Software and Systems Modeling (2022) 21:2097–2099
https://doi.org/10.1007/s10270-022-01057-2

EDITORIAL

Reflection on the differences between modeling and programming

Jeff Gray[1] · **Bernhard Rumpe**[2]

Published online: 1 November 2022

For a 2012 editorial [1], the relationships between modeling and programming languages were discussed. A decade later, it seems appropriate to revisit this issue because there is still not a widely accepted consensus on the main purpose of modeling languages and their models in software development. In this editorial, we highlight the most important aspects when identifying commonalities and differences between modeling and programming languages. We are fully aware that a detailed examination of all these relationships, properties, and their interdependencies is far beyond this short reflection.

An important property of a program is its ability to be **executed**. Many models are also created with executability in mind. An executable model must incorporate all relevant aspects, including structure and behavioral properties, such that a code generator or interpreter can derive executable code from the model. However, models sometimes also serve different purposes that programming languages do not handle well or not at all, which is why modeling languages for software development are needed.

Abstraction is often better served by appropriate modeling languages, where abstraction mechanisms are part of the language and defined within the model. Examples are superstate, sub-state mechanisms in Statecharts, or simulation relations between Statecharts, or logical implication that can be applied as a refinement relation between logical formulae, such as preconditions or invariants. Programming languages mainly concentrate on super/sub-classes in object-oriented languages when introducing abstraction mechanisms.

Along with abstraction, an important need is the ability or even necessity to specify structural and behavioral properties that are not directly executable. **Underspecification** is an intrinsic abstraction concept that programming languages do not offer. Underspecification permits a broader range of possible behaviors, in comparison to a program that in the same situation always describes the identical deterministic behavior (we ignore a random function as an ineffective mechanism to mimic underspecification). Underspecification is needed because during development, not all requirements are already known and sometimes design decisions need to be delayed until further information is available. Examples are choices of (non-deterministic) transitions in Statecharts, groups of actions in activity diagrams (where their sequential execution is not yet defined), intervals of time delays for execution, or undetermined types for attributes in conceptual models.

Support for **underspecification** enables a language to offer **refinement** techniques such that a specification model is refined into a more detailed specification or even a fully determined implementation. Refinement has several forms, for example being expressed through language constructs within the models (e.g., sub-/super-classes in class diagrams) or as operators between models, such as replacing an abstract behavioral specification through a more detailed one.

Variability in software product lines can be expressed using underspecification techniques, because an underspecified, generalized specification for structural and behavioral aspects of a variability point can be refined through a set of available feature realizations. When using specification and refinement, features (then by construction) are correct implementations of their variability point. Programming languages provide variability mainly at the structural level of super- and subclasses and provide the possibility to override methods and their behavior in an unconstrained way. It is a well-known problem that a programming language only provides deterministic implementations of methods, but no general mechanism for specifying the desired behavior (with Bertrand Meyer's Eiffel and its use of pre-/postcondition concepts as a praiseworthy exception).

Often, it is an advantage that models may have less **precision and detailedness** than program code. However, it must not be forgotten that the precision of the model does not necessarily correlate with the precision of the language

✉ Bernhard Rumpe
bernhard.rumpe@sosym.org

Jeff Gray
jeff.gray@sosym.org

[1] University of Alabama, Tuscaloosa, Alabama, USA

[2] RWTH Aachen University, Aachen, Germany

used—*very imprecise models can be created in a very precise language*. Yet, it is challenging to create a precise model using an imprecise language (e.g., a natural language).

When thinking precisely about refinement or synthesizing code from models, the precision of a model cannot be different from the precision of the generated code. This requires a precisely defined and unambiguous modeling language. Of course, models are sometimes only used as "thought models" and remain in a developer's head as an unclear level of precision without an explicit graphical, diagrammatic, tabular, or textual representation.

Furthermore, the degree of **compactness** plays a major role in a model's practical use. Sophisticated language constructs and potential library elements may reduce the size of a model specification and lead to more effective development due to the reduction of repetitive and verbose overstating of model properties.

UML, SysML and many domain-specific languages (DSLs) have shown that models and their various forms support the definition of explicit **views** on various different aspects of the final system to be implemented. Programming languages focus on the statically definable structural part, which are classes, attributes, methods, and things like singleton objects defined within static variables. UML allows the explicit modeling of state-based behavior, interaction sequences, interaction patterns, activities and their logical dependencies, and component structures. However, there are many different forms of usage for this kind of view; e.g., a constructive form to derive or synthesize code or also in a testing form, by checking the validity of the implementation against the specification. Such synthesis and testing activities can be done during design time (generation, testing) or during runtime (interpreting, monitoring) and may have various consequences. Although a modeling language has more expressivity in terms of these views, it depends on the tooling used and how to deal with these views constructively, analytically or in verificational form. Programming languages generally do not offer these different options.

Considering these differences, shared characteristics, and uses of programming and modeling, it is obvious that there is one technical difference: a programming language is a coherent, clearly defined and relatively stable homogeneous language. In contrast, the various aspects and viewpoints in a modeling language should span the phases of early requirements, through high-level and detailed design, up to implementation, testing, and verification. This broader coverage often requires the use of modeling sub-languages that are appropriate for the level of precision needed at each stage of development. When describing a software system, a **heterogeneous** set of models must consistently work together. This complicates development of sophisticated tooling in the modeling domain.

It seems that programming languages have solved this issue better by incorporating smart **mechanisms of import and reuse**, which allows many concepts to be removed from the core language and deferred to external libraries. Using specific **libraries or frameworks** considerably changes the style of programming (e.g., the callback functions in GUI frameworks). Also, design patterns and especially architectural patterns (e.g., the state pattern) enforce a special form of code. These kinds of styles are also possible in modeling, but the modeling community still continues to discuss which styles are most helpful.

In summary, programming languages can influence model engineers on how to define, use and manage our models. However, there are some major differences, and in particular various needed forms of abstractions during early phases of development that do not have a strong analogy with programming languages. Agile development and other methodologies that support development of larger, complex systems might benefit from a dedicated and explicit modeling activity if done well and assisted by useful tools.

1 Content of this issue

1. **Expert Voice**
   - "Modeling should be an independent scientific discipline" by Jordi Cabot and Antonio Vallecillo

2. **Theme Section on Model-Driven Requirements Engineering**
   - Guest editors: Ana Moreira, Gunter Mussbacher, João Araujo, and Pablo Sánchez

3. **Regular Papers**
   - "An investigation of the relationship between joint visual attention and product quality in collaborative business process modeling: a dual eye-tracking study" by Duygu Findik-Coskunçay and Murat Perit Çakir
   - "An efficient line-based approach for resolving merge conflicts in XMI-based models" by Alfonso de la Vega and Dimitris Kolovos
   - "Instant and global consistency checking during collaborative engineering" by Michael Tröls, Luciano Marchezan, Atif Mashkoor, and Alexander Egyed
   - "Model-driven management of BPMN-based business process families" by Andrea Delgado, Daniel Calegari, Felix Garcia, and Barbara Weber

Funding Open Access funding enabled and organized by Projekt DEAL.

Reference

1. France, R., and Rumpe, B.: On the relationship between modeling and programming languages. In: Journal Software and Systems Modeling*(SoSyM)*. Springer Berlin/Heidelberg. ISSN 1619-1366. Vol. 11(1), pp. 1-2, (2012). Available at https://www.sosym.org/editorials/files/FR12a.pdf