THE ARCHITECTURE OF A UML VIRTUAL MACHINE

Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, Nosa Omorogbe

SKYVA International
One Cabot Road
Medford, MA 02155, U.S.A.
++ 1 781 306 7600

Object Oriented, Ltd.
Kramgasse 5
6004 Luzern, Switzerland
++ 41 41 418 7071

dirk@riehle.org, sfraleigh@skyva.com, dirk@bucka-lassen.dk, nomorogbe@skyva.com

ABSTRACT

Current software development tools let developers model a software system and generate code from the models to execute the system. However, generating code and installing a non-trivial system induces a time delay between changing the model and executing it that makes rapid model prototyping awkward if not impossible. This paper presents the architecture of a virtual machine for UML that interprets UML models without any intermediate code-generation step. As its main contribution, the paper shows how to embed UML in a metalevel architecture so that a key requirement of model-based systems, the causal connection property between models and model instances, is always guaranteed. With this architecture, changes to a model have immediate effects on its execution, providing users with rapid feedback about the model's structure and behavior. This approach supports model innovation better than today's code-generation approaches.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming—metalevel architectures; D.2.2 [**Software Engineering**]: Design Tools and Techniques—Evolutionary prototyping, Rapid prototyping, Object-oriented design methods; D.2.11 [**Software Engineering**]: Software Architectures—Languages, UML; D.3.3 [**Programming Languages**]: Language Constructs and Features—Classes and objects, Frameworks; D.3.4 [**Programming Languages**]: Processors—Code generation, Interpreters, Runtime environments, Virtual machines.

General Terms

Design, Languages.

Keywords

Metamodeling, Causal connection, UML virtual machines.

1 INTRODUCTION

Traditionally, object-oriented modeling languages like UML and OPEN have served to describe the design of a software system [27, 18]. The implementation of the system is carried out as a separate frequently time-consuming step, in which detail is added to the design-level models. The separation of design from implementation poses a significant problem, because implementing a system can take a long time and its design and implementation can easily get out of sync. Long time-to-market, missing documentation, and hard-to-change systems are the consequence.

With the industry-wide adoption of UML, a new breed of software development tools is gaining prominence. These tools generate code directly from the models. Users of a tool ideally model their domains using UML and then publish the models in the form of generated code into a runtime system. The runtime system connects the code with its environment, for example, databases or web-servers.

Model-driven code-generation has several advantages over the traditional approach, including:

- *Shorter time-to-market.* Users model their domains rather than implement them. A modeling language like UML is better suited to express domain models than a programming language like Java or Smalltalk.
- *Increased reuse and fewer bugs.* The tools hide the details of how the models are hooked up into the runtime system, freeing users from knowing intricate details about used frameworks or system components.
- *Easier-to-understand system and up-to-date documentation.* Because design and implementation are always in sync, so is the documentation. The system is easier to understand and better documented.

However, code-generation does not solve all the problems. In particular, it has the following drawback:

- *Delay between model change and model instance execution.* Generating code from models, compiling this code, shutting down the existing system, installing and configuring the new system, and starting it up can take from minutes to hours.

This time delay makes exploration and simulation of new models with immediate user feedback awkward if not impossible, thereby significantly hindering the innovative exploration of the model solution space. The resulting models easily become sub-optimal.

This paper presents the architecture of a virtual machine for UML. The virtual machine represents the modeling language (UML), the models described using UML, and the model instances as first-class entities. For executing a model, the virtual machine instanti-

OOPSLA '01, October 14-18, 2001, Tampa, Florida.

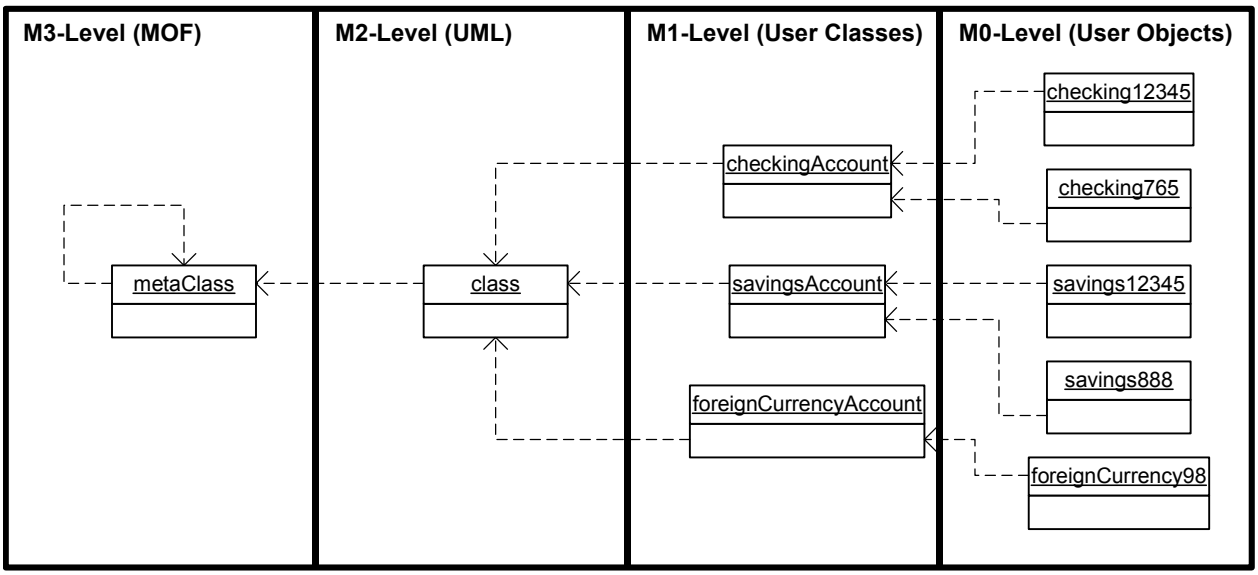

Figure 1: Four-level modeling architecture with Bank Account example. (Dotted lines, right to left, show an instance-of relationship.)

ates and interprets the model according to UML semantics. With all models on all levels being explicitly represented, changing a system's model leads to immediate (controlled) effects on the running system. As a result, the system provides short feedback cycles and allows for simulation and rapid exploration of model variants, better supporting innovation than possible with code-generation approaches.

The virtual machine has a logical architecture that is based on the UML four-level modeling architecture and a physical architecture that realizes the logical architecture as an object-oriented framework [23]. The logical architecture is a metalevel architecture, and the physical architecture implements it. As the paper's main contribution, we discuss how the logical and physical architecture fulfill the causal connection requirement that seamlessly and recursively integrates a model with its instances. Thus, our discussion focuses on the structural aspects of the architecture, leaving a discussion of the behavioral aspects to further papers. We discuss the changes we applied to the UML specification and the enhancements that we added to UML to overcome its limitations.

The architecture of the virtual machine has been explored in two projects at UBS and has been consolidated as a core piece of the flagship product of SKYVA International. The virtual machine is part of a system that combines a modeling environment with a model execution environment, much like CLOS, Smalltalk, or Self systems provide both a programming environment and a program execution environment [19, 7, 28]. We report about our experiences of using SKYVA's system in industry projects.

Section 2 reviews the architecture of UML-based systems in general. Section 3 presents the architecture of the virtual machine and how it realizes the requirements set up in Section 2. Section 4 reviews key implementation aspects. Section 5 reviews changes and additions we have applied to the UML specification. Section 6 then reports about our experiences with the virtual machine. Section 7 finally lists related work and Section 8 shows how this work will proceed further and what challenges we see ahead.

2 MODEL-BASED SYSTEMS

The UML specification defines four logical levels of modeling, called the M0, M1, M2, and M3-level (M stands for model) [27, 16]. These four levels define the logical architecture of any UML-based system and therefore form the requirements for a virtual machine that is capable of representing and executing such a system.

- The M0-level contains objects that represent the currently running system. These objects are also called user objects or domain objects.
- The M1-level contains objects that represent a model of the currently running system. These objects are also called user classes or domain classes.
- The M2-level contains objects that represent the modeling language, in our case UML. This level is also called the metamodel level.
- The M3-level contains objects that represent the language in which UML is represented. This level is the also called the meta-metamodel level.

Every level provides the means to describe the next lower level, so the M3-level is used to describe the M2-level (UML), the M2-level is used to describe the M1-level (system models), and the M1-level describes the M0-level (running systems). In theory, the number of levels is unbounded, but for most practical purposes, four levels are sufficient.

2.1 Causal Connection

The four-level architecture is an object architecture that helps us understand a system's logical object structure (as opposed to its physical architecture, see Section 3). Effectively, the logical architecture serves as a high-level explanation of how objects relate to each other (which object models which other object; which object is an instance of which other object).

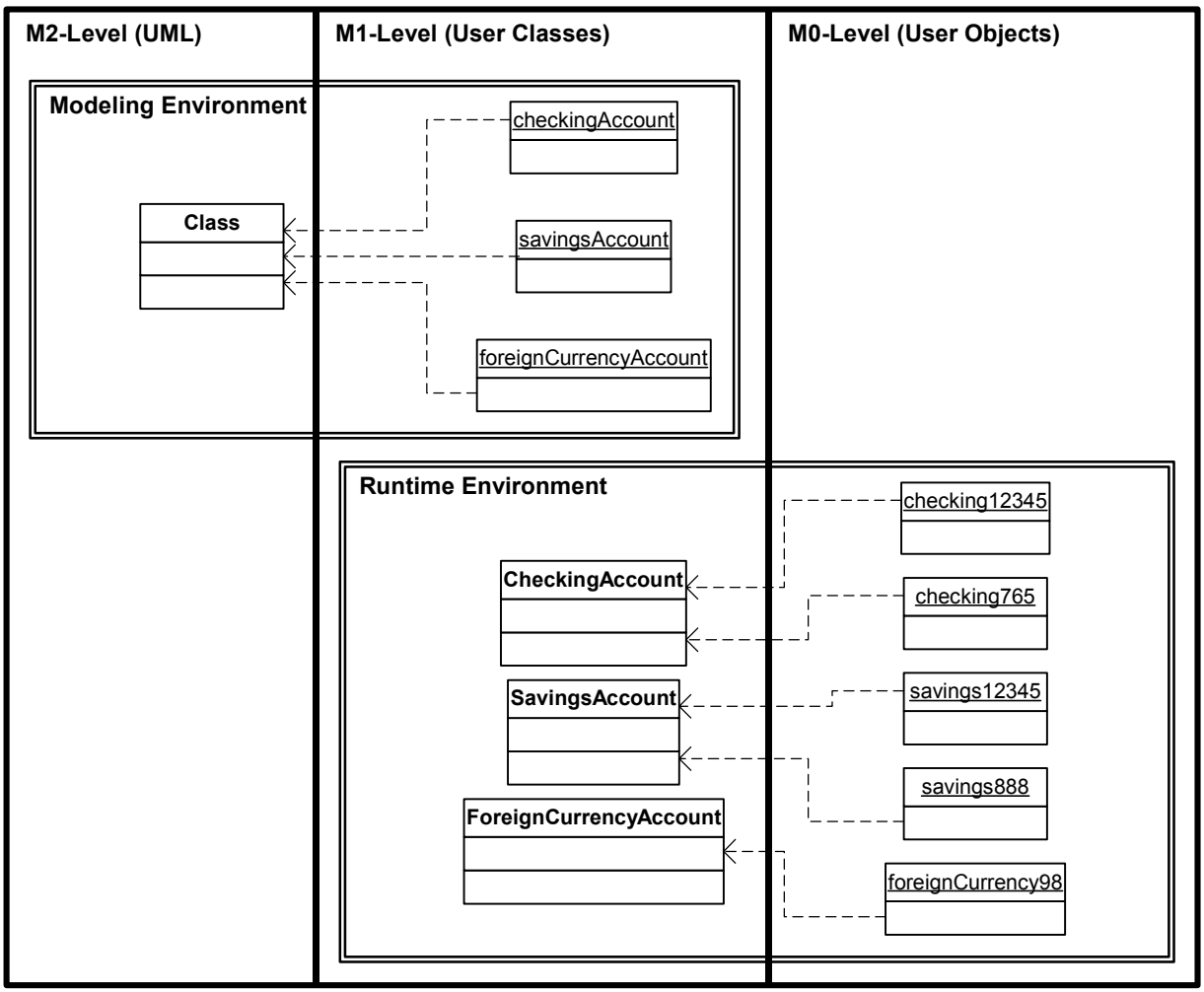

Figure 2: Separation of modeling from runtime environment in code-generation approaches.

Figure 1 displays these four levels and an example. The examples are bank accounts. Each rectangle represents an object (using UML as the design notation). The dependency arrows, from right to left, indicate a logical "instance-of" relationship.

During development, software developers use UML objects (M2-level objects) to define user classes (M1-level objects). In the example, developers define three types of bank accounts, Checking-, Savings-, and ForeignCurrencyAccount. At run-time, the user classes are instantiated, displayed in Figure 1 as the user objects checking12345, savings12345, etc. These user objects are M0-level objects.[1]

For a system of this architecture to be in a valid state, we define the *causal connection* property, as known from metalevel architectures [29]:

DEFINITION: CAUSAL CONNECTION

A modeling level is *causally connected* with the next higher modeling level, if the lower level conforms to the higher level and if changes in the higher level lead to according changes in the lower level.

In a model-based system, in which modeling levels are causally connected, changes to a model cause the structure and behavior of all model instances to change accordingly.

2.2 Code-generation Approach

Current software development tools use a code-generation approach to causally connect modeling levels. Typically using graphical editors, M2-level objects (UML classes) are instantiated to give users M1-level objects, representing domain classes. For example, inside such a software development tool, the checking-Account object logically represents what users of the tool perceive as their CheckingAccount class. Users of such tools design a domain model consisting of M1-level objects, which are instances of the M2-level UML classes.

[1] M0-level user objects are different from M1-level Instance objects. Instances of the UML Instance concept add to the specification of an M1-level class, typically by participating in an illustrating scenario of how instances of the class behave. However, they do not directly represent an instance of the class.

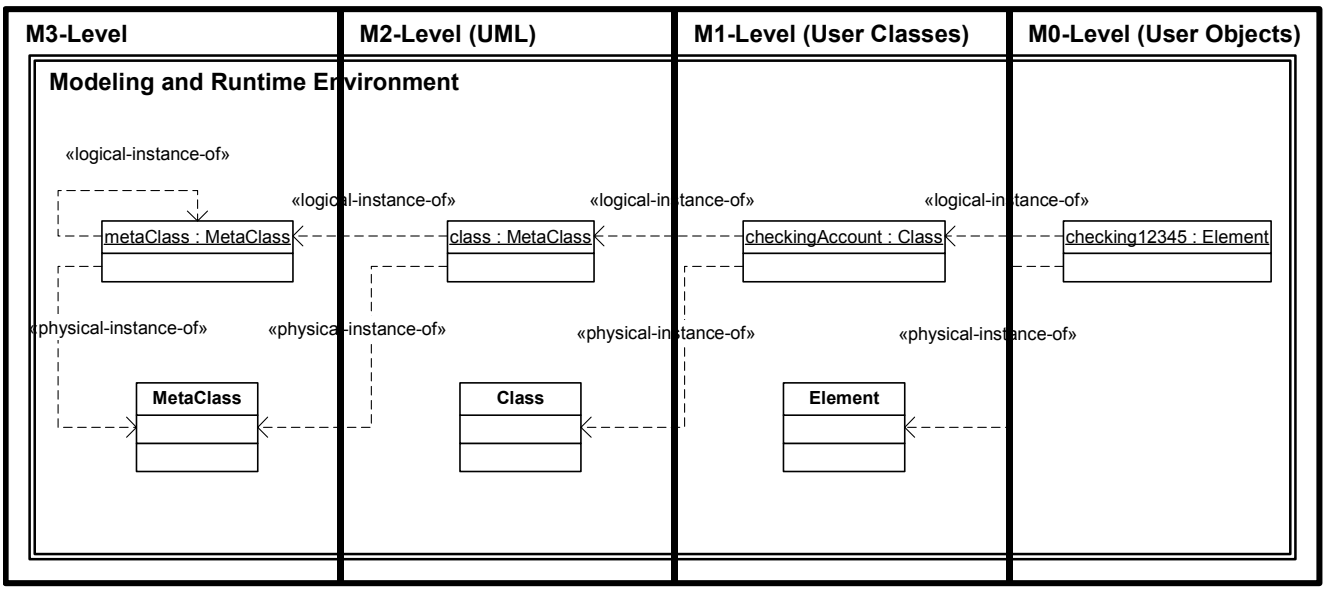

Figure 3: Convergence of modeling and runtime environment.

To create and handle user objects like checking12345, the tool creates a programming-language-level class for each M1-level object. The checking12345 object becomes an instance of the programming-language-level class corresponding to the checkingAccount object. Typically, there is a one-to-one correspondence between a model-level class and a programming-language-level class or interface [8]. This code-generation approach fully separates the modeling-language M1-level objects from the M0-level objects. This separation corresponds to the two types of environments in which the objects are handled: M1-level objects solely exist in a modeling environment, and M0-level objects solely exist in a runtime environment.

Figure 2 illustrates this separation. M1-level objects in the modeling environment are mapped on the corresponding M1-level classes of the runtime environment. The causal connection between model and model instances is maintained by the code-generator. Only if the modeling environment generates code, do the model changes carry over into the runtime environment, leading to the execution of the model instances according to the changed model.

A single round-trip between the modeling environment and the runtime environment can take several minutes if not hours. For non-trivial systems, the delay between model change and model execution makes rapid exploration of new models impossible, because the time delay between model change and execution is too long. Users, who may have to wait for hours until a model change becomes executable, tend not to explore a wide range of model options but go the easiest possible path, not exploring possibly better alternatives.

2.3 Virtual Machine Approach

An interpreter approach avoids the intermediate step of generating M1-level classes out of M1-level objects. Rather, the runtime system directly interprets the M1-level objects. All objects exist in the same memory space, making the causal connection between a model and its instances immediate. Modeling and runtime environment converge, letting users explore new models with rapid feedback on how these models execute. Simulation results can be immediately at hand, allowing for a rapid prototyping style on the modeling level as known from interpreted object systems.

Figure 3 shows how modeling and runtime environment converge. Users can now interactively explore model behavior and model variants in real-time. This leads to a working style as known from Smalltalk and Self where users incrementally define and explore models. This working style supports the innovative creation of new models in their respective application domain.

We have designed and implemented a system that directly supports the logical four-level architecture and embeds it in a combined modeling and runtime environment. The centerpiece of this system is a virtual machine that directly interprets UML models.

A UML virtual machine, like any virtual machine, is an abstract computing machine. It provides an instruction set and a memory model for representing objects [12, 6].

As the instruction set of the virtual machine, we use UML itself. UML provides several behavior modeling capabilities that can be used to describe the behavior of a model (including itself).[2] Models are persistently represented using XMI, the OMG standard for representing UML models using XML [AA, AB].

For the memory model of the virtual machine, we use the memory management facilities of our implementation language, Java. Every logical object from any M-level is represented as a Java object. As Section 6 shows, we add additional capabilities for model and model instance management and garbage collection.

[2] The current UML specification (UML 1.3) is so imprecise that it is unlikely that two different UML virtual machine implementations behave the same. We address this question in Section 3.5, 6, and 7.

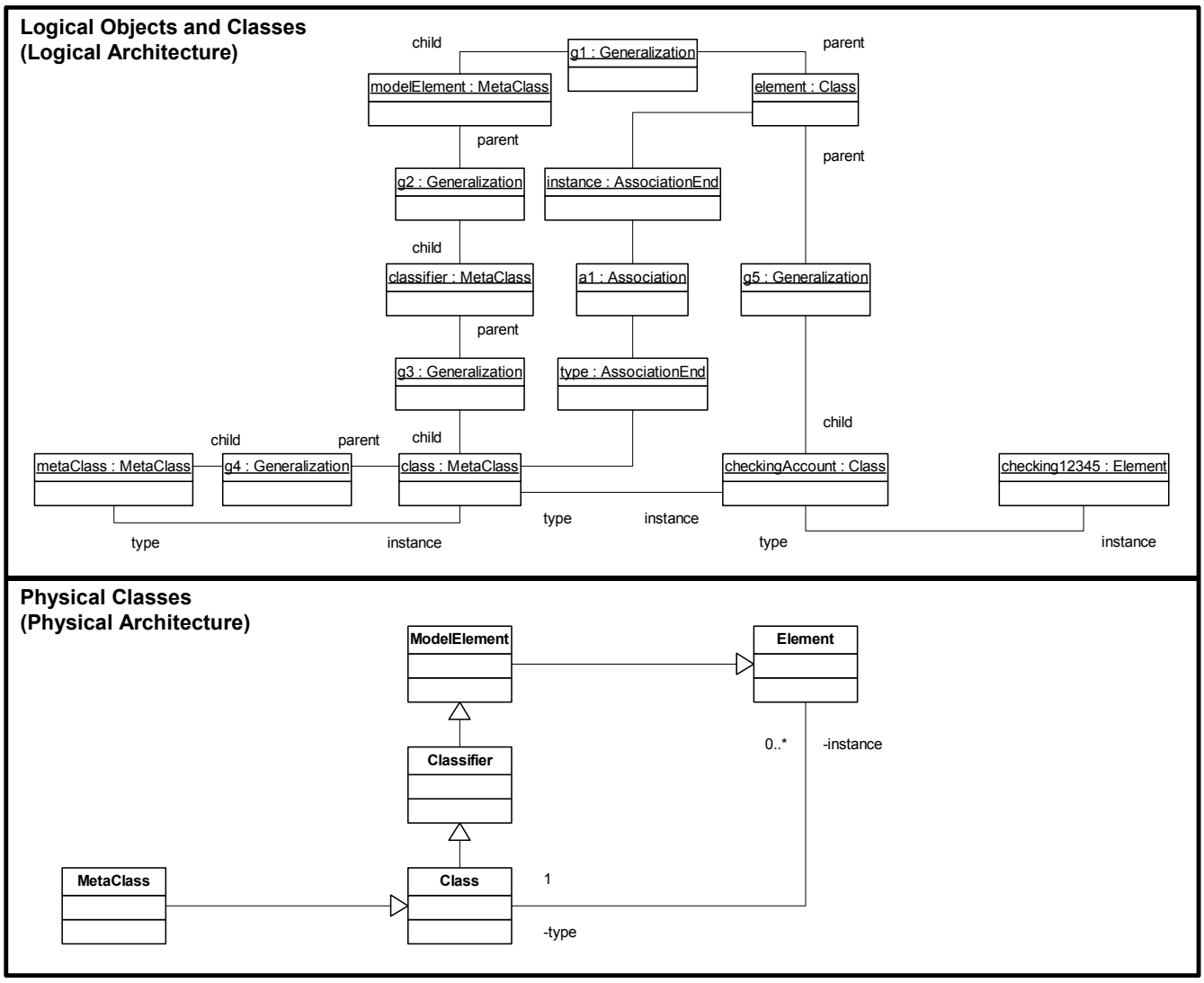

Figure 4: Structure of logical and physical architecture.

3 VIRTUAL MACHINE ARCHITECTURE

The architecture of the virtual machine has two parts: a logical architecture and a physical architecture.

- The *logical architecture* describes the logical structure of how objects relate; it is an extension of the architecture of UML-based systems as discussed in Section 2.
- The *physical architecture* realizes the logical architecture; it takes the form of an object-oriented framework that implements the logical objects.

The logical architecture defines how to achieve the causal connection property, and the physical architecture implements how to achieve this property. There can be different physical implementation architectures, driven by different needs. Our architecture focuses on the efficient representation of model instances. This section discusses the architecture and how it fulfills the causal connection property.

3.1 Logical Architecture

The logical architecture is a pure object architecture: everything is a first-class object, including classes and their specification of instance behavior. Thus, there are objects representing the UML classes, objects representing user classes, and objects representing user objects. This part of the logical architecture conforms to the architecture of UML-based systems as discussed in Section 2. Figure 4 shows the logical architecture in its upper half. In the logical architecture layer, we can see classes like *element*, *modelElement*, *classifier*, etc. (Logical objects are set in italics.)

Because these classes are objects and not programming-language-level classes, the only way of connecting them are object links. Hence, we represent relationships like inheritance and association using objects. For example, Figure 4 shows some inheritance relationships, represented as objects named *g1*, *g2*, *g3*, etc. Using UML terminology, any such object is a generalization object, and it connects a parent class with its child class.

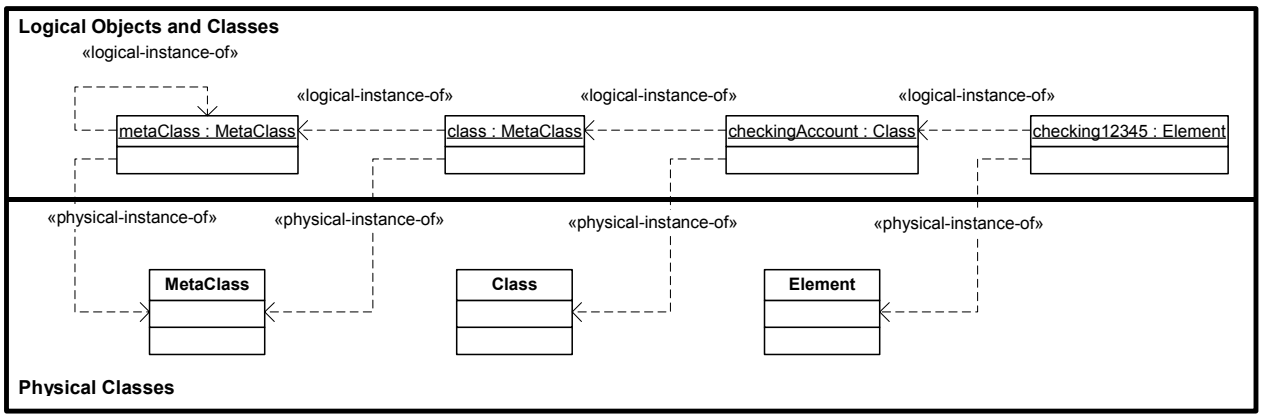

Figure 5: Logical and physical instance-of relationships.

The root class of all classes in the logical architecture is called *element*. It specifies the properties common to all objects in the logical architecture. A prominent subclass of *element* is *modelElement*, the root class of all UML classes. A class in the logical architecture defines which attributes its instances may have and with which elements its instances may be associated.

The logical architecture defines the association *a1* between the *element* class and the *class* class. This association specifies the link between a logical object and its describing logical class. An element object in an instance of this association plays the instance role and a class in an instance of this association plays the type role. This association links every element with its class and is the primary means for fulfilling the causal connection property.

The logical architecture is a single rooted class hierarchy. Thus, every object is a direct or indirect instance of *element*. Figure 4 shows several instances of the instance/type association between *element* and *class*: *checking12345* is an instance of *checkingAccount*, *checkingAccount* is an instance of *class*, and *class* is an instance of *metaClass*, just like the four-level modeling architecture for UML-based systems requires.

Summarizing, the logical architecture mirrors the architecture of UML-based systems as described in Section 2. It also adds several classes like *element*, and several new associations like the instance/type association that specifies how to logically connect an object with its class.

3.2 Physical Architecture

The physical architecture is a set of programming-language-level classes that interact in well-defined ways; they form an object-oriented framework [23]. The physical architecture provides the physical classes for the logical objects.

Every object in the logical architecture has both a logical class and a physical class. The logical class defines its model-relevant properties and the physical class is the programming-language-level class from which the object is physically instantiated.

In its lower half, Figure 4 shows the physical architecture. It shows the five programming-language-level classes Element, ModelElement, Classifier, Class, and MetaClass. Element is the physical class of all logical M0-level objects, Class is the physical class of all logical M1-level classes, and MetaClass is the class of all logical M2-level classes. ModelElement and Classifier are superclasses that are usually not instantiated.

The logical architecture provides a logical class for every class in the UML specification. Thereby, the full UML specification is provided as first-class objects. Many of these logical classes have a corresponding physical class, but not all. If a physical class exists, its instances are used as instances of the corresponding logical class. If no immediately corresponding physical class exists, the closest physical superclass in the inheritance hierarchy is used.

Figure 5 shows examples of physical and logical instance-of relationships. The physical instance-of relationship exists between a physical class and a logical object. The logical instance-of relationship exists between a logical class and its logical instances. For example, the *checking12345* object has the physical class Element and the logical class *checkingAccount*. The *checkingAccount* object has the physical class Class and the logical class *class*. The class object has the physical class MetaClass and the logical class *metaClass*.

The logical class of an object defines the properties of the object from a logical modeling perspective: it defines its attributes and associations; it defines its state model and runtime behavior. At any given point in time during the execution of the system can an object ask its class about its properties and may the class change the properties of its instances.

The implementation of physical classes like Element and Class makes sure that their instances behave according to their logical classes. Element is the root class of all model-relevant physical classes (except for the data type implementations), including Class and MetaClass. It provides a generic attribute and association handling mechanism (and more) that is discussed in Section 3.4.

3.3 Model Representation

Representing models using this architecture is straightforward. Figure 6 shows a domain model of a Customer and some Account classes. Figure 7 shows this model's representation using logical objects as defined in the UML specification.

The model representation in Figure 7 represents every detail of the UML model in Figure 6 as an object, following the UML specifi-

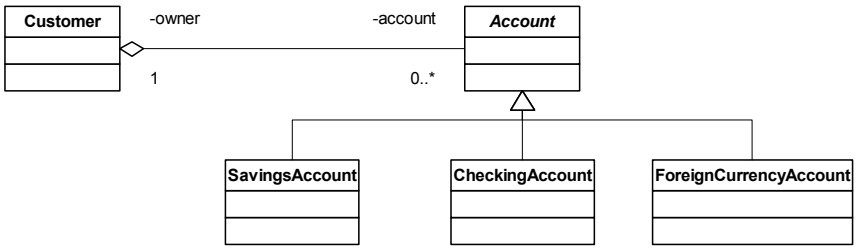

Figure 6: Domain model of Customer and Accounts.

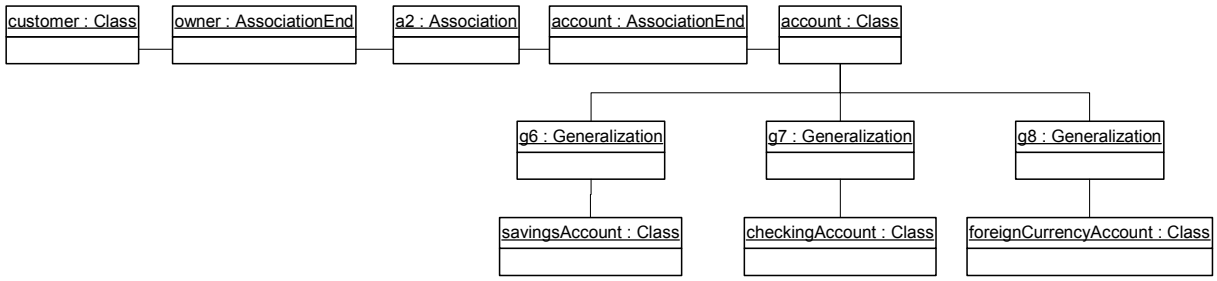

Figure 7: Logical class representation of Customer and Accounts domain model.

cation. For example, the inheritance arrow between Account and CheckingAccount is represented as an instance of the UML Generalization class.

The class name given for an object in Figure 7 is the name of the physical class. The logical class is not shown. For example, account is physically an instance of Class. Logically it is an instance of *class*. The generalization *g7* is physically an instance of Generalization (a physical class) and logically an instance of *generalization* (a logical class object).

The same mechanism that lets us represent M1-level domain models also lets us represent M2-level models, including UML itself. Figure 8 shows a small excerpt from the UML specification, and Figure 9 shows its object representation.

The physical class of a UML class is MetaClass and the logical class is *metaClass*. For example, the logical classes *generalization* and *association* are instances of MetaClass. The relationships are represented using UML, which is a defined operation, because *metaClass* is a subclass of *class*. Hence, all modeling functionality applicable to *class* is also applicable to *metaClass*.

3.4 Physical Class Model

The structural backbone of the architecture is repeated in Figure 10. It takes the structure of a metalevel architecture [29, 10].

The following two classes are of particular importance:

- *Element.* This is the physical (super-)class of any logical object in the system.
- *Class.* This is the physical (super-)class of all logical objects representing classes in the system.

We discuss these two classes in turn.

3.4.1 Element

UML defines the modeling capabilities with which an M1-level user class can be described. Consequently, an M0-level object must provide capabilities to do whatever its M1-level class specifies. Thus, the UML specification indirectly determines the capabilities of any M0-level object and hence the interface and implementation of the Element class.

The structural capabilities include:

- It may have attributes.
- It may have links to other elements, where linked-to elements are conceptually different from attributes.
- It may provide association objects for a given link, if so specified in the model.
- It may be a node in an object composition hierarchy.

From this feature set, we derive the core functionality of the Element class. We add functionality that supports handling of elements for the virtual machine. Listing 1 shows this functionality, reduced to the essentials.

The Element class makes some simplifying assumptions, for example, it does not support multi-valued attributes and it pragmatically distinguishes associations of multiplicity 0..1 from those of multiplicity 0..n. Section 5 discusses some of these simplifications. Section 4 discusses some efficiency considerations for implementing the Element class.

With these capabilities, any physical Element instance can successfully play the role of a logical instance of a logical class. Whatever is modeled for that class, Element provides the functionality to represent it.

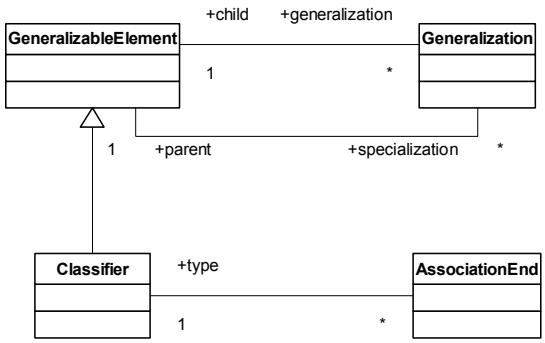

Figure 8: Small and simplified excerpt from the UML specification.

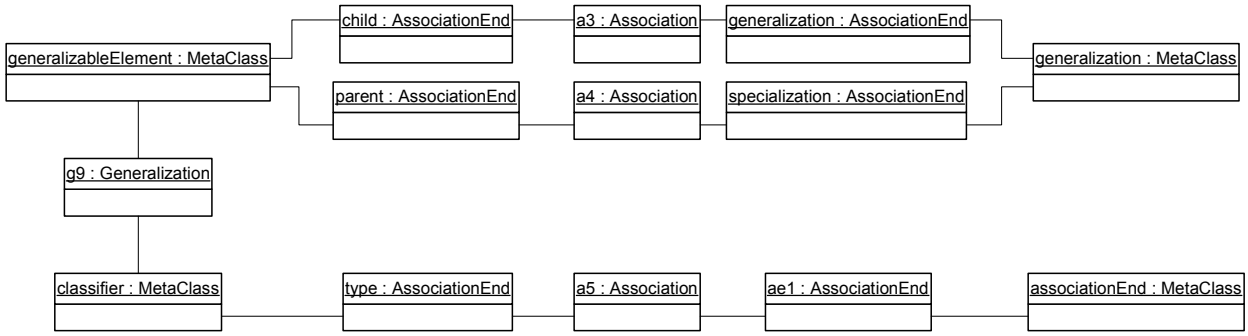

Figure 9: Object representation of the model excerpt from Figure 8.

3.4.2 Class

The single Element class is fully sufficient to represent all logical objects and hence the whole logical architecture. Element instances can play the role of the *checking12345* object and the role of the *checkingAccount*, class, and *metaClass* classes.

However, this is hard to program with and not very efficient. It is hard to program with, because tool implementers and infrastructure classes have to program against a generic interface rather than a more specific interface that expresses the functionality of the class of the object. It is not very efficient because a generic implementation cannot take advantage of constraints that a specific class may be aware of.

For this reason, we implement large parts of the UML as subclasses of Element. By making them subclasses of Element, they inherit its capabilities and hence become full-fledged UML-specified objects themselves. Many of the methods, specifically for attribute and association access, are mere convenience wrappers around the generic methods inherited from Element. However, every such class is free to add functionality that makes using it easier. One such class is Class. Listing 2 shows its interface.

The Class interface provides methods that are derived from the UML specification. For every known feature and association, query and mutation methods exist. As mentioned, these methods are convenience wrappers of the more generic Element methods. Listing 3 shows the implementation of two such methods.

Of more interest is implementation functionality specific to class Class. A prime example is the provision of keys that unambiguously identify attributes and associations. For modeling the attributes and associations of a class, UML provides dedicated Attribute and Association classes. While suitable for modeling, they are too heavy for executing models, and hence we have replaced them with more lightweight key objects for attribute and association access. These keys provide essential typing information and make attribute and association access more efficient. Section 4 discusses these and other implementation considerations.

3.4.3 Data types

The implementation of primitive data types like integer and string and non-primitive data types like money and currency is outside the scope of the Element class hierarchy. Their classes, however, are represented as UML DataType instances. For their implementation, we use the standard Java classes Integer, String, etc. and the data type framework JValue [22].

3.4.4 Match Between Logical and Physical Model

Every physical class is represented as a logical class in the logical architecture. The physical model mirrors (parts of) the logical model. The class model of Figure 10 exists both on the physical and logical level. Therefore, we have both a physical and logical class Element. The same holds true for MetaClass.

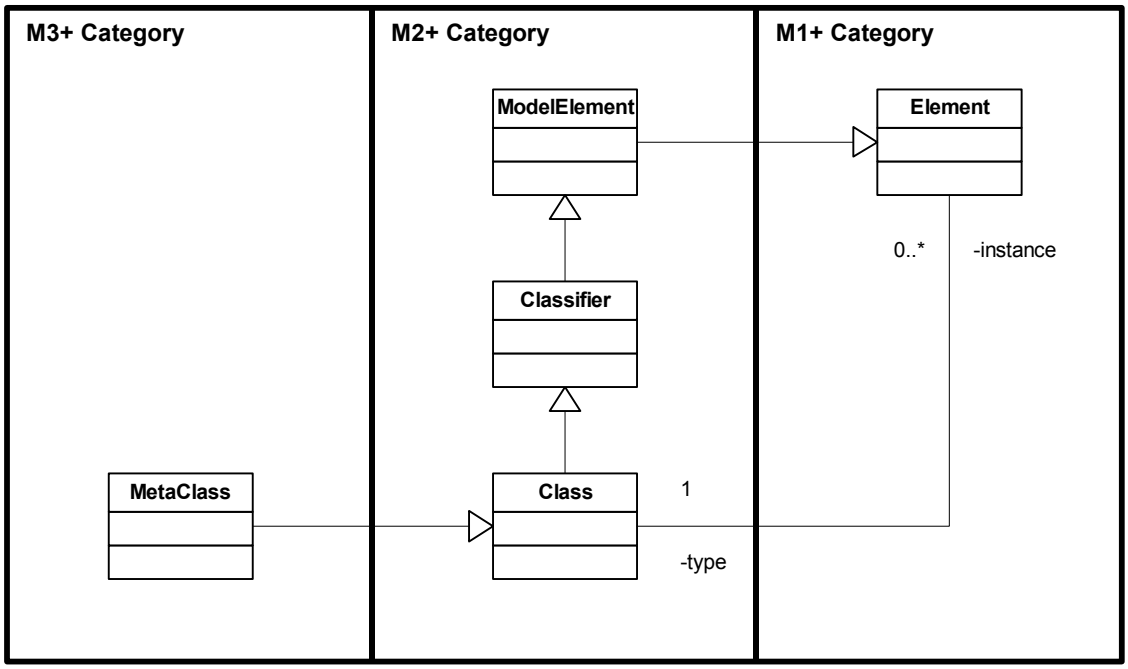

Figure 10: Key classes from the physical architecture.

We match the physical with the logical model, because it reduces the intellectual burden for programmers that implement modeling tools. Programming with a statically typed interface is easier and less error-prone than programming with a generic interface.

3.5 Model Execution

The discussion so far has focused on how to represent models across modeling levels. This subsection discusses how we execute these models. We can only provide a partial solution, as we make several restricting assumptions about model execution based on our runtime architecture. As far as we know, every existing tool with claims similar to ours makes restricting assumptions about the runtime architecture and therefore constrains what and how to model. There are two main reasons for this: first, UML is not defined precisely enough to allow for unambiguous model execution, and second, even if UML was defined precisely enough, one still needs a dedicated runtime architecture on top of which the models run and with which they are integrated. This runtime architecture necessarily imposes constraints on what can be executed and what cannot be executed. In this section, we therefore only demonstrate that model interpretation is possible, and leave the details to further papers.

UML provides several modeling techniques for specifying the behavior of classes. Most of these techniques focus on *illustrating* behavior rather than completely specifying it. Examples of these techniques are object collaboration diagrams and message sequence diagrams. Illustrating behavior means that any such diagram illustrates one specific case of object interaction and behavior, but it does not specify the overall possible set of interactions. While helpful to communicate design intent to human readers, these techniques are not sufficient for fully specifying object behavior as needed by a virtual machine to execute model instances.

The one modeling technique that strives for complete behavior modeling is UML state charts. We use them as the primary tool for modeling object behavior. Every class has a state chart that describes the state space of its instances and the possible transitions that may occur. Each instance interprets the state chart whenever it receives an event in its mailbox. It then reacts accordingly by possibly changing its state and sending out events that represent the state transitions. The state machine interpretation is part of the Element class implementation.

We further enrich class descriptions using OCL to ensure constraints like business rules between elements. State charts serve well to describe the behavior of individual objects, but they do not scale up to describe the behavior of larger parts of a system. We use OCL to describe inter-object dependencies as constraints between objects. This way, state transitions in one object are translated into events relevant to other objects that are not connected with the originating object through a state chart.

Finally, UML is a modeling language and not a programming language, so we add algorithmic detail through hand-programmed policy classes that fit into a well-defined extension architecture [23] (also known as plug-in architecture [15]). This extension architecture is part of our runtime architecture. It both supports and constrains developers in what is possible in terms of model execution. Developers of a model do not only use UML to describe system behavior, but also implement policy classes (using the Strategy design pattern [4]) and hook them up into the model. This approach is a common pattern: all tools we have seen that promise to generate executable systems from models make assumptions about specific runtime environments and thereby constrain developers and the model space.

Hand-programmed policy classes do not contradict the idea of a UML virtual machine. First, UML is not fully executable; hence

```
public interface Element {
    // UML: attributes (single-valued)
    // The FeatureKey identifies the attribute.
    public Object getAttributeValue(FeatureKey fk);
    public void setAttributeValue(FeatureKey fk, Object value);
    ...

    // UML: associations with 0..1 multiplicity
    // The AssociationKey identifies the association.
    public boolean hasLinkedElement(AssociationKey ak);
    public Element getLinkedElement(AssociationKey ak);
    public void setLinkedElement(AssociationKey ak, Element e);

    // UML: associations with 0..n multiplicity
    // The AssociationKey identifies the association.
    public Iterator allLinkedElements(AssociationKey ak);
    public void addLinkedElement(AssociationKey ak, Element e);
    public void removeLinkedElement(AssociationKey ak, Element e);
    ...

    // UML: association object access (functionally dependent on association)
    // The AssociationKey identifies the association.
    public boolean hasAssociationElement(AssociationKey ak);
    public AssociationElement getAssociationElement(AssociationKey ak);
    ...

    // VM model: access to type object
    // With respect to its class, this element plays the instance role.
    public Class getElementClass();
    ...

    // VM implementation: backpointer
    // Provides list of objects linked to this element.
    public Iterator backpointer();
    ...
}
```

Listing 1: Element interface, reduced to the essentials.

we need something like policies to add algorithmic detail. Second, UML is not a programming language but rather a modeling language. Until an executable Action semantics specification for UML is available, UML cannot also play the role of a programming language, and hence has to be complemented by one.

The need for more complete behavior specification and executable semantics of UML models has been recognized. The OMG issued a Request for Proposals (RFP) for executable Action semantics and several groups have responded to it [17]. One of the most important uses of such executable Action semantics will be its application to UML itself, effectively providing a formal and as complete as possible UML virtual machine specification.

3.6 Causal Connection

We need to show that the presented architecture fulfills the causal connection property so that changes in a model lead to immediate changes in the model instances.

As Figure 4 shows, the *element* class is logically the superclass of all other logical classes. Every instance of *element* (independently of on which modeling level it resides) has a link to its class. Thus, every element is connected with the specification of its behavior.

The Element class implementation makes use of this link to determine how an element is to behave and then makes it behave that way. This ensures that every element behaves according to its specification.

In UML, a model is defined as a package of interconnected classes. In the discussed architecture, changes to a class and hence to a model immediately affect any model instance. Thus, a model is causally connected with its model instances.

Schema evolution remains a difficult topic. SKYVA's system lets users specify how instances of an old class version are to be converted into instances of the new class version. These bridges between model versions are one of our extensions to UML. They integrate configuration management with model evolution with instance upgrading.

4 IMPLEMENTATION

At its primitive level, the implementation of the Element class and its related classes draws on the access of attributes and associations. Getting the implementation of these primitives right determines the overall performance more than any other factor.

All publicly available UML implementations that we have seen use string-based access to attributes and associations. We use key-based rather than string-based access to attributes and associations. This made our virtual machine implementation perform significantly faster and removed dynamic typing problems that occur when attributes or associations are named using potentially misspelled strings.

A key object indicates a specific attribute or association. Effectively, it is a shallow representation of the corresponding Attribute or AssociationEnd object. For attribute access, we provide dedi-

```
public interface Class extends Classifier {
    // UML: a class has features that can be accessed by name
    public boolean hasFeature(String featureName);
    public Feature getFeature(String featureName);
    public Iterator allFeatures();
    public void addFeature(String featureName, Feature f);
    public void removeFeature(String featureName);
    ...

    // UML: a class has association ends that link it to associations
    public boolean hasAssociationEnd(String roleName);
    public AssociationEnd getAssociationEnd(String roleName);
    public Iterator allAssociationEnds();
    public void addAssociationEnd(AssociationEnd ae);
    public void removeAssociationEnd(AssociationEnd ae);
    ...

    // VM implementation: provide access to keys
    public Iterator allFeatureKeys();
    public FeatureKey getFeatureKey(String attributeName);
    public Iterator allAssociationKeys();
    public AssociationKey getAssociationKey(String roleName);
    ...
}
```

Listing 2: Class interface (showing inherited Classifier methods for convenience).

```
public Feature allFeatures() {
    return allLinkedElements(FEATURE_KEY);
}

public void addFeature(Feature f) {
    addLinkedElement(FEATURE_KEY, f);
}
```

Listing 3: How Class uses the generic methods of Element.

cated feature key objects, and for association access, we provide dedicated association key objects, as illustrated in Listings 1, 2, and 3. For every attribute of a class, one feature key is created and updated whenever the attribute's definition changes. The same is done for associations and association keys. Because a class changes less frequently than its instances, creating and updating the keys does not pose a significant runtime overhead.

Every key provides a unique index into an array for efficient access to the attribute's value or linked-to element. The class managing the keys has a complete overview of all its keys and can therefore compute a perfect distribution of indices for the keys. Every element manages its attribute values and linked-to elements in an array. Looking up an attribute's value or a referenced element is as simple and fast as the access to a field in the array.

A key does not only provide an index into an array but also provides typing information. Effectively, type information is copied from the model into the key and transformed with the goal of speeding up type checking.

Whenever an attribute or association is accessed, it must be checked whether the key is actually a valid key identifying an attribute and association that really exists for the given object. This type checking is necessary to protect against bugs in the virtual machine implementation and in the policy implementations by which users add programmatic behavior to models.

A sequence of checks ensures integrity of access: the owner of the key and the class of the element must be identical, and the typing information found in the key must match the meta-information stored directly in the element, for example, whether a linked-to element is truly a composed element or merely a regularly associated element.

Finally, key objects are shared immutable objects. A managing object controls their instantiation (see the Flyweight design pattern [4]). This way, no client code can create key objects. This lets the system maintain control over typing information. No typing information ever enters the system from the outside without being checked at the system boundaries. Client code like policies request key objects, they do not create them. While this cannot prevent that a message is being sent to the wrong object, it at least ensures that the message is always a valid message.

We discuss further implementation considerations and optimizations in a related paper [24].

5 UML EXTENSIONS

In Section 3 on the virtual machine architecture, we describe how we make UML part of a larger logical architecture. In this section, we discuss some of our extensions to UML and some of the simplifications we applied.

5.1 Technical Domain Models

For executing application domain models, users need a means to express how to present the model instances in user interfaces for user interaction, how to represent them for persistence in data-

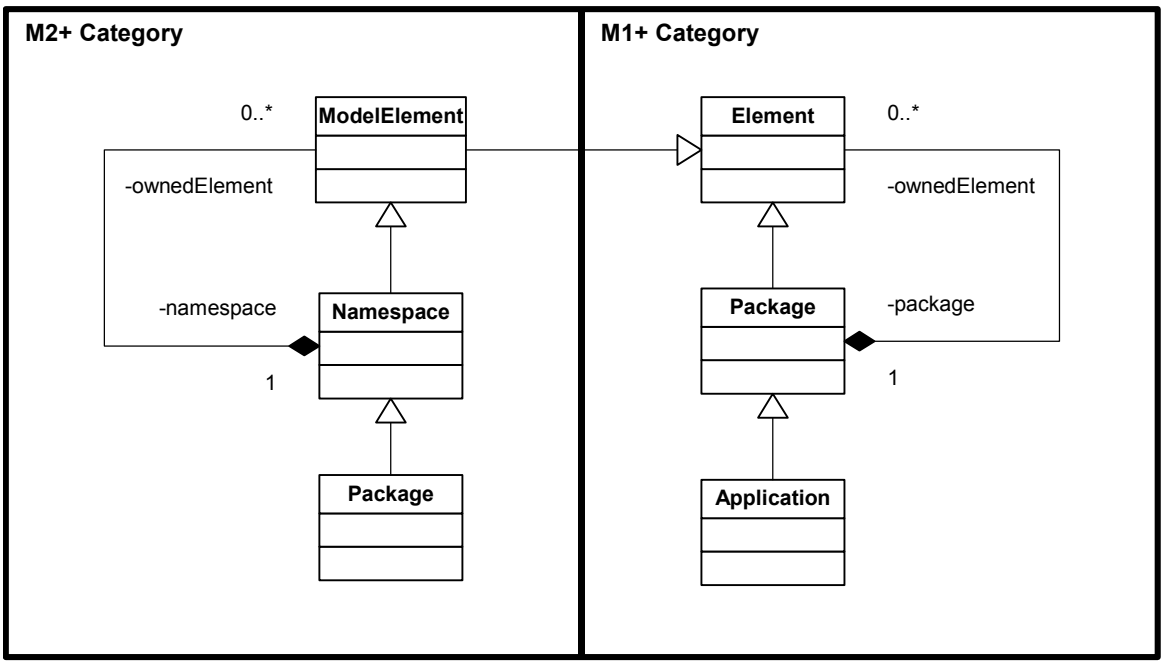

Figure 11: Managing elements in packages and applications.

bases, and how to distribute them across different processes and computer systems.

This information is cast as technical domain models that exist next to the application domain models (technical meaning a technical (infrastructure) application domain as opposed to a business application domain).[3] Users can extend and configure the technical domain models as much as they can define and then extend and configure the application domain models. Default models may apply, but users can always customize them.

The first user of these technical domain models is the modeling environment. It uses models to describe user interface layouts, database schema mappings, and system distribution. These models are specific to a given runtime environment and can differ widely from implementation to implementation.

One common need, however, that is likely to be found in any system implementation, is the need to organize model instances in packages and applications. We therefore extend the UML package and project concepts to apply to every element, not just UML model elements. Figure 11 shows the resulting model.

Figure 11 shows two Package classes. On left, it shows the traditional UML Package class, which is a subclass of Namespace and which is reserved to contain only model elements. On the right, Figure 11 shows a general Package class that can contain any kind of element, be it a UML model element or a regular non-UML element.

The semantics of the general Package class are weaker than those of the UML Package class: it accepts different elements with the same name and it does not support package inheritance. This shift in semantics is the reason why we decided to introduce the general Package class rather than to extend the existing UML Package class to contain any type of element.

A special subclass of the general Package class is Application. Its instances are the root objects for a given running application.

5.2 UML Simplifications

In our UML implementation, we have applied a number of simplifications. Structural simplifications include:

- No multi-valued attributes of elements.
- Attributes of elements are always UML data types.
- Only binary associations between classes.

So far, we have not encountered any problems due to these restrictions.

6 EXPERIENCES

We have built three systems with this type of architecture, two at UBS, and one at SKYVA International.

The two systems built at Ubilab, the late IT research laboratory of UBS, were a research prototype that provided a full-fledged metalevel architecture as described in this paper, and a production-level system that let users model, represent and edit corporate loans. The research prototype was used to explore and demonstrate the idea of model-driven software systems based on a dynamic object model, as we called it at that time. The production-level system is used in UBS' corporate customer business. Its primary functionality is in capturing and presenting corporate loans. Effectively, it is an object-oriented product data manage-

[3] These technical domain models are equivalent to the class libraries and frameworks that come with Smalltalk and Java.

ment system. Both systems did not use UML on the M2-level but rather a simpler proprietary metamodel. Both systems worked well for their purpose. However, both systems focussed primarily on representing and editing structure, which is much simpler than representing and executing behavior.

The UML virtual machine built at SKYVA International is part of SKYVA's main product for supply chain management and collaborative e-commerce. It has been used in a number of industry projects and is the most extensive base of our implementation experience. The system exhibits most characteristics discussed in this paper. It has a full-fledged UML-based metalevel architecture. It lets users model their application domain and execute these models. Feedback on model changes is immediate and supports users in exploring domain models.

Still, it is a hybrid system: for innovative model exploration and system configuration, we use an interpreted approach, and for production-level system execution, we use a code-generation approach. Hence, we have two runtime environments: one UML virtual machine that is embedded in the modeling environment and that allows for lightweight model execution, and one that is a separate system capable of carrying high volume transactions and mass data persistence.

Our biggest remaining problem is modeling of behavior and execution of the modeled behavior. At the time of writing, we still have to implement a significant amount of code (policies) to add behavior to the models. UML's behavior modeling features are not sufficient to completely describe desired behavior and our behavior modeling extensions and implementations have not fully overcome this problem. However, other companies, for example Project Technology [20] and Kennedy Carter [AC] have shown that precise behavior modeling is possible with (an extended form of) UML. Their systems allow the execution of models based purely on modeled rather than implemented behavior. Key to their approach as well as our approach is knowledge about the target runtime architecture.

SKYVA's system comprises significantly more components than the virtual machine discussed in this paper. The modeling environment provides an elaborate repository-based infrastructure that supports model persistence, configuration management, collaboration in a team and more. The system provides not only technical domain models but generic application domain models as well, targeted at supply-chain management and e-commerce.

We have found that generic domain models are best supported by UML extensions that reflect the domain concepts. Similarly, we have found that UML extensions without a complementing generic domain model are of limited use, because we cannot integrate separately developed domain models. This is in contrast to industry's current attempts to provide standards by extending UML only without providing (generic M1-level) domain models. Also, we view the lack of a standardized Element class as a major hindrance for integrating independently developed domain models in UML-based modeling.

7 RELATED WORK

We discuss related work on virtual machines, metalevel architectures, model-driven software systems and UML-based software development tools.

Virtual machines for programming languages like Smalltalk [6], Self [28, 9], and Java [1, 12] have both inspired this work as a metaphor and provided solid implementation advice. Like Smalltalk and Self, and unlike Java, we fully represent the modeling language using the system's own capabilities.

The architecture of the virtual machine, however, probably owes most to CLOS [2, 19, 5]. CLOS' simple yet elegant metalevel architecture directly influenced how we extended UML with M1 and M3-level classes to turn the logical model into a reflective system.

While the described architecture shares structural and behavioral properties with the CLOS metalevel architecture and related reflective architectures [13, 29, 10, 21], we address a problem that has not been addressed by any such programming-language centric architecture: the modeling of application domains using a dedicated modeling language and the execution of the resulting models. To the best of our knowledge, we are the first to combine a modeling language with a metalevel architecture with a virtual machine approach.

Others have recognized the need for model-driven software systems that provide an explicit model of themselves. Most notably, Tilman provides an account of an object-oriented framework that is used to capture models and model instances and to persist them between instantiations [26]. Tilman's application domain is form-based workflow-oriented business applications. A similar system is discussed by Kovacs [11]. Kovacs and his colleagues built a system for product configuration and workflow management of large high-energy physics detectors. Similarly, Manolescu discusses a system that explicitly represents workflow descriptions next to the actual workflows [14]. Common to these systems is that they all have a descriptive layer next to an instance layer. However, in all three cases, the modeling language is specific to that system.

Currently, a lot of industry research and development is invested into UML-based modeling tools. As discussed in the introduction, the common pattern is to model a system using UML and then to generate code and publish the code into a runtime system [8]. A few systems have enhanced UML with precise behavior modeling so that no programming by hand is necessary. An example is Project Technology's BridgePoint system [20], originally based on the Shlaer Mellor methodology [25]. BridgePoint lets users model embedded systems using an enhanced form of UML that supports precise behavior modeling. From the models, code is generated. Another example is Kennedy Carter's iUML tool suite that supports modeling of embedded systems using UML with precise action semantics [AC]. iUML also let's users simulate the modeled system, providing feedback about the system. Project Technology and Kennedy Carter have worked together on one of the submissions to the OMG RFP for precise action semantics for UML.

8 CONCLUSIONS

This paper presents the architecture of a virtual machine for UML. The virtual machine consists of a logical extension of the UML four-level modeling architecture plus an object-oriented framework that implements this architecture. The virtual machine explicitly represents UML, UML models, and UML model instances (actual instances of running systems). This approach lets users

immediately see the effects of model changes. This feedback supports rapid model prototyping and innovative exploration of models better than possible with today's code-generation approaches.

While we use rapid user feedback as the motivation to choose an interpreted approach over a code-generation approach, other advantages of the UML virtual machine approach weigh in as well. For example, it is significantly easier to develop a 24x7 always-on system, because the availability of explicit models makes system evolution easier. We have documented this and other business drivers behind our approach in [AD].

We see several areas for further research: the execution speed of the virtual machine, better precise behavior modeling, the definition and integration of modeling language extensions to integrate otherwise unrelated domain models in a coherent system, and model evolution support. Our experience indicates that all of these areas are important and must be addressed, contributing significantly to the usefulness of the virtual machine approach.

We have found many shortcomings in the UML specification that limit its usefulness. However, these shortcomings are being recognized. Future specifications of UML and related technologies will provide a basis on top of which UML virtual machines can be standardized. Then models will become exchangeable between virtual machines and lead to the same system behavior.

ACKNOWLEDGEMENTS

We would like to thank Thomas Gross, Christian Nester, Philipp Oser, Johan Ovlinger, Alan Perry, Wolf Siberski, Hans Wegener, and the anonymous reviewers for valuable feedback that helped us improve the paper.

REFERENCES

[1] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[2] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. "Common Lisp Object System Specification." *SIGPLAN Notices,* 23 (Special Issue), September 1988.

[3]

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[5] Gregor Kiczales, Jim D. Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[6] Adele Goldberg and David Robson. *Smalltalk: The Language and Its Implementation.* Addison-Wesley, 1983.

[7] Adele Goldberg and David Robson. *Smalltalk: The Language.* Addison-Wesley, 1989.

[8] William Harrison, Charles M. Barton, and Mukund Raghavachari, "Mapping UML to Java." In Proceedings of *Object-Oriented Programming Languages, Systems, and Applications* (OOPSLA 2000). ACM Press, 2000.

[9] Urs Hölzle. *Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming.* Ph.D. Thesis STAN-CS-TR-94-1520. Stanford University, 1994.

[10] Akinori Yonezawa and Brian C. Smith (editors). *Proceedings of the International Workshop on New Models for Software Architecture '92: Reflection and Metalevel Architecture* (IMSA '92). Japan: November 4-7, 1992.

[11] Zsolt Kovacs. *The Integration of Product Data with Work Flow Management Systems through a Common Data Model.* Ph.D. Thesis. University of the West of England, 1999.

[12] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, 1998.

[13] Patti Maes and Daniele Nardi. *Meta-Level Architectures and Reflection.* Elsevier Science Publishers, 1988.

[14] Dragos-Anton Manolescu. *Micro-Workflow: a Workflow Architecture Supporting Compositional Object-Oriented Software Development.* Ph.D. Thesis. University of Illinois at Urbana-Champaign, 2001.

[15] Klaus Marquardt. "Patterns for Software Packaging, Installation and Activation." In *Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing* (EuroPLoP 1998). Universitätsverlag Konstanz, 1998.

[16] OMG. *Metaobject Facility Specification 1.3.* OMG Document 99-06-05. OMG, 1999. Available from www.omg.org.

[17] OMG. *Action Semantics for the UML RFP.* OMG Document 98-11-01. OMG, 1998. Available from www.omg.org.

[18] Donald G. Firesmith, Brian Henderson-Sellers, Ian Graham, and Meilir Page-Jones. *Open Modeling Language Reference Manual.* SIGS Publications, 1998.

[19] Andreas Paepcke (editor). *Object-Oriented Programming: the CLOS Perspective.* MIT Press, 1993.

[20] Project Technology. *BridgePoint Tutorial.* Project Technology, 2000. Available from www.projtech.com.

[21] Gregor Kiczales (editor). *Proceedings of Reflection 1996.* Xerox Parc, 1996.

[22] Dirk Riehle. *The JValue Value Object Framework, Version 0.5.1.* Available from www.jvalue.org.

[23] Dirk Riehle. *Framework Design: a Role Modeling Approach.* Ph.D. Thesis, No. 13509. ETH Zürich, 2000. Available from www.riehle.org/diss.

[24] Dirk Riehle, Michel Tilman, and Ralph Johnson. "Dynamic Object Model." In *Proceedings of the 2000 Conference on Pattern Languages of Programs* (PLoP 2000). Washington University Technical Report Number WUCS-00-29.

[25] Sally Shlaer and Neil Lang. *Shlaer-Mellor Method: The OOA96 Report.* Project Technology, 1996. Available from www.projtech.com.

[26] Michel Tilman and Martine Devos. "A Reflective and Repository-Based Framework." In *Implementing Application Frameworks.* Wiley, 1999. Page 29-64.

[27] OMG. *Unified Modeling Language Specification 1.3.* OMG, 1999. Available from www.omg.org.

[28] David Ungar and Randall B. Smith. "Self: The Power of Simplicity." In *Proceedings* of *Object-Oriented Programming*

Languages, Systems, and Applications (OOPSLA 1987). ACM Press, 1987.

[29] Chris Zimmermann (editor). *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, 1996.

[AA] OMG. *XML Metadata Interchange.* OMG Document 98-10-05. OMG, 1998. Available from www.omg.org

[AB] OMG. *XML Metadata Interchange Specification V1.1.* OMG, 1998. Available from www.omg.org.

[AC] Kennedy Carter. *I-OOA Modelling Tool Technical Overview.* Kennedy Carter, 2000. Available from www.kc.com.

[AD] Dirk Riehle and Erica Dubach. "Why a Bank Needs Dynamic Object Models." Position Paper for OOPSLA '98, Workshop 15. Available from www.riehle.org.

9 NOTES

Brian Foote. "Object, Reflection, and Open Languages." Relationship between reflection, extensibility (and frameworks?)

Discuss need for specialized target architecture.

Abstract syntax trees as representation form.

Change class names to recursive but first Capital letter?

The next sections discuss the structural aspects of the architecture of the virtual machine and its implementation.

Words: 8205