Software and Systems Modeling (2022) 21:839–841
https://doi.org/10.1007/s10270-022-01001-4

EDITORIAL

Explicit versus implicit models: What are good languages for modeling?

Jeff Gray[1] · **Bernhard Rumpe**[2]

Published online: 7 April 2022

Although modeling is used in almost all science and engineering disciplines, the explicit definition of modeling languages is an invention of modern computing technology. It was necessary to define modeling languages precisely so that computers could process the models described in the languages, as a way to support various analysis and synthesis needs (e.g., model analysis and code generation). However, these were not the first languages that were defined precisely. Natural languages, for example English in the Oxford dictionary and German in the Duden, received a rather formal definition prior to digital automation with computers. The ideas of grammars and related lexicographic definitions were transferred from the natural language context to the first computer languages, namely programming languages and specification languages, where it received much more precision. And for diagrammatic modeling languages, the concept of a meta-model was popularized, often as a variant of a class diagram.

The precise definition of modeling languages allows us to make models explicit. The advantages of precise modeling languages are well-known, namely (E1) support for syntactically concise descriptions, (E2) attachment of precise semantics, (E3) a human-readable representation that provides a language summary, (E4) models defined by a language can be processed by computers to help identify inconsistencies, incompleteness, and other issues, and also (E5) provide the foundation for high-level analysis techniques as promoted in various forms by the formal methods community. Finally of course, other benefits also include constructive use, including (E6) automatically deriving code, tests, and related artifacts from the models, and (E7) the ability to interpret a model.

✉ Bernhard Rumpe
bernhard.rumpe@sosym.org

Jeff Gray
jeff.gray@sosym.org

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

[2] RWTH Aachen University, Aachen, Germany

Over time, a larger set of modeling and specification languages have been defined, among them the standards UML and SysML (with all their benefits and deficits), but also many logic-based languages, architectural modeling languages, testing notations, as well as deployment and configuration languages. Currently, we also can find requirements-capturing languages that are a mixture between traditional natural languages and structurally constrained textual languages, which can be regarded as precise modeling languages in a textual syntactic form.

However, there are also approaches to embedding models within existing languages. A famous example is the use of the State Pattern as one of the initial 23 design patterns from the Gang of Four in the early 1990s, where an efficient encoding of state machines in an ordinary, general-purpose programming language (GPL) is described. This embedding is very useful and can be defined directly in the GPL. When describing the layout, e.g., as a grid of a GUI AWT/Swing window, we actually specify the grid layout in a high-level abstraction, but do that within the Java PL. As a third example, class diagrams can also be encoded directly as GPL classes, with the exception of associations that need additional encoding decisions.

Modern programming languages with enhanced forms of syntactic compactness, such as Scala, support the definition of APIs in such a way that the resulting code even looks like it is not a traditional program, but rather an explicitly defined domain-specific modeling language. As one example, the tool chain integration DSL of Gradle is such an interesting approach. Like the makefile language, it looks like a DSL, but actually is an ordinary Groovy program using the Gradle library functions. Similar observations can be made with other forms of modeling languages, where a model is sometimes encoded directly in appropriate concepts of the GPL.

In other examples, a model may conform to a specific style of programming (i.e., a kind of design pattern), assisted by an appropriate predefined and reusable API, which itself is realized in a generic library and allows a relatively compact encoding of the "model." These encodings are often so

straightforward that a direct decoding, also called reverse-engineering, is possible such that the original model can be extracted from the code. Yet, the extracted model still needs an explicit representation form and thus must become an explicit member of a modeling language.

Other forms of models remain completely implicit. For example, the task dependency graph defined in Gradle and the grid layout presentation in Java Swing are just data structures that do not have explicit modeled representations beyond the encoding in the GPL. They only come to life when the code is executed, but are usually rather invisible in the code structure. A traditional reverse-engineering does not help. We thus can distinguish implicit models from statically extractable vs. constructed through execution, even though the classification might have some overlaps.

Implicit models embedded in the code of a GPL also have advantages: (I1) the toolchain only consists of an ordinary compiler (and a runtime library), which can (I2) efficiently be used to generate running and testable systems as well as simulations, and (I3) dynamic reconfigurations are possible. This supports the addition of new states or redirected transitions in the State pattern or changes in the grid layout of AWT.

Arguably, dynamic reconfiguration can also be seen as a serious challenge because analytical considerations of how a system will behave can be invalidated when arbitrary dynamic reconfigurations are possible. But there is a serious second disadvantage, namely embedded models are not amenable for explicit analysis at design time. As an example, a compiler or classic code analyzer normally does not know about a project-specific embedding of a state machine or an activity diagram in the code, and thus cannot analyze the embedded model for desired properties like completeness and consistency. It does not make a difference if the model has been defined in an explicit modeling language and code was generated, or whether the implicitly defined model was extracted into the explicit presentation. In both cases, an explicit modeling language is involved to be able to analyze the model. For analysis purposes, it may be sufficient to define the modeling language only through abstract syntax, such as a meta-model, which would allow us to actually consider the data structure of the State pattern an (incomplete) meta-model of state machines.

The most important challenge with dynamic reconfiguration is when a modeling language that is applicable to all activities of development must allow controlled underspecification of various forms. This is a necessary prerequisite for modeling, because during development we often do not know all of the technical details, and more importantly, we do not want to predefine design decisions too early so that we can explore the possible design space in later phases of the development. Modeling languages must allow underspecification and can do that in various forms, among others via (U1) offering variability mechanisms to define alternatives, (U2) parameters and configuration options that parameterize models, (U3) logical constraints to relate such parameters and the system behavior/structure, (U4) logic-based pre- and postconditions to describe bandwidths of possible realizations of behavior, and (U5) non-determinism/choice within behavioral notations that help in describing alternative behavioral decisions.

All of these concepts can be implicitly embedded in a GPL. However, programs are executable and their embedded models also are executable. In an execution, a deterministic and completely defined behavior occurs. The underspecification forms noted above as (U1), (U2), and (U5) can typically be executed through either random choice (e.g., calling a random() function) or default behavior selections (e.g., always take the first available option). The other forms of underspecification described as (U3) and (U4) can fail in constructive executions because solutions for undecidable logical expressions must be found. In all cases the "executed" model is not equivalent with the intended model and further design decisions may force other "executed" behavior that was specified, but never seen and anticipated in earlier tests.

Underspecification is relevant and cannot be handled only by executable models. However, there are solutions that have been intensively investigated by the Formal Methods community (e.g., logic reasoning, model checking, data and control flow analyses, abstract interpretation, etc.). These approaches can help to explore many or even all behavioral paths and configurations "in parallel" and can be applied when using explicit models in dedicated modeling languages.

As a consequence, however, we should accept that some modeling languages should NOT allow developers to describe all kinds of system properties, but restrict themselves in their expressivity. In general, the more restrictive a modeling language is, the greater the ability to offer effective and efficient analysis techniques. Thus, it is often a complicated balancing act between the expressivity of models and the power of available analysis techniques. At least it is clear that with pure embedding of models in a general-purpose language, i.e., implicit models, we get the advantage of simulation, but usually lose all advantages of explicit analysis.

Recently, most of the modeling techniques described in SoSyM discuss explicit modeling techniques that are based on explicit modeling languages. SoSyM, however, also welcomes papers that enlarge and improve our knowledge on modeling technologies, where implicit models may also be used. We look forward to seeing additional contributions in this area.

1 Content of this Issue

1. **BPMDS 2020 Special Section**
   Guest Editors: Pnina Soffer and Selmin Nurcan
2. **Theme Section on "AI-enhanced Model-Driven Engineering"**
   Guest Editors: Lola Burgueño, Jordi Cabot, Manuel Wimmer, and Steffen Zschaler
3. **Regular Papers**
   - "DEVS-based formalism for the modeling of routing processes" by Maria Blas, Horacio Leone, and Silvio Gonnet
   - "Model-based ideal testing of hardware description language (HDL) programs" by Onur Kilincceker, Ercument Turk, Fevzi Belli, and Moharram Challenger
   - "A generic approach to detect design patterns in model transformations using a string-matching algorithm" by Chihab Eddine Mohamed Omar Mokaddem, Houari Sahraoui, and Eugene Syriani

Funding Open Access funding enabled and organized by Projekt DEAL.