (as the MOF in its M_3-level role) that it can abstract from the metamodels to be captured, without requiring them to share a common (super-)structure.

Summarizing, through "level compaction" and/or using repository superclasses in a language defining metamodel, it is possible to remove language definition levels in the linguistic dimension. Fig. 13 shows an extreme case, where one could do away

with a MOF format as well and integrate other modeling approaches, such as CWM, as modeling libraries within the logical levels.
The appeal of a minimal length language stack (in the linguistic dimension) is the simplicity of the associated architecture and the resulting lack of redundancy. All levels can be treated uniformly and neither data nor basic modeling principles have to be replicated.

In favor of a language stack with two or more levels it can be noted that each language introduced makes the associated storage format more concrete and more tailored to the paradigm one aims to cover. Hence, the representation can be more compact and easier to read and write for both humans and tools.

4 Canonical Architectures

We will now use the concepts, notation, and architectural options previously introduced to characterize and evaluate the three main canonical architectures currently underpinning modeling tools. This is not intended to be an exhaustive characterization, but to layout the major reference architectures against which other more specialized architectures can be compared.

4.1 Four-Layer Architecture

Certainly today's most prominent architecture for metamodeling infrastructures or tool designs is the OMG's four-layer architecture (see Fig. 15(a)).

Since this architecture is not unambiguously specified we can only offer interpretations of it. One alternative, visually suggested by Fig. 15(a), is a logical language stack of "MOF ← UML ← M_1"[6], but that would neglect the MOF's role as a repository format for all the levels. However, just casting the MOF as a pure repository format would neglect the MOF's role as a logical language definition for the UML metamodel at M_2., Fig. 15(b) therefore best seems to capture the apparently intended dual role of the MOF and hence best captures the spirit of the whole architecture. Note that it explicitly shows the MOF's ability to represent itself.

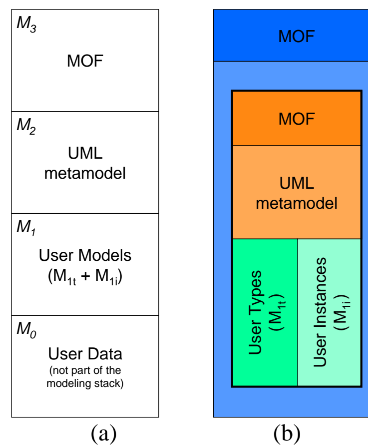

Fig. 15. Four-Layer Architecture.

A non-technical but nonetheless very real advantage of the four-layer architecture is that it defines a standard, including standard implementation technologies. It furthermore allows several modeling standards such as the UML and CWM to be fitted under one (MOF-based) architecture.

[6] We are using M1 as a shortcut for M1t and M1i combined, in part because the OMG does not explicitly distinguish between M1t and M1i.

Its main drawback is the lack of support for more than two user modeling levels. While the architectural style does not prevent an extension of the user modeling levels (within M_1), the standardized UML metamodel restricts them to two. Although the UML's solution for providing a language extension feature to modelers—the stereotype mechanism—has been improved from version 1.5 to 2.0, it still does not offer the same power for user domain metamodeling as another user modeling level would offer (as exemplified in Fig. 8(b)).

4.2 Two-Level Cascading

The popularity of the two-level cascading approach is testified by the many practical examples of its use. Fig. 16(a) informally depicts the approach of providing a format for creating user defined languages and then, after a generation step, using the user language definition to create user models.

Fig. 16(b) uses our notation to more precisely capture the promotion of the "User Language" instance data to "User Language" types that then can be used to create models. Tools such as MetaEdit+ [7] and Fujaba [8] use this approach. Also the MDR approach using JMI technology [3] and the Software Factories approach [4] use the same underlying principle.

The advantages of this approach are:
- the efficiency of the generated modeling facilities.
- the specificity of the API for accessing user models.
- the fact that metacase tool vendors may produce metamodels for their customers and only ship a generated tool, without giving away the corresponding meta-model data as well.

Its disadvantages are:
- the need to replicate the definition of basic modeling primitives, such as instantiation, specialization, etc. time and again.
- the need to duplicate model content by keeping it both as user instance data (for manipulation during the language definition phase) and as tool type data (for creating user models).
- an inconvenient "edit-generate-compile-validate"-cycle when developing the modeling language (e.g., a DSL).

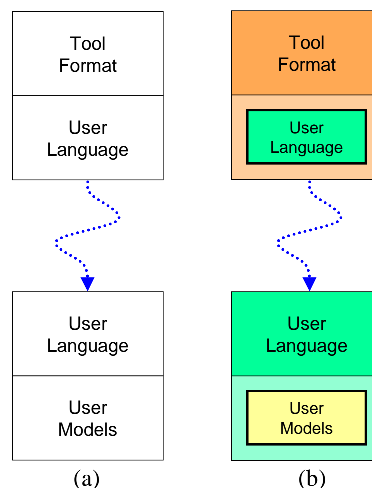

Fig. 16. Two-Level Approach.

4.3 Orthogonal Classification Architecture

Perhaps the antithesis to the two-level cascading approach described above is the so called orthogonal classification architecture (OCA) based on level-compaction [5].

Fig. 17(a) shows the two (linguistic and ontological[7]) dimensions of this approach featuring just one format level (L_1) used for representing an unbounded number of ontological levels. Although the OCA does not dictate any particular number of linguistic or ontological levels, it lends itself to be used with a single (MOF-like) universal format and an unbounded number of user domain modeling levels based on the library metaphor (see section 3.3 and Fig. 17(b)). A tool with this architecture as its basis is ConceptBase [9]. The one format level in ConceptBase is based on the Omega level of Telos [10]. Any other modeling data in ConceptBase is expressed as instances of this one "format" level. In ConceptBase terminology all model data is expressed as propositions.

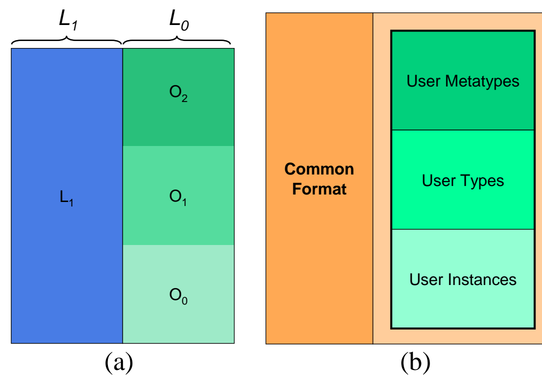

Fig. 17. Orthogonal Classification Approach.

The advantages of the OCA are:

- the complete uniformity with which all ontological levels can be treated. One does not need to consider various kinds of level boundaries except logical stacking.
- the completely redundancy-free storage of modeling data. No single level has to be represented twice so as to use it in two roles.
- a single tool can be used to manipulate all levels in the same manner. There are no limits to experimenting with content in levels since the basic representation format virtually allows unlimited expressiveness. Well-formedness conformance to a higher logical level, of course, is a different matter and may also need to be supported. Locking mechanisms could be used to prevent users altering data in levels they are not supposed to change or even see.

Its disadvantages are:

- the unfamiliarity of the library approach to the majority of modelers
- the fact that current established technologies and market rules are better suited to standardize languages, rather than libraries.

5 Conclusion

As model-driven development gains popularity, supporting tools are becoming an increasingly important part of software development. The internal architecture of such tools is not only of concern to tool builders but also to tool users since it determines the basic functionality available. Unfortunately, at present there is no framework for characterizing and evaluating such architectures, as previous work on clarifying

[7] For the purpose of this discussion we can equate "ontological" with "logical" instantiation.

metamodeling infrastructures has never attempted to include tool representation issues. In this paper we have laid the foundation for such a framework by introducing concepts—including the as yet undistinguished "embedding" and "spanning"—to capture core architectural elements. Using this framework we then discussed the architecture design space and outlined the main canonical architectures in use today.

At one end of the spectrum there is the "Two-level Cascading" approach which supports multi-level modeling technology in terms of classic two-level object-oriented technology. At the other end there is the "Orthogonal Classification Architecture" which provides a genuine multi-level modeling platform, typically in the context of a single linguistic format definition. In between these two extremes, various combinations may be applied to achieve different balances between their pros and cons, as exemplified by the OMG's four-layer architecture.

We believe that an evaluation framework for tool architectures, allowing concrete technical comparisons to be made will be an invaluable help for making strategic decision in the near future, and we hope that our contribution in the form of this paper represents a useful step in this direction.

Acknowledgements

We would like to thank Andy Schürr and his group and Pierre-Alain Muller for stimulating discussions and for information on Fujaba, and TopModL respectively.

References

[1] OMG: Unified Modeling Language, v1.5. OMG document formal/03-03-01, (2003)
[2] Gonzalez-Perez, C. and Henderson-Sellers, B.: Templates and Resources in Software Development Methodologies. To appear, Journal of Object Technology, May/June (2005)
[3] Matula M.: Netbeans Metadata Repository. http://mdr.netbeans.org/ (2003)
[4] Greenfield, J., Short, K.L., Cook, S. and Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Hungry Minds Inc. (2004)
[5] Atkinson, C., Kühne, T., Model-Driven Development: A Metamodeling Foundation. IEEE Software, vol. 20, no. 5 (2003) pp. 36-41
[6] Bézivin, J., Gerbé, O.: Towards a Precise Definition of the OMG/MDA Framework. Proceedings of ASE'2001, San Diego, USA (2001)
[7] Kelly, S., Lyytinen, K. and Rossi, M.: MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment. In Proceedings of the 8th International Conference CAISE'96, Springer-Verlag (1996) pp. 1-21
[8] Klein, T. Nickel, U.A., Niere, J., Zündorf, A.: From UML to Java And Back Again, Tech. Rep. TR-RI-00-216, University of Paderborn (1999)
[9] Jeusfeld, M.A. et al.: ConceptBase: Managing conceptual models about information systems. Handbook of Information Systems, Springer-Verlag (1998) pp. 265-285
[10] Mylopoulos, J., Borgida, A. Jarke, M. Koubarakis, M.: Telos: representing knowledge about information systems. Vol. 8. No. 4, ACM Trans. on Information Systems (1990)