Tube: Interactive Model-Integrated Object-Oriented Programming*

Axel Rauschmayer
Axel.Rauschmayer@ifi.lmu.de
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67
D-80538 München, Germany
Phone: +(49) 89 2180-9126
Fax: +(49) 89 2180-9175

Patrick Renner
renner@in.tum.de
Institut für Informatik
Technische Universität München
Boltzmannstr. 4
D-85748 Garching b. München, Germany
Phone: +49 (89) 289-18206
Fax: +49 (89) 289-18207

Abstract

Software engineering is hampered by the fact that software systems quickly become so complex that they are hard to understand, evolve and maintain. Closer integration of code and model helps, because the model serves as a map to the code and the code fills in the details for the model. Simultaneously, one avoids consistency problems. TUBE, a programming language and an integrated environment, achieves this integration by using *topic maps* to manage both code and data (including meta-data and non-code artifacts). This enhanced expressiveness is complemented by an interactive way of system construction that cannot be achieved by static programming languages.

Keywords: Software Design and Development, Software Development, Prototype-Based Object-Oriented Programming, Model Integration

1 Introduction

One of the greatest problems in software engineering is that a software system quickly becomes so complex that it is hard to understand and thus to evolve and maintain. The usual solution is to provide the developer with a formal or semi-formal description of the *model*, a view of the system at a high level of abstraction. The model and code artifacts existing independently leads to several problems, however: You cannot look up the details in the code when reading the model and you cannot find out about design decisions when editing the code. Additionally, separate evolution of the two kinds of artifacts sooner or later leads to them becoming inconsistent. Then how can we integrate code and model? There are two schools of thought that answer this question: First, the "the model is the code" faction consists of people who like the Unified Modeling Language UML or more generally, modeling. *Model-Driven Architecture* (MDA, [9]) is their way of producing UML models that can be executed. Second, the "the code is the model" faction is represented by various kinds of agile development methods, *eXtreme Programming* (XP, [3]) being the most popular example. This group wants to make the source code so expressive that it clearly represents the concepts that have been implemented in it. Both schools have their problems: in MDA, a lot of complexity is hidden in generators. When having to adapt them, one is back to writing regular source code, in a manner that is much more difficult than regular coding. In XP, the expressiveness of normal source code is rarely enough for clearly displaying all of the modeling knowledge.

The core idea of our programming language and integrated environment TUBE is to let model and source code live in the same space. To express modeling concepts, we turn to a data structure that comes from the field of knowledge representation: *topic maps* [4]. If we start with a topic map to express our concepts and then add code to make them executable, we get a light-weight version of MDA, without inheriting its complexity. In spirit, TUBE is also closely related to *literate programming* [8].

The code structure is laid out using the two basic constructs from topic maps, topics and associations. Topic maps being graph-based, these correspond to nodes and edges in graphs. Different semantic aspects of the code

*Supported by Deutsche Forschungsgemeinschaft (DFG) project WI 841/6-1 "InOpSys"

can be modeled by annotating topics and associations with semantic information. This leads to our semantics being very flexible and declarative. With this preparation, all kinds of data can be integrated with the code and we get the following benefits: Tracing and linking between code and non-code artifacts is easy; further annotations (read: meta-data) of the code do not interfere with its execution (see *multi-dimensionality* in [11]); and we can use an arsenal of existing topic map tools to present, manage and query our inter-linked software system.

The fine-grained and explicit way of representing code gives us an "assembly language" for code structure where many aspects can be modeled and queried using just a few atomic constructs. On the other hand, it is now the responsibility of an editor tool to shield the programmer as much as possible from unnecessary details and to give meaningful visualizations (at varying levels of abstraction) of the structure (see Sect. 5).

When programming in our integrated development environment (IDE), we wanted to have a feeling of interactivity that resembles how modeling and ontology building happens. Static programming languages such as Java [6] have great IDEs, but these are mainly supporting a static view of a system. We therefore turned to the dynamic programming language *Self* [12] for inspiration: TUBE copies many of its semantic concepts and its way of interactive program construction.

The rest of the paper is organized as follows: We first give an informal overview of the semantics (Sect. 2). Due to space constraints, the formal definition of the semantics can only be sketched (Sect. 3). Sect. 4 shows advanced ways of modeling structure in TUBE. Sect. 5 presents our implementation of TUBE. The paper concludes (Sect. 8) after outlining related (Sect. 6) work and our comprehensive future research (Sect. 7).

2 Programs as Hyper-Graphs

In this section we informally introduce the elements of the TUBE programming language and describe the structure of a program, the message dispatch algorithm and the role of the embedded program language.

2.1 Topic Maps

Topic maps have their origin in knowledge representation and the semantic web and are therefore strongly focused on describing domain knowledge. They are standardized as ISO-Standard ISO/IEC 13250 [4]. Topic maps consist of three main concepts (so-called *topic map items*): topics, associations and occurrences. *Topics* are used to represent a *subject* (any real-world "thing" or conceptual entity). Topics are related to each other through *associations*. Associations are labeled by a set of *role names*, one for each topic that they relate. To provide topics with attributes (e. g., to link to documents or other additional information), one can attach *occurrences* to them. An occurrence is a (key,value) pair where the value is either a reference to a (conceptual or real-world) resource or a literal. Associations and occurrences can themselves be represented as topics for further annotation through associations. This so-called *reification* enables meta-descriptions to be part of the topic map object level. For further information on topic maps, consult [10].

2.2 Topics and the Embedded Programming Language

In TUBE, all programming elements such as methods, data objects and prototype objects are represented by topics. There are two main kinds of topics: First, *content topics* store information and can be evaluated; we do not distinguish between code, literals and expressions. Second, *primitive topics* store primitive language constructs that are not available via the *embedded language* (see below). An example is the setter primitive that replaces an existing child by a new one (the association stays the same).

The embedded programming language[1] provides the basic data objects as well as the syntax for defining the content of a method. With this approach, we are able to separate the structure of a program (which is handled by the topic map) and the algorithms and expressions (which are provided by the embedded language). As we will point out in the following chapters, this leads to a very expressive programming style which enables the developer to much better integrate model aspects of the software with code than would be possible without the use of graphs.

2.3 Associations and Message Dispatch

In the TUBE prorgamming language, program execution is graph traversal initiated by messages. Message dispatch comprises two major steps: Finding the right topic and evaluating it. Therefore, two primitive semantic operations are used, the locate() and the eval() operation (see Sec. 3). To control message dispatch, four different kinds of associations determine semantic behavior: *child*, *delegate*, *parent* and *associate* associations (Fig. 1). All of the associations can have a name and additionally be tagged as *shared*. Note that whenever we talk about a delegate

[1] In our implementation, we use Python as the embedded language

or a child etc., we mean the topic that the corresponding association points to.

2.3.1 Finding a Topic

A message consists of a name and a set of parameters. If we send a message to a topic, we want to find a child whose association name matches the message name. This is the topic that we need to evaluate.

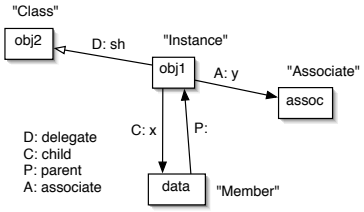

Figure 1: Structuring objects in a TUBE graph: Constructs like the class-instance relationship or relations to members and associated objects are modeled explicitly in the graph

2.3.2 Evaluating a Topic

To evaluate a topic, eval() is invoked on that topic. As a first step, the parameters that come with the message (e. g., a dictionary $\{x = 10, y = 7\}$) are destructively set as the topic's children. During the execution of the embedded language code, the program has access to all of its direct children and associates and thus to the parameters as well.

2.3.3 Sending a Message and Delegation

In the simplest case, the topic that has received the message has a child whose name matches that of the message. Then the child is copied by the locate() operation and eval() is executed in place. The copying is done to create fresh instances of the parameter variables for each invocation. If the topic has no child with the name given in the message, locate() searches recursively for a matching child among the delegates (see Fig. 2). When the currently searched topic has no more delegates, the parents and their delegates are searched.

The distinction between parents and delegates (see Fig. 3) can be seen as the distinction between nesting and inheritance: If topics are nested (associated with a parent-association) the found topic stays in its context (*lexical scope*) by being copied to where it was found during locate(). If a topic is a delegate of another topic, the

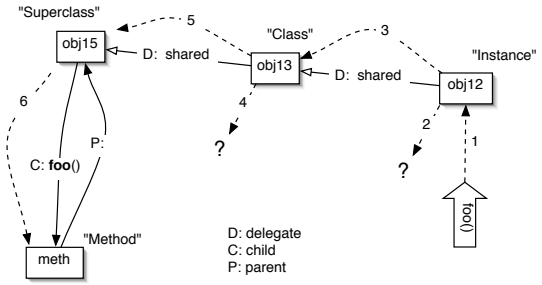

Figure 2: Recursive searching for a matching child following the delegates hierarchy implementing inheritance and class membership.

found child is copied to the entry point of the message (the topic that it was originally sent to) and uses therefore the entry point's context (parents etc.). Note that by copying nodes this way, we are building the static context of code. There are two points where dispatching can begin: At the `me`-topic (the entry point) and the `glob` topic that manages the global namespace.

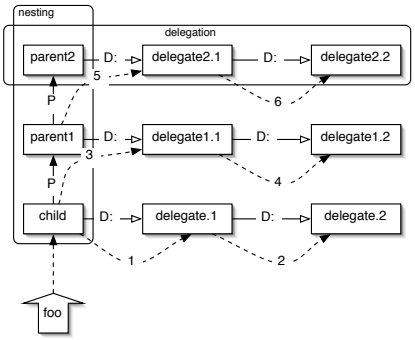

Figure 3: Hierarchy of nesting and delegation: Delegates are searched for a matching child first, then the parents and their delegates.

2.3.4 Copying and Sharing

We use copying (*cloning*, [13]) of prototypes for object creation rather than class instantiation. Copying is implemented by a generic operation that is directed by association annotations: When copies are made of a topic, the *shared* tag of an association determines if associated topics are copied as well: Non-shared topics are copied (*deep copy*, e. g., to model "instance variables" from the Java world) and shared topics are associated to the same topic as the prototype (*shallow copy*, e. g. for "static class vari-

ables"). This allows us to declaratively specify the copy semantics of our objects.

3 Formal Semantics

To define the semantics of TUBE, we have split the language into two parts: On one hand, the *embedded language* is responsible for defining basic language constructs such as expressions, loops etc. On the other hand, the *structural language* defines message passing, structural elements etc. This section gives a brief overview of the operations that make up the semantics of the structural language. When used in the programming language, these are also called *primitives*, the setter primitive being the most prominent example. The embedded language is Python (see Sect. 5.1); for its semantics, consult [14].

3.1 Atomic Semantic Operations

The following operations are the basic building blocks of the semantics:

exec : $\text{Topic} \times \text{Topic} \times \text{Topic} \to \text{Void}$
: Evaluate the topic content either (1) in the embedded language, pass `glob` and `me` as parameters or (2) as a TUBE primitive

clone : $\text{Topic} \to \text{Topic}$
: Copy the topic/association including the annotations

newTopic : $\to \text{Topic}$
: Create a new topic. Associations are topics, too. A separate relation $\text{assoc}\langle\rangle$ stores triples (topic,source,target).

3.2 Composite Semantic Operations

We define the following composite operations using rule-based *strategies* (see next section):

locate : $\text{Topic} \times \text{Msg} \to \text{Topic}$
: Find a topic, set up its context

eval : $\text{Topic} \times \text{Params} \to \text{Void}$
: Assign the values given by the parameter dictionary to the corresponding children, evaluate the topic content

jmpToDynDelg : $\text{Topic} \times \text{Msg} \to \text{Void}$
: Jump to the dynamic delegate, execute it in place

graphCopy : $\text{Topic} \to \text{Topic}$
: Copy a contiguous graph component

set : $\text{Topic} \times \text{Msg} \times \text{Value} \to \text{Void}$
: Assign a new value to the child of a topic

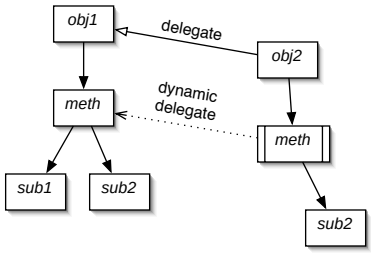

Figure 4: In `obj2`, we would like to override just the sub-method `sub2` of `meth`. The two versions of `meth` are connected by an implicit (and invisible) association called a *dynamic delegate*. A box with bars indicates that this topic is marked as having a dynamic delegate.

3.3 Strategies

Each composite semantic operation is defined by a *strategy*, a named parameterized set of rewrite rules that have functional guards and functional "where" definitions. A strategy returns a value by adding the term $\text{return}\langle\rangle$ to the term store. If there are no rules that can fire, it is considered a failure of the operation. Due to space constraints, we do not show the definitions of the semantic operations, but instead demonstrate a smaller helper strategy below. $\text{findMatchAtCurrentLevel}()$ looks for a topic that has a child whose name matches a message identifier. Terms are written with angle brackets, functions with parentheses.

$\text{findMatchAtCurrentLevel}(x, \text{msg})$
local: cur
rules:
$\to \text{cur}\langle x\rangle$
$\text{assoc}\langle e, n, m\rangle, \text{cur}\langle n\rangle \mid \text{name}(e) = \text{msg}$
$\to \text{return}\langle m\rangle, \cdots$
$\text{assoc}\langle e, n, m\rangle, \text{cur}\langle n\rangle \mid \text{isDelg}(e)$
$\to \text{cur}\langle m\rangle, \text{assoc}\langle e, n, m\rangle$

4 More Structural Modeling

In this section we slightly extend the feature introduced in Sect. 2 and give examples of more sophisticated structural modeling.

Sub-Methods and Nested Overriding. The idea is as follows: Instead of having large monolithic methods with sections that are named by comments, we turn each section into a helper method that has just that (potentially very long, but descriptive) name. If we iterate that process, we get a tree of methods that call sub-methods. Ideally, none of these methods has more than a few lines of

code (7 ± 2 pieces fit into short-term memory and can thus be grasped at one glance). This practice is common in XP and gives us a semantic skeleton of the program if we ignore the source code and look only at the method names. In contrast to Java and Self, TUBE can represent the tree explicitly, as lexical scope can be nested arbitrarily. To really make this construct useful, though, we have to provide for selective overriding of nested topics. To see how this works, take a look at Fig. 4. We do not want to override all of `meth`, just `sub2`. We need to solve two problems if this is to work: First, children of `meth` in `obj2` have to be able to find the children of the overridden `meth`. Second, if the overriding topic calls the overridden one (like send super calls in Java), overriding children have to get called first.

To solve the first problem, one can mark a topic as "having a dynamic delegate". From now on, this topic is connected to its dynamic delegate via a virtual delegate association. We compute its target by searching among the delegates of the parent for a child that has the same name as the source topic. Note that the delegates of the parent could be recursively dynamic, leading to general applicability of this principle. Having this kind of dynamic delegation helps to avoid too many explicit associations and keeps our program structure flexible. The second problem is solved by providing an operation jmpToDynDelg that executes the delegate *in place*[2]. It then sees the overriding children before its own.

Packages and Method Groups. Modeling nested namespaces—as needed for packages (modules)—is easy in TUBE. One just adds intermediate associations: For example, if there is a global object `glob.obj` that we want to put into a namespace called `pkg`, we add an empty topic. It becomes a child called `pkg` of `glob` and has the object as a child called `obj`. We can even make the namespace `pkg` show up only when browsing the program: it becomes invisible to message dispatch if we use a delegation association (instead of a child one) from `glob` to `pkg`. In the same manner, we can structure objects by naming groups of related methods.

5 Implementation

Currently, the prototype of the TUBE implementation is split into two parts: A front end, the Tube Editor, that browses and modifies a TUBE topic map. And a back end that loads and interactively executes a program.

[2]Internally, we copy the dynamic delegate "before" the delegating topic and point to it with a delegation association.

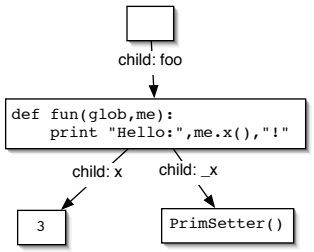

Figure 5: The root topic is the object, it has no content. Its child `foo` is a method. `foo`'s child `x` contains the default value for a parameter, whereas child `_x` allows one to assign a value to `x` via the primitive operation `PrimSetter`. We are not showing the parent associations.

```
>>> from tube import TubeEnvironment
>>> env = TubeEnvironment()
>>> env.read("paper-demo.tube")
>>> env.glob.foo(x="world")
Hello: world !
>>>
```

Figure 6: Executing the program from Fig. 5 from the Python command line.

5.1 Back End: Python

The back end is seamlessly integrated into the dynamic language Python which becomes the runtime environment of TUBE: After having started the Python command line and imported the TUBE Python module, one can load a serialized version of a TUBE program. Afterwards, every TUBE topic is represented by a Python object and serves as an interface between the world of Python and the world of TUBE. Accessing an attribute of a topic starts the locate algorithm and, if successful, returns another topic. eval is started by putting arguments in parenthesis behind a node. Example: `glob.foo()` first locates the child `foo` of node `glob` and then evaluates it. Evaluation jumps back into the Python world; executable code is always defined as a *Python* function that has the two *Python* arguments `glob` and `me`. These are TUBE topics and are the ticket back into the TUBE world. Accordingly, each *Tube* parameter `x` is accessed via `me.x`. Fig. 5 shows a small example with Python source code, Fig. 6 the interaction with the Python interpreter to run it.

5.2 Front End: TubeEditor

The front end is a separate application for editing, browsing and visualizing TUBE programs (Fig. 7). Browsing

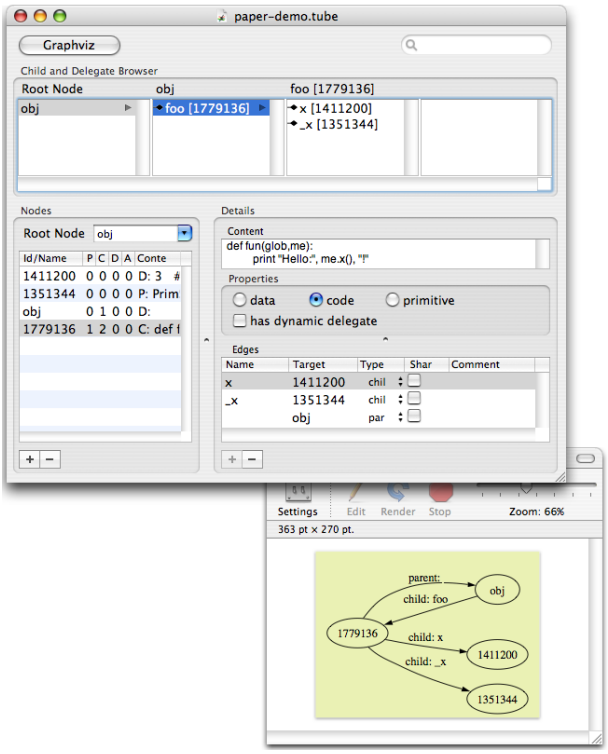

Figure 7: The window on top shows the example from Sect. 5.1 in the TUBE editor: the widget on top is a Smalltalk-style structural browser. The bottom left shows a list of all topics and the bottom right is for editing topics and associations. On bottom, a window from the GraphViz application contains a visualization that has been exported by the editor.

means traversing the child and delegate structures. In editing, one configures topics and associations. For visualization, TubeEditor relies on the external GraphViz application from AT&T. The editor is implemented in Objective C using the Cocoa framework of Mac OS X.

6 Related Work

The notion of having a runtime environment in which one can interactively construct programs and mix code with data is obviously not new. Examples of programming languages that have found very elegant ways of providing these facilities are: Lisp [7] and Self [12]. Especially the latter inspired our semantics for graphs. Self is a very mature development environment and many considerations have gone into performance optimizations. Conversely, TUBE is a prototype with emphasis on model integration and dynamic execution. Language features of TUBE that are new compared to Self are: nested lexical scope, dynamic delegates for nested overriding and complete uniformity of methods and objects as universal closures.

AHEAD *Algebraic Hierarchical Equations for Application Design* [2], a framework for large-scale generative programming has influenced the way TUBE will support multiple artifact kinds and flexible composition. AHEAD's focus on large-scale static generative programming is different from TUBE's preference for dynamic features and flexible modeling.

Starting with Java "Tiger" 1.5, Java also gets meta-data facilities [5] that point in the direction of model integration and are used for generative programming.

7 Future Research

We anticipate the following milestones in developing TUBE:

Pure Java implementation. We are going to implement the current prototype in Java, using "Jython", the Java implementation of Python. Browsing, searching and visualization abilities will be greatly extended and persistence is going to be completely topic-map-based.

Aspect-oriented programming (AOP). Nested overriding already gave a glimpse at full-featured support. We need to investigate how an existing program can and should be modified when adding an aspect. As an added benefit, using graph composition for applying aspects will allow us to compose both code and data.

Component-based programming. We are going to use the constructs introduced by the last stage and some of the modeling abilities that are already there to permit grey-box components and invasive composition [1]. We don't have the dichotomy between instances and class that is found, for instance, in Java. And our way of using associations is quite compatible with the connectors-and-ports approach used by the component community.

Meta-Programming. TUBE only having topics and associations as basic constructs is going to help us towards designing a very simple and elegant meta-protocol. Reflection will benefit in the same manner. Finally, the flexibility and non-obtrusiveness of annotations in topic maps lead to perfect hooks and parameters for meta programs.

Type system. It is clear that much functionality that should be part of a standard library, such as graph traversal, would profit greatly from a type system and related mechanisms such as multiple dispatch (used in languages such as Common Lisp [7]). Ideas in this area include light-weight types, inferred interfaces and stateful multi-methods.

Further IDE enhancements. Leverage TUBE's expressiveness to manage a complete software development process (requirements engineering, issue tracking, cross-model annotations etc.); persistent snapshots of the system for version control and for import and export of "modules"; UML integration.

8 Conclusion

In this paper, we have presented TUBE, a programming environment that integrates code and knowledge-representation-based data. This leads to many synergies: it allows us to express more modeling knowledge directly in the software system; we can integrate all artifacts that are relevant to its understanding; and code structure can be explored using professional knowledge representation tools. Picking the dynamic programming language Python as meta and object language of our prototype allowed us to write meaningful programs right from the start.

Acknowledgements. We thank our professors Martin Wirsing and Bernd Brügge for their support both professionally and financially.

References

[1] Uwe Aßmann. *Invasive Software Composition*. Springer, 2003.

[2] Don Batory, Jack Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *ACM Transactions on Software Engineering (TSE)*, 30(6):355–371, June 2004.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

[4] M. Biezunski, M. Bryan, and S. Newcomb. Iso/iec 13250, topic maps (second edition), 2002.

[5] Joshua Bloch et al. A Metadata Facility for the Java Programming Language, 2002. Java Specification Request 175.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2000.

[7] P. Graham. *ANSI Common LISP*. Prentice Hall, 1995.

[8] Donald E. Knuth. Literate programming. *The Computing Journal*, 27(2):97–111, 1984.

[9] J. Miller and J. Mukerji, editors. *MDA Guide Version 1.0.1*. Object Management Group, 2003. OMG Document: omg/2003-06-01.

[10] J. Park and S. Hunting, editors. *XML Topic Maps*. Addison-Wesley, 2003.

[11] A. Rauschmayer and P. Renner. Knowledge-Representation-Based Software Engineering. Technical Report 0407, Ludwig-Maximilians-Universität München, Institut für Informatik, May 2004.

[12] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 227–242, 1987.

[13] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3):223–242, July 1991.

[14] G. van Rossum. *Python Reference Manual*. PythonLabs, May 2004. Release 2.3.4, `http://docs.python.org/ref/ref.html`.