

A Unified Approach to Modeling and Programming

Ole Lehrmann Madsen¹ and Birger Møller-Pedersen²

¹ Department of Computer Science, Aarhus University and the Alexandra Institute
Åbogade 34, DK-8200 Århus N, Denmark
`ole.l.madsen@cs.au.dk`

² Department of Informatics, University of Oslo
Gaustadalléen 23, N-0371 Oslo, Norway
`birger@ifi.uio.no`

Abstract. SIMULA was a language for modeling and programming and provided a unified approach to modeling and programming in contrast to methodologies based on structured analysis and design. The current development seems to be going in the direction of separation of modeling and programming. The goal of this paper is to go back to the future and get inspiration from SIMULA and propose a unified approach. In addition to reintroducing the contributions of SIMULA and the Scandinavian approach to object-oriented programming, we do this by discussing a number of issues in modeling and programming and argue³ why we consider a unified approach to be an advantage.

1 Introduction

Before the era of object-orientation, software development suffered from the use of different approaches, languages, and representations for analysis, design and implementation as e.g. when using the methodologies of structured analysis and design (SA/SD) [25]. One of the strengths of object-orientation is that it provides a unified approach to modeling as well as to programming, including both a conceptual framework and a set of language mechanisms in support of this.

Current mainstream object-oriented software development, however, seems to be going in the direction of separation of modeling and programming. The numerous technologies and books on object-oriented programming are primarily concerned with the technical aspects of programming and pay very little attention to modeling aspects. New programming languages are apparently defined with no concern for modeling. For modeling languages like UML [9] we see three different developments: informal models in languages that have to be tailored (UML profiled) to specific execution platforms or domains in order to be able to generate more than just skeleton code, executable models in e.g. executable

³ This paper is a position paper accompanying a keynote speech and is therefore not a traditional scientific paper.

UML (fUML) [8], and Domain Specific Languages (DSLs) that are executable by virtue of lending themselves to a specific domain (and often to a framework implemented in some programming language).

Although informal models may play an important role in the development process and DSLs have their mission for domains with specific requirements on syntax and semantics, our main concern is modeling and programming in general purpose languages, as e.g. represented by UML and programming languages like C++ [23], Java [7], and C# [10].

If models are executable one may ask what the principal difference is between a programming language and a modeling language. If the intention is that the users of an executable modeling language are not supposed to handle any generated code in a programming language, then users will require tools for this modeling language that are comparable with the best tools for the best programming languages. They will also require the powerful mechanisms that recently have been added to programming languages (e.g. generics, classes and functions combined, and functions as parameters). Users will also expect to have programming language mechanisms that are used in everyday programming. Otherwise the situation will be (as is the experience with incomplete code generation from models) that code will stay the main artifact, and when things become critical, the model is often dropped and only the code is further developed.

Another issue is efficiency. A lot of effort has been put into making compilers for programming languages that exploit the execution platforms to a maximum. In order to compete, the same would have to be done for an executable modeling language. The alternative would be to rely on code generation to a programming language with efficient code generators, but that implies the risk of ending up with two artifacts.

While programming languages have developed a number of mechanisms not yet found in modeling languages (see above), modeling languages similarly have developed a set of mechanisms that have not found their way into mainstream programming languages (like associations and state machines). In this way programming languages may be seen as a driving force for the development of modeling languages and vice versa.

It has often been emphasized that one of the strengths of object-oriented *programming* is the ability to represent (model) phenomena and concepts from the application domain. Others have questioned the modeling aspect [1, 2] by noting that just a small percentage of the code relates to the real world. It is, however, not just modeling of real-world phenomena and concepts that is important – in any application domain, modeling is important, be it the domains of drivers, communication protocols, etc. [19].

All of this makes the case for a unified programming and modeling language, or a programming language with modeling capabilities. Distinguishing between programming and modeling languages blurs the fact that all programming should be modeling in the appropriate domain. Programming should not just be a technical issue about instructing a computer. *To program is to understand*: if models are described in a separate language then programming easily downgrades to just

“*getting away with it*”⁴ and one loses the advantage of a tight coupling between programming and modeling.

The dual support for modeling and programming may be traced back to SIMULA [4] where one of the main goals was to provide a language for modeling as well as for programming. The focus on modeling and programming has been one of the main characteristics of the Scandinavian School of object-orientation. The authors were involved in making BETA [18], another representative.

The overall purpose of this paper is to go back to the future and get inspiration from some of the ideas, goals and strengths of SIMULA with respect to modeling and programming and outline a unified approach to modeling and programming.

The approach will have to be based upon an analysis of current mainstream programming and modeling languages, identifying candidate elements that should be supported by such a unified approach and how they should be supported by language mechanisms. Similarly, the approach will have to identify and understand programming language mechanisms that do not apply for modeling – and the other way around. Low-level implementation mechanisms may not apply to modeling just as non-executable mechanisms cannot directly become part of a programming language.

2 The Scandinavian Approach to Object-Orientation

2.1 The Contributions from SIMULA

It is well-known that SIMULA (developed by Ole-Johan Dahl and Kristen Nygaard in the sixties) is the first object-oriented programming language, but it is less well-known that the language was designed with support for both programming and modeling, and that it formed the basis for a pure modeling language, DELTA, already in 1973. Nygaard originally worked with operations research and his main motivation for designing a programming language was that he then could describe computer simulations. Dahl on the other hand was a computer scientist with an exceptional talent for programming.⁵ Together they formed a unique team that eventually led to the first SIMULA language, SIMULA I, which was a simulation language. Dahl and Nygaard realized that the concepts in SIMULA I could be applied to programming in general and as a result they designed SIMULA 67 – later on just called SIMULA.

The SIMULA I [3] report from 1965 opens with the following sentences:

The two main objects of the SIMULA language are:

- *To provide a language for a precise and standardised description of a wide class of phenomena, belonging to what we may call “discrete event systems”.*
- *To provide a programming language for an easy generation of simulation programs for “discrete event systems”.*

⁴ Free from Kristen Nygaard

⁵ Kristen Nygaard in his obituary for Dahl [20]

As it may be seen, SIMULA should support *system description* as well as *programming*. At that time the term system description was used in a way similar to the term modeling today.⁶

SIMULA contains many of the concepts (object, class, subclass, virtual method, etc.) that are now available in mainstream object-oriented languages including UML. An exception is the SIMULA notion of active object with its own action-sequence, which strangely enough has not been adopted by many other languages (one exception is UML). For Dahl and Nygaard it was essential to be able to model concurrent processes from the real world application domains.

SIMULA users often experienced that they learned more from creating a SIMULA description of their system than from the actual simulation results. Nygaard together with Erik Holbæk-Hanssen and Petter Håndlykken therefore decided to develop a language, DELTA [11], purely for system description.

DELTA was based on the essential mechanisms from SIMULA. In addition, it had mechanisms for expressing true concurrency – as opposed to the pseudo parallel processes (coroutines) in SIMULA, and so-called time-consuming actions for describing continuous state changes over time – as opposed to the discrete event mechanisms supported by SIMULA. DELTA contained most of the mechanisms found in SIMULA but non-executable mechanisms were added to higher the level of descriptions. DELTA thus provided mechanisms that are beyond programming in the sense that a DELTA description may in general not be executed. DELTA did not get widespread acceptance outside a small community in Scandinavia, but in many aspects it was ahead of its time compared to subsequent work on specification languages and modeling languages.

Based upon SIMULA and DELTA, BETA was designed to be a language that could be used for design as well as for programming. BETA was, however, designed more than 25 years ago (first published paper was at POPL'83 [13]). Since the design of BETA, new requirements to modeling (and programming) have emerged, but we think that the experience from the design and use of BETA may be useful for a unified language approach.

A main contribution of BETA besides the language was the development of a conceptual framework for object-oriented programming, and such a conceptual framework is essential for modeling.

2.2 The Scandinavian Approach to Modeling

In BETA (and SIMULA and DELTA⁷) the *program execution* is considered to be a (*physical*) *model* of some *referent system* (part of the application domain). The *program text* is considered to be a *description* of the model (program execution) and thereby a description of the referent system. Note that the program execution (i.e. the dynamic process taking place during execution of the program) is considered to be the model. The program text is not a model but a description of the model. At the time of SIMULA this was a common interpretation of the

⁶ In the following the two terms may thus be used interchangeably.

⁷ Slightly differently formulated since DELTA is not executable.

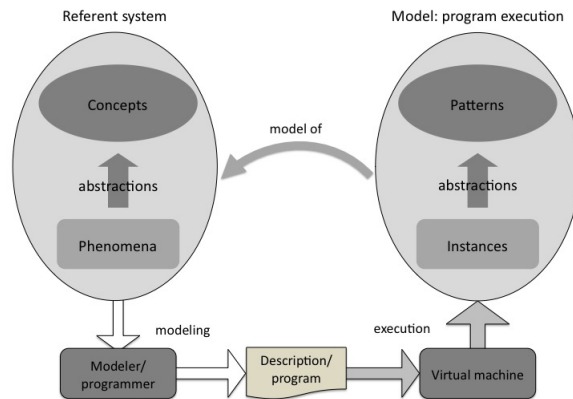


Fig. 1. Relationship between referent system, model and description

term *model*. This is in contrast to UML where the UML description is considered to be the model. The BETA model would in UML terms correspond to what would be generated at M0 (with executable UML), while a BETA program would correspond to a UML model at M1.

An analogy is a model train. The model of the real train station (or the one to be built) is the one consisting of tracks, switches, lights and small trains that move on these tracks. The model is not the description of how to build this train station.

At some point in time during the development of object-oriented methods and modeling languages, probably in connection with the introduction of graphical descriptions, model became the term used for the diagrams or descriptions. A box in a diagram representing really only a description of a part of a system may easily be confused with a model of the same part.

An essential part of the BETA project was the development of a conceptual framework for object-oriented programming. The purpose of the conceptual framework is to provide conceptual means for understanding and organizing knowledge about phenomena and concepts from the real world (application domain). And also to be explicit about the kind of properties of the real world that can be described in state-of-art object-oriented languages and in this way also serve as a driving force for the development of new abstraction mechanisms. Further development of a conceptual framework should be central in research on modeling. The reader is referred to [17, 18, 14] for a further elaboration of the modeling aspects of BETA.

2.3 Implications for Language Design

Apart from the direct support for active objects with their own action sequences, one of the main characteristics of the Scandinavian modeling approach to object-

oriented modeling is that classes model application domain *concepts*, and subclasses therefore specialized domain concepts. An implication for language design is that classes are used for defining types and not just for implementing types. Other approaches to object-oriented programming maintain the distinction between classes and types, while modeling languages have followed the Scandinavian approach with classes and subclasses modeling domain concepts.

The implication regarding the program execution as a model is that language mechanisms are designed in order to describe the desired properties of model elements. In the same way as the mechanism of class is made in order to describe concepts, the mechanism of object is made in order for objects to be models of *phenomena*. For the design of DSLs this has become obvious. When making a language for describing e.g. train control systems one start by identifying the phenomena of train controls systems (tracks, switches, lights, etc.) and their properties, and then design language mechanisms that suits the purpose of describing such systems. For general purpose languages this will amount to identifying the desired properties of phenomena in general and then devise language mechanisms for this. As an example, when designing SIMULA it was important to describe phenomena with their own action sequence. Value properties of phenomena lead to attributes of objects (developed as part of the record concept prior to SIMULA), while potential behavior properties lead to procedures (methods) belonging to objects. Another example is that in BETA it was possible to describe that objects were parts of another object. For a programming language this is not needed (can be done by a const mechanism), while composition (as also found in UML) may be important for modeling.

3 Language Design Issues

As mentioned, modeling languages and programming languages have a common core of concepts, language constructs and other issues. In addition, there are a number of issues normally associated with modeling just as there are issues normally associated with programming. We believe that most of these issues apply just as well to modeling as to programming. In this section we discuss a number of general language design issues and in the next section we discuss concrete language constructs. We believe that research in modeling and programming languages will benefit from identifying these issues and discuss them in a broader perspective. Issues like scope rules, type systems, semantics, etc. that obviously apply to both modeling and programming are not discussed in this paper.

3.1 Syntax

For mainstream languages, programming languages most often have a textual syntax while modeling languages have a graphical syntax. There is, however, no law or other justification for this. There are several examples of programming

languages with a graphical syntax, and we shall later argue that a textual syntax may also be useful for a modeling language.

One may argue that a graphical syntax is better suited for communication than a textual syntax. Since modeling languages are primarily used in the initial phases of a project, the communication aspect is important. This is the case for communication in a team, between teams, with managers and perhaps also with customers. All of this argues for a graphical syntax. There is the well known phrase: *A picture says more than a thousand words*. There is, however, also a saying the other way around: *A word says more than a thousand pictures*.⁸ As an example, the word “vehicle” communicates the concept in a much more efficient way than thousand pictures of vehicles. The point is that we use words to capture essential concepts and phenomena – as soon as we have identified a concept and found a word for it, this word is an efficient means for communication.

While the argument for graphical descriptions is that they are good for providing overview, the fact is that graphical descriptions quickly become unmanageable when they get large. In addition when a model becomes stable and well understood, a textual description is more compact, and at that stage of a project there is no problem for the developers to work with at textual description of a model.

One of the benefits of object-oriented programming is that it has provided a common set of language constructs for programming as well as modeling. Objects, classes, subclasses, virtuals, etc. are core language elements in programming as well as modeling languages, and these are independent of what kind of syntax is used.

In our opinion text or graphics are just a matter of syntax, i.e. both types of language should/could have a textual as well as a graphical syntax. In addition, we find it useful to be able to mix textual and graphical descriptions in the same page/diagram/window. SDL [12] is an example of a language that has both a textual and a graphical syntax. In addition, it is possible to mix text and graphics in the same descriptions.

The original design of BETA was based on a textual notation, but later a graphical notation was added. The Mjølner tool [22] provided an integrated text, structure and graphical editor. The textual syntax and the graphical syntax were just different presentations of the same underlying abstract syntax tree, and the user could switch between these presentations. There was thus a one-to-one correspondence between the textual and the graphical syntax. This was the case for the abstraction mechanisms, but of course not for low-level control structures, assignments and expressions. Later, the graphical notation was changed into UML, the rationale being that this was a de facto standard graphical notation, but this created impedance problems. For a unified language this should not be an issue, and the one-to-one-correspondence principle is the only option.

As mentioned, SIMULA was designed as a modeling and a programming language, but based on a textual syntax. One may wonder whether or not a graphical notation was an issue. Graphical illustrations have been in common use since

⁸ Quote by Kristen Nygaard

the early days of programming, for documenting programs. This was of course also the case for SIMULA. Such illustrations were informal and the intention was to give the reader an idea about the structure of a given program. Current mainstream modeling languages have a formal graphical syntax and this is fundamentally different from using informal drawings for illustrating programs. We realize that some form of standardization of diagrams is necessary, but insisting on a formal graphical syntax for all usages of graphics may not be a good idea.

3.2 Constraints

Some of the elements of modeling languages impose constraints on the elements of the model. Constraints have also been an issue for programming languages for decades. At a general level, constraint languages impose structural relations or state changes by means of equations and/or relations. A language like Prolog may be seen as an example of a programming language based on equations. For object-oriented programming the work of Alan Borning, Bjorn Freman [5] and others are well-known examples of adding constraints to object-oriented languages. One of the first attempts was to use equations for stating the relations between graphical elements in a window. This is an example where equations often are easier to understand than a set of imperative statements.

Constraints have not made it into mainstream programming languages. The reason may be twofold: (1) it is in general not possible to solve arbitrary equations and/or relations – it is necessary to impose some kind of restrictions upon the kind of equations/relations that can be used, (2) constraints are rarely primitive language elements. They require a constraint solver that may be more or less complex. And since there is a broad variety of the kind of restrictions that can be imposed on the equations/relations, constraint mechanisms and their corresponding solver merely seems to belong to the library/framework level. The difficult part is to identify which primitive language elements to add to a given language in order to support constraints.

3.3 Domain Specific Languages

The notion of Domain Specific Language (DSL) is an issue often associated with modeling languages and therefore with graphical syntax. However, the idea of domain specific languages is orthogonal to programming and modeling languages. It has been an issue long before modeling languages were considered.

When SIMULA 67 was generalized (from the simulation language SIMULA I) to become a general purpose programming language, the simulation capabilities of SIMULA I were defined in a framework in terms of a predefined class `Simulation`. Class `Simulation` was the first example of an application framework and it is also considered an example of a DSL embedded within SIMULA. SIMULA actually has a special syntax that only applies together with class `Simulation`.

In general any application framework may be considered a DSL, and most often a DSL comes along with a framework. We do, however, not advocate

defining a special syntax for each application framework. If several frameworks are used together, different syntaxes may be confusing. There are, however, cases for DSLs in very specific application domains where special syntax is required.

While DSLs started in languages with textual syntax (embedded or standalone languages), the syntax for DSLs within modeling is often graphical, the reason being that special kinds of well-established notations have to be supported. Examples are DSLs for feature modeling, where feature diagrams have to be supported, and TCL [24], a train control language, where the descriptions have to have the forms of tracks, switches, lights, etc.

For standalone languages, it is straightforward to associate a graphical syntax with the defining framework. For embedded DSLs we suggest that techniques for embedding a graphical language into a host (graphical) language be developed. This may also be used with complex language elements like state machines and associations which could then be defined as frameworks with an associated embeddable graphical syntax. We return to this in Sect. 4.

3.4 Object Models and Scenario Descriptions

Interaction diagrams like communication- and sequence diagrams in UML are useful for describing the essential scenarios of a given system. In a similar way, object models (instance specifications) in UML (at level M1) are useful for describing snapshots.

As mentioned, we consider the execution (level M0 in UML) to be the model, and therefore we think that language constructs for describing snapshots of the execution including objects and their relationships are important. Few programming languages include support for describing snapshots.

For the same reason it is a good idea to define a notation for describing snapshots/scenarios that is consistent with the language independent of whether it is a modeling or programming language or a combination. A tool may then check the consistency between scenario descriptions and the program/model just as the notation may be used in debuggers. Currently only modeling languages like UML seem to have this – few if no programming languages have an associated scenario notation. With the current use in UML, the language will have to have the notion of composite structure in order to provide a context for e.g. sequence diagrams with lifelines corresponding to the parts of such a composite structure.

3.5 Programming by Examples

When one or more scenarios have been made, one has to design a description of a model that covers the scenarios in the sense that the scenarios correspond to the model. In order to support this, it will of course be useful to be able to construct the model description more or less automatically from the scenarios. There are several authors that present techniques for deriving models from scenario descriptions. This includes techniques to construct state machines from sequence diagrams.

This is, however, not only an issue for modeling. It may be seen as analogous to programming by examples. The literature contains many examples of papers that present techniques for deriving programs from examples. One example is the paper by Henry Lieberman from 1982 [15].

3.6 Miscellaneous

UML has a number of mechanisms like *Components* and *Deployment* that we do not consider in this paper. Most of these are just as relevant for programming as for modeling. Components are just special objects (just like in UML), and programs also have to be deployed.

Use Cases appear to be quite separate from the rest of UML and they might just as well be used together with a programming language. There might, however, be a case for a tighter integration with sequence diagrams and thus with the execution, but for some reason this was not done when a major revision of UML (UML2) was made.

4 Language Concepts

4.1 State Machines

State machines are common for modeling, but have never become mainstream in programming languages. A common approach is to use the *design pattern* as described in [6]. In practice, however, it is necessary to be able to alternate between modeling and programming. When using the state-pattern it may be difficult to identify a given state machine in a program and good tools for code generation and reverse engineering are therefore required.

Another common approach is to define a state machine *abstraction* in a class library. This puts requirements on the programming language with regard to the ability to define a suitable abstraction for state machines. In order to support a graphical syntax for a state machine abstraction it is necessary to be able to associate such a syntax with the abstraction/library. As mentioned in Sect. 3.1, general support for embedding graphical syntax in a host language may be a solution.

Direct support by means of language constructs is of course a possibility. In [16] a mechanism for changing the virtual binding of a class is suggested as a means for supporting state machines. With direct language support it is of course straightforward to support a graphical syntax.

State machines are complex entities – this holds for simple state machines, but even more when composite states are included. This may imply that direct language support is not the the best solution – the language should be powerful enough to define the appropriate abstraction(s).

4.2 Associations

Associations are perhaps the modeling mechanism that is most often mentioned as lacking proper support in programming languages. In 1987 Rumbaugh [21] proposed programming language support for associations, and since then there have been several alternative proposals.

The reason that no mainstream programming language has direct support for associations may be due to the complexity of associations. Rumbaugh has a figure that shows the location of associations in the hierarchy of containers. This may be taken as an indication that associations should perhaps not be a built-in mechanism. It is, however, difficult to identify one or more less primitive mechanisms (besides references) that should be included instead. In addition to suggestions for language support, there are many examples of proposals for association libraries as in [26].

We are not aware of design patterns for supporting associations in general. There are specific design patterns for e.g. composite as in [6]. This may be an indication of the complexity of associations. There are in fact many ways to define such abstractions, i.e. lack of standardization. The latter is perhaps due to the fact that UML is open for interpretation or perhaps that there is a need for variation here.

As for state machines we think that the best approach is to define abstractions (in class libraries) that support associations. This is not possible in most mainstream languages, so it is a challenge for programming language design. With respect to graphical syntax we again advocate to develop support for embedding a graphical syntax into a host language.

4.3 Asynchronous Events

Messages in mainstream object-oriented languages are synchronous in the sense that a method invocation blocks until the entire message has been executed and a possible value has been returned.

UML state machines have events that are both reception of asynchronous signals and methods calls. Most users of state machines in UML are using them in application domains where asynchronous signals are the means of communication, e.g. telecom or process control. fUML recognizes this by only supporting signals as events, while method calls are executed independently of the state of the object. This is in contrast to e.g. state patterns proposed for object-oriented programming languages, where method calls are the events.

In a unified language there seems to be two options: (a) support both kinds of events and make both of them controlled by states, (b) select one of them and provide the other as a framework on top of the language (not all modeling mechanisms have to be language mechanisms, in programming languages one is used to make frameworks instead of new language constructs). Starting with object-oriented programming (and not with executable UML), the most obvious choice is perhaps to have method calls as events, while for a unified language the choice may not be that obvious.

For many years the support for asynchronous method calls has been an issue for object-oriented programming platforms as well. There are numerous programming languages that support asynchronous method calls. Programming languages with asynchronous communication are often based on actors. Actors are objects with their own thread of control. They share no state with other actors; they communicate exclusively via asynchronous message passing.

Mapping of external events into synchronous or asynchronous method calls or more direct support in the language is also an independent issue.

4.4 Action Sequences

With respect to action sequences there is in general a major difference between programming languages and modeling languages. Programming languages are based on well established (sequential) statements such as assignments and control structures. For concurrent programming there is much less consensus – in fact concurrent object-oriented programming seems to be caught in a tar pit of low-level technical details of how to avoid locking problems, memory problems, and efficient use of thread-safe libraries. If one recalls the original goals of SIMULA as a modeling language that was also able to describe concurrent processes from the real-world, mainstream concurrent programming seems far away from supporting modeling.⁹

For modeling languages the picture seems pretty blurred in the sense that there are many suggestions for describing action sequences, including state machines, active objects, activity modeling, etc. Compared to programming languages, there is clearly an attempt to model action sequences at a higher level. This also holds for concurrent processes. UML 2 activities are typically used for business process modeling, for modeling the logic captured by a single use case or usage scenario, or for modeling the detailed logic of a business rule.

We do not think that the various proposals for modeling (sequential and concurrent) action sequences in say UML have matured to a point where they are usable in practice and can replace traditional programming language statements. We do, however, advocate that the same means for describing action sequences should be used for programming as well as modeling. We especially advocate a modeling approach to concurrency since we need more high-level concurrency abstractions in order to deal with the future of multi-core processors.

4.5 Other Language Constructs

In UML an object can be a member of several classes, which is not possible in mainstream programming languages. This is desirable for modeling as well as programming. In Chapter 18 in [18] we discuss a number of means for classification, including multiple classification and dynamic class membership.

⁹ We do not claim that SIMULA is the solution to concurrent programming – SIMULA may be able to describe real world processes, but has no support for synchronization of true parallel processes.

We have mainly discussed language mechanisms that are often associated with modeling and argued that they are just as relevant for programming. In a similar way, there are a number of programming language mechanisms that have not found their way into modeling languages. These include statics, metaclasses (reflection), generics, traits, (higher order) functions, general block-structure (arbitrary nesting of classes and methods), aspects, modules, and many more. If such mechanisms are useful for programming they are probably also useful for modeling. We do (of course) not argue that a unified language should contain the union of all possible language constructs. The point is that if a construct has proved useful for programming, it might as well be useful for modeling and vice versa.

5 Conclusion

We have argued for a unified approach to modeling and programming based on the fact that this was one of the strengths of object-orientation as provided by SIMULA. We think that the benefits of a unified approach are fading away since the design of programming languages and modeling languages seems to be more or less separate activities in different communities. If programming and modeling are separate activities, we will constantly be confronted with impedance mismatch between different representations just as programmers may not be concerned with modeling and vice versa. Eventually the code will win, either due to lack of time in a given project and/or because the code is the real thing.

We have also argued that programming is a modeling activity even in technical domains that are not related to the so-called real world. In our opinion any object-oriented program should reflect the concepts and phenomena in the given domain and the basic core of object-orientation includes a conceptual framework as well as a set of language constructs that facilitate modeling. We have also argued that further development of the conceptual framework for object-orientation is an important activity within modeling and programming.

The approach to modeling has implications for language design as well as for the design of notations for illustrating scenarios and including the model (program execution).

We have discussed a number of issues often associated with modeling and argued that most of these issues are relevant for programming as well. The other way around, most issues relevant for programming are also relevant for modeling. One conclusion is that programming and modeling may benefit from a unified approach.

For modeling there may be a need for concepts at a higher level than primitives in programming languages. Examples are state machines and associations. A programming language should support defining these concepts as abstractions (class libraries). We acknowledge that current state-of-art of language abstractions may not be sufficient to define such abstractions. This calls for further research in order to develop more powerful abstraction mechanisms.

We have argued that a textual or graphical syntax is relevant for modeling and programming, and we have argued for techniques to support embedding of graphical syntax within a host language. This is especially relevant to support graphical notations associated with abstractions for e.g. state machines and associations.

Modeling languages may contain non-executable parts that express constraints on the elements of the description. With a unified approach there is of course still a need for such non-executable elements, but these should be integrated with the language. There are programming languages that have support for non-executable elements like invariants, assertions, and pre- and postconditions just as there are so-called constraint-based languages. There is thus good reason to integrate non-executable parts into a unified modeling and programming language.

Finally, there are issues related to the model itself (where model is the program execution). Modeling languages include elements for describing scenarios and object models. Bearing in mind that the model is the program execution, it is of course important that language elements for describing the model in terms of scenarios and/or objects are treated as a first-class issue and not left to just be the design of a debugger. We need means for presenting the model.

References

1. Steve Cook. Object Technology – A Grand Narrative? In Dave Thomas, editor, *ECOOP 2006 – European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, Nantes, France, 2006. Springer Verlag.
2. William Cook. Peek Objects. In Dave Thomas, editor, *ECOOP'2006 – European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, Nantes, France, 2006. Springer Verlag.
3. Ole-Johan Dahl and Kristen Nygaard. SIMULA—a Language for Programming and Description of Discrete Event Systems. Technical report, Norwegian Computing Center, 1965.
4. Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based Simulation Language. *Communications of the ACM*, 9(9):671–678, 1966.
5. Bjørn N. Freeman-Benson and Alan Borning. Integrating Constraints with an Object-Oriented Language., June 29 - July 3 1992.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
7. James Gosling, Bill Joy, and Guy. Steele. *The Java (TM) Language Specification*. Addison-Wesley, 1999.
8. Object Management Group. Semantics of a Foundational Subset for Executable UML Models FTF Beta 2, 2009.
9. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3, 2010.
10. Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley, 2003.

11. E. Holbæk-Hanssen, P. Håndlykken, and K. Nygaard. System Description and the DELTA Language. Technical Report Report No 523, Norwegian Computing Center, 1973.
12. ITU. Specification and Description Language (SDL), Recommendation Z.100, ITU T., 1999.
13. B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Abstraction Mechanisms in the BETA Programming Language. In *Tenth ACM Symposium on Principles of Programming Languages*, Austin, Texas, 1983.
14. Bent Bruun Kristensen, Ole Lehrmann Madsen, and Birger Møller-Pedersen. The When, Why and Why not of the BETA Programming Language. In Brent Hailpern and Barbara G. Ryder, editors, *History of Programming Languages III*, San Diego, CA, 2007. SIGPLAN.
15. Henry Lieberman. Designing Interactive Systems From The User's Viewpoint. In P. Degano and E. Sandewall, editors, *Integrated Interactive Computing Systems*, Stresa, 1987. North-Holland.
16. O. L. Madsen. Towards Integration of Object-Oriented Languages and State Machines. In *Technology of Object-Oriented Languages and Systems (TOOLS Europe '99)*, Nancy, 1999.
17. O. L. Madsen and B. Møller-Pedersen. What Object-Oriented Programming May Be—and What It Does Not Have to Be. In S. Gjessing and K. Nygaard, editors, *ECOOP'88 – European Conference on Object-Oriented Programming*, volume 322 of *Lecture Notes in Computer Science*, Oslo, Norway, 1988. Springer Verlag.
18. O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley, 1993.
19. Ole Lehrmann Madsen. From ECOOP'87 to ECOOP 2006 and Beyond. In Thomas Dave, editor, *ECOOP 2006 – European Conference on Object-Oriented Programming*, volume LNCS 4067 of *Lecture Notes in Computer Science*, pages 186–191, Nantes, France, 2006. Springer.
20. K. Nygaard. Ole-Johan Dahl. *Journal of Object Technology*, 1(4), 2002.
21. J. Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In N. Meyrowitz, editor, *OOPSLA '87 – Object-Oriented Programming, Systems Languages and Applications*, volume 22 of *Sigplan Notices*, Orlando, Florida, USA, 1987. ACM Press.
22. E. Sandvad. An Object-Oriented CASE Tool. In J. L. Knudsen, M. Löfgren, O. L. Madsen, and B. Magnusson, editors, *Object-Oriented Environments—The Mjølner Approach*. Prentice Hall, 1994.
23. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading MA, 1986.
24. A. Svendsen, G.K. Olsen, J. Endresen, T. Moen, E. Carlson, K.-J. Alme, and O. Haugen. The Future of Train Signaling. In *Model Driven Engineering Languages and Systems (MODELS 2008)*. Toulouse, 2008.
25. Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press Computing Series, 1979.
26. Kasper Østerbye. Design of a Class Library for Association Relationships. In *LCSD'07*, Montréal, Canada, 2007.