Programming ⊂ Modeling ⊂ Engineering

Bran Selić(✉)

Malina Software Corp., Nepean, ON, Canada
selic@acm.org

Abstract. The proven ability of some modeling languages to be used as both design and implementation languages has raised hopes of a seamless blending between design and implementation, a capability that only seems possible in the domain of software engineering. In turn, this has led to methodological questions on how best to take advantage of models and modeling technologies in the development of complex software systems. To provide some insight into this question, we start with a review of the role and types of models that are found in traditional engineering disciplines and compare that to current practices in model-based software engineering in industrial practice. The conclusion reached is that, despite some very unique characteristics of software, there does not seem to be any compelling reason to treat the handling of models in software engineering in a radically different way than what is done in engineering in general.

Keywords: Model-based engineering · Software engineering methodology · Modeling languages

1 Introduction

"The design *is* the implementation," we would proclaim triumphantly to our colleagues, most of whom, being experienced practitioners of software development, remained skeptical. This took place during the early days of our experience with a new computer language, which we had devised for our own domain-specific purposes. In today's terminology this language would qualify as a *domain-specific modeling language*. It was intended specifically for writing of complex real-time applications in the telecommunications space (e.g., implementing communications protocols) [12]. The language had all the features of a truly modern high-level language: it was object oriented and component based. The *high-level structure* (architecture) of a system was specified by hierarchical networks of interconnected parts captured as reusable classes, while the *high-level behavior* of these classes was specified using a simplified custom variant of Harel's statecharts [4]. It departed from traditional programming language conventions of the time (1980's and 1990's) in that it relied fundamentally on a visual syntax for rendering its high-level concepts. Since these were based on graph-like formalisms, a graphical syntax was a natural choice. This allowed us to bypass the tedious and error-prone manual translation of such specifications into equivalent linear programming language text, while at the same time enabling a more direct visualization of design intent. The language was executable and expressive enough to support creation of complete production-quality implementations.

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part II, LNCS 9953, pp. 11–26, 2016.
DOI: 10.1007/978-3-319-47169-3_2

At the time, most telecom software development proceeded according to the conventional waterfall approach: once the requirements were defined and frozen, a high-level design was first produced and documented. This documentation was then used as a basis for further refinement, wherein the individual elements of the high-level design were detailed out and, when that was complete, the results were captured in a set of detailed design documents. At that point a design "freeze" would be declared and the implementation phase would commence, typically using a third-generation programming language such as C. One of the most vexing issues was that, due in great part to the semantic gap between the implementation language and the language of the design, it was easy for the implementation to diverge from the design. This corruption of design intent typically occurred gradually and silently, since it is often very difficult to detect these types of flaws by human inspection of program code.

The motivation that led to the introduction of our new domain-specific language was to avoid these shortcomings and to accelerate development. By using this *modeling* language, we envisaged that one simply had to start by specifying the high-level design and, through a process of continual incremental refinement, eventually conclude with a complete final implementation. Because all such refinement occurred in the context of the original high-level design while using the same specification language, the likelihood of undetected design corruption was significantly lowered. Moreover, since a highly intuitive diagrammatic form was used to specify the high-level (architectural) elements, we thought that the design, when supplemented with appropriate descriptive text, could be discerned directly from the implementation. And, as a final step, the corresponding implementation would be generated automatically by a computer-based code generator, which translated the final design specification (i.e., model) into an executable program.

However, as users accumulated experience with the language, this compellingly simple and highly appealing vision of software development proved to be naïve and incomplete. In this essay, we examine the reasons why this was the case and some steps that need to be taken to make such an approach practical.

2 On Models and Their Use in Engineering Practice

Models have been an integral and critical element of general engineering practice since time immemorial. And, although *software* engineering is undoubtedly unique in many ways, it is, nonetheless, a form of engineering[1]. Consequently, before we focus on the use of models in software development, it is helpful to review the role of models in traditional engineering disciplines and what benefits (and pitfalls) this brings.

Reaching deep into the history of engineering, we encounter the instructive case noted by Marcus Vitruvius Polio. He was an "architect" in Ancient Rome at the time of Emperor Augustus (circa the first century BC). Vitruvius' legacy lives on through his

[1] The American Heritage Dictionary® of the English Language (5th edition) defines engineering as "the application of scientific and mathematical principles to practical ends such as the design, manufacture, and operation of efficient and economical structures, machines, processes, and systems." (http://www.thefreedictionary.com/engineering).

book, *De Architectura*, which is probably the oldest surviving engineering text. In Book X of this opus, he discusses a case of the use of models in engineering practice and concludes:

> *"For not all things are practicable on identical principles, but there are some things which, when enlarged in imitation of small models, are effective, others cannot have models but are constructed independently of them, while there are some which appear feasible in models, but when they have begun to increase in size are impracticable…"*[2]

The specific case behind these conclusions was the failure of what originally seemed to be a very clever siege defense mechanism: a rotating crane mounted on a city's wall that would grab an enemy's siege device as it was approaching the wall and then bring it "within the wall(s)" to be dealt with. This was based on a convincing demonstration of the effectiveness of such a device using a small scale model. Unfortunately for the defenders, in this particular case the enemy responded by constructing a siege engine of such exceptional proportions that it was simply not feasible to construct a crane of appropriate size that could deal with it.

What we can glean from Vitruvius' here is that: (a) models were used as "proof of concept" devices to stakeholders even in Antiquity (and probably earlier), (b) models sometimes served as "blueprints" to guide construction of the actual artifacts, and (c) that we have to be careful with predictions made using models.

2.1 What Is a Model?

It is helpful to define what we mean here by the term *model* in the engineering sense:

> ***Definition:*** *An engineering model is a selective representation of some system intended to capture accurately and concisely all of its essential properties of interest for a given set of concerns.*

A key component in this definition is the view that a model should be constructed *for a particular purpose* (i.e., for a "set of concerns"). A useful model reduces the amount of information that needs to be absorbed by removing or hiding from view those properties and elements of the modeled system that are deemed inessential *for the purposes of that model*. We discuss below the different purposes that engineering models serve, but, invariably, these include some type of analysis or examination of the model, based on which predictions can be made. Engineers want to ask questions of their models (Will it do what it is supposed to do? Will it be strong enough? How much will it cost?). By reducing the amount of information to be considered, answering such questions can be made easier, whether the analysis is performed by humans or computers.

When considering engineering models, we must not underestimate the human side of modeling, because the model itself as well as the results of model analyses typically have to be viewed and understood by humans. For example, stakeholders of a systems

[2] Taken from clause 5, Chap. 16, Book X, in [7].

must be able to unambiguously specify their concerns and requirements. Since it is often the case that they may not have the requisite technical background to understand technical specifications, human-centric ones are needed. Even in situations where models and analysis results are being examined by engineering experts, there is still a crucial need to facilitate understanding. This and other critical concerns pertaining to engineering models are discussed in the following section.

2.2 The Essential Properties of Useful Engineering Models

We postulate here that the following are essential properties that a *useful* engineering model must possess:

1. An engineering model must have *clearly defined purpose*. This is because the purpose of a model determines which elements of the modeled system are to be included in the model and which ones are to be excluded. A model that is intended to serve too many different kinds of analyses will result in an excess of information, hindering practical analyses. This naturally leads to the conclusion that we will likely multiple models for any given system.
2. Engineering models must be *abstract*. As discussed previously, a good model should contain only information about the modeled system and its immediate environment that is relevant to the purposes of the model.
3. Engineering models must be *accurate*. Obviously, the information contained in the model must be sufficiently faithful in its representation of the modeled elements such that the results of the analyses performed on the model can be trustworthy.
4. Engineering models must be *analyzable*. That is, the model must be constructed in such a manner that it is conducive to the desired analyses to be performed on it. These analyses are typically used to predict the properties of interest of the modeled system.
5. Engineering models must be *understandable* to their human stakeholders, because practically all design involves human judgement. Hence, models must be presented in a form that matches the worldview and intuition of its target audience. Models that are cryptic and difficult to decipher may obscure crucial flaws and misunderstandings. For models specified using computer-based modeling languages, this usually implies using a domain-specific syntax. After all, the concrete syntax of a language is the most immediate interface between human readers and underlying computer representations. Reducing the gap between these two reduces what Fred Brooks refers to as “accidental” (i.e., needless) complexity [1].
6. Last but certainly not least, an engineering model must be *cost effective* to construct. Clearly, it must be substantially cheaper and more efficient to construct a model than it is to construct the modeled system.

2.3 How Engineering Models Are Used

They following summarizes the various purposes behind engineering models:

1. Models are needed to assist in the *understanding* of both complex problems and corresponding systems. Determining what a system is doing (or, what it is supposed to do) and how it does it, is a major hurdle even when dealing with moderately complex systems. By reducing the amount of information presented, a model reduces the degree of complexity that needs to be comprehended.
2. Engineering models are also used to make *predictions* about the system under consideration. These predictions can be generated in many different ways; by using mathematical models, logical inference, search-based methods, or even informal reasoning and intuition. Predictions derived from models are used in two distinct cases: (a) to determine whether or not a proposed design choice will lead to a desirable solution, or, (b) to validate that a proposed model corresponds to reality (i.e., the prediction matches a known data point of the modeled system).
3. In addition, engineering models are often used to *communicate* knowledge and intent between human stakeholders. The design of complex systems typically involves different categories of stakeholders with different and sometimes conflicting sets of concerns, who ultimately have to reach consensus. For example, an architect may present a small scale model of a proposed design for a building, which can then serve as a basis for discussion of possible changes and alternatives. Once again, this means that it is critical that the model can be in a form that is understood by stakeholders.
4. Models can also serve as "blueprints" used to guide *implementation*. In this case, the model captures the design intent, which is to be realized by the implementers. In most engineering disciplines, design and implementation are distinct: they typically require different sets of expertise, different tools, and different processes. Consequently, transferring and conserving design intent during implementation can be a difficult and error-prone process. To minimize the likelihood of corruption of design intent, implementation-oriented models need to be sufficiently detailed and precise to be properly interpreted by the implementers.

2.4 The Two Categories of Engineering Models

There is an important difference between models used for implementation (item 4 above) and models used for the other purposes listed. Namely, whereas the first three purposes require *abstract* models that are designed to *minimize* the amount of irrelevant information, models used as blueprints should be sufficiently *complete and detailed* to ensure correct realization of the specified design intent. In this sense, these two types of models are in opposition to each other, although they serve complementary functions. We refer to the models used for understanding, prediction, and communication as *descriptive* models, since their ultimate purpose is to facilitate human comprehension. In contrast, models that are used for implementation are referred to here as *prescriptive* models. This is a very useful and important distinction, since it can serve as a guide

when constructing models, helping us, among other things, to avoid mixing of multiple contradictory purposes within a single model.

To illustrate the contrast between the two kinds of models, consider the case of a modern submarine as representative of the type of complex engineering system being built nowadays. Figures 1 and 2 show two different models of this system. The model in Fig. 1 is a descriptive model. Its primary purpose is to explain the basic principles of submarine operation. To that end, the model includes only the information essential to that purpose. (Note, however, that, despite the need to reduce the amount information presented, the model uses silhouettes of the submarine cross-section and of the waterline to help us understand the model more easily – this is the human aspect of modeling mentioned earlier.)

In contrast, the model in Fig. 2 is rich with detail, since it is intended to be used in the construction of the actual system. Clearly, models of this type tend to be much more difficult to understand and, hence, are not well suited to analytical reasoning or formal treatment.

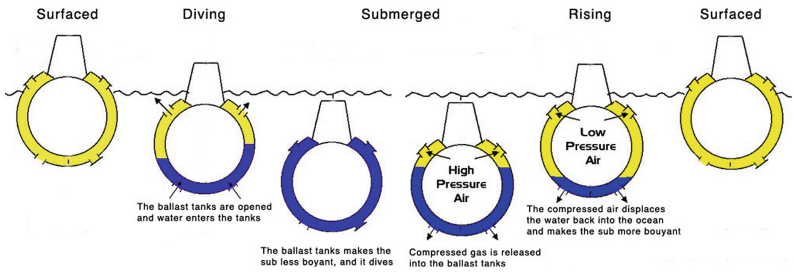


Fig. 1. A descriptive model of a submarine

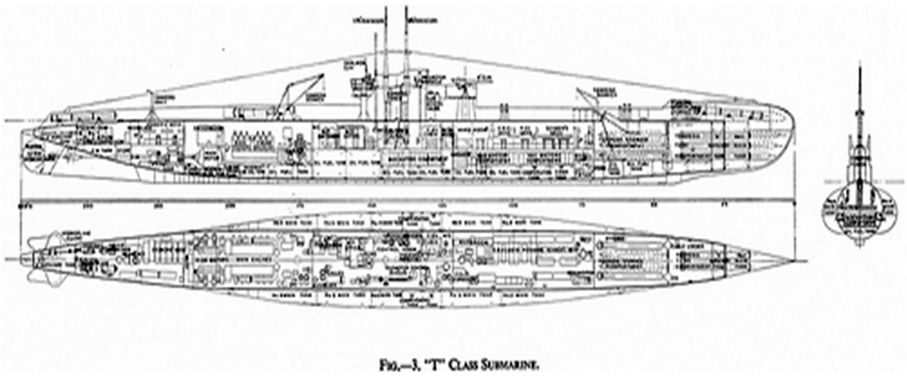


Fig. 2. A prescriptive model of a submarine

3 On Complex Engineering Systems and Their Models

It is interesting to contrast the highly intricate model shown in Fig. 2 with the far simpler one in Fig. 1. Why does a system that is conceptually so simple result in such a complex implementation? Where does all that additional complexity come from and is it really necessary? To answer these questions, we must examine what distinguishes a professional real-world engineering system from, say, a prototype or one built by non-experts.

3.1 On the Complexity of Engineering Systems and Its Sources

Modern engineering systems are becoming increasingly sophisticated and, consequently, more complex. For our discussion, it is helpful to single out the notion of the *primary functionality* of a system. This is the behavior of a system that is typically captured through use cases, and represents its *raison d'etre*. In case of submarine, for example, this is simply the ability to dive under water and re-surface as required.

Needless to say, if a system must support numerous different use cases, the primary functionality of an engineering system can be a source of great complexity. However, in practice it is rarely the only one. The other main source of complexity stems from the fact that industrial-grade engineering systems are generally required to be *dependable* and *practical*. By "dependable" we simply mean that a system is able to perform its primary functionality correctly when required. On the other hand, a system is deemed "practical" if operating it does not involve unwarranted or unreasonable effort (a form of accidental complexity) or excessive cost. For instance, a submarine whose internal cabin temperature exceeds 50°C would not be considered practical, no matter how well it performs its primary functionality.

Ensuring that a complex engineering system is both dependable and practical is usually achieved through additional ancillary mechanisms that supplement and support its primary functionality (e.g., air conditioning in submarines, power steering in automobiles, safety brakes in elevators). They exist solely to support the primary functionality and do not have a meaningful purpose outside the context of their system. In this essay, we shall refer to the set of such mechanisms as the *infrastructure* of a system.

Experienced professional engineers are fully aware of the importance and impact of a well-designed infrastructure. This is especially true given that, in many complex engineering systems, it is the case that *the complexity of the infrastructure exceeds by far the complexity of a system's primary functionality* (e.g., consider the models in Figs. 1 and 2)[3].

[3] Sadly, this is something that is still not well understood by many software engineers, where it is common practice to focus exclusively on use cases before any other considerations. The result is often a cumbersome and ineffective infrastructure.

3.2 Infrastructure in Complex Software Systems

But, is it meaningful to talk about the infrastructure in software systems? Surely, this is all handled by the underlying hardware, leaving software designers to worry only about implementing the primary functionality. This view has even led some software practitioners to conclude that software development is a discipline that transcends engineering[4]. Thus, one of the pioneers of computer science, Edsger Dijkstra, put it: "I see no meaningful difference between programming methodology and mathematical methodology"[5]. By this, he meant that software should be developed by a process that consists of designing computational algorithms and then proving their correctness using formal mathematical arguments.

While this approach may hold for certain limited categories of software applications, it definitely does not stand up in case of software that interacts directly with the physical world, such as the embedded software found in various *cyber-physical systems* [6]. Because these systems are typically required to interact continually with their environment, physical phenomena can have a major impact on their design, resulting in a significant amount of infrastructure to ensure dependability (e.g., fault-tolerance mechanisms, security mechanisms, performance-enhancing mechanisms (e.g., memory caches, pre-fetching mechanism)) and practicality (e.g., user interfaces).

However, there is typically much more to the infrastructure of software applications than just the mechanisms for coping with physical phenomena. Underlying most of today's software is at least an operating system and, possibly, one or more application frameworks. These provide much of the needed infrastructure in support of dependability and practicality. They are typically very sophisticated software programs whose complexity often exceeds that of most applications that run on top of them. Although from an application point of view an operating system hides behind its application programming interface (API) – an abstraction interface – its dependability and practicality characteristics must be accounted for in application design. For example, when considering the response time of a software application, the performance overheads of the underlying operating system can play a significant part.

Even more relevant is the fact that not all infrastructure functionality can be relegated to the operating system or application framework. As explained in the "end-to-end" argument by Saltzer et al., given that an operating system is designed to be generic (i.e., application independent), it cannot take on infrastructure functions that are specific to a particular application [11]. For example, the handling of a particular component failure may require non-standard application-specific recovery procedures. Therefore, such functions must be included directly in the application code.

In summary, we conclude that software systems also need an infrastructure, and that this can contribute in a major way to the overall system complexity.

---

[4] "Because [programs] are put together in the context of a set of information requirements, *they observe no natural limits* other than those imposed by those requirements. Unlike the world of engineering, *there are no immutable laws to violate",* Wei-Lung Wang in a letter to the editor published in the Communications of the ACM (vol. 45, 5), 2002.

[5] In EWD1209 (http://www.cs.utexas.edu/~EWD/).

3.3 Modeling Software Infrastructure

Because the infrastructure plays an ancillary role in an engineering system and does not directly contribute to the primary functionality of a system (although it can influence it), it is often omitted from or merely implied in many descriptive models. For example, when specifying the functionality of some software application, we rarely go into the complex details of how the operating system performs its job.

Consider, for example, the simple model of a software application depicted in Fig. 3, which shows two components, **PeerA** and **PeerB,** exchanging a simple high-level message ("hello"). However, what actually happens in this system involves a number of infrastructure (operating system) elements that are not visible in the application model. A typical "complete" scenario of such a situation might proceed as shown in Fig. 4. Here, we can see that the roles of components **PeerA** and **PeerB** are actually realized by two operating system threads, which use a set of ancillary operating system entities to transfer the high-level message from one end to the other using a reliable positive acknowledgement communications protocol.

Fig. 3. A descriptive model of a software application

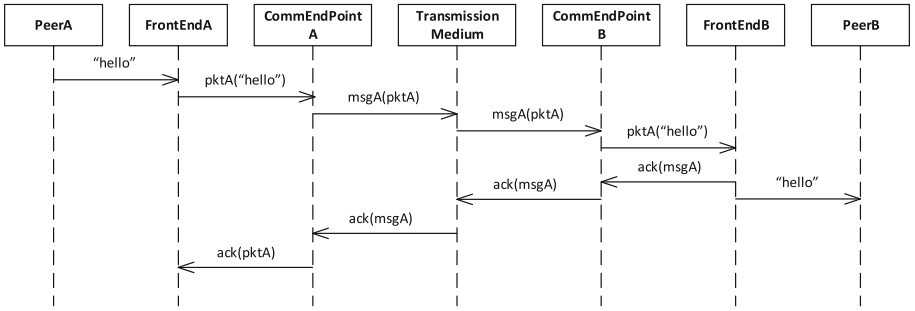


Fig. 4. A prescriptive model of the system shown in Fig. 3

The standard approach for dealing with this type of situation is to model the software system as being structured in a set of hierarchically arranged layers and then representing what happens within each layer separately, as shown in Fig. 5. Thus, the model in Fig. 3 only shows the content of the application layer, while completely hiding the presence and operation of the underlying operating system layer.

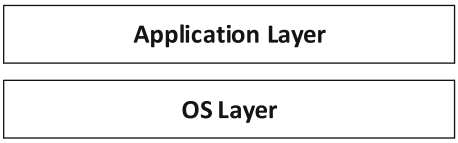


Fig. 5. A layered (descriptive) model of a software system

However, the use of layering is merely a modeling pattern used in descriptive models [13]. It does not in any way reduce the overall complexity of the underlying system, but merely helps us in understanding the system. (In fact, although commonly used in practice, software layers are a purely conceptual construct (i.e., an abstraction) that is not supported as a first-class concept in any standard programming language.)

3.4 Prescriptive and Descriptive Models in Industrial Practice

In the process of design, designers invariably start with high-level descriptive models. These capture putative architectures designed to satisfy the main system requirements. In fact, it is often the case that these models are often used to elicit requirements, since they serve as a convenient focal points for resolving potential stakeholder conflicts [8]. In order to make informed design decisions, it should be possible to analyze these models to determine whether or not they satisfy the requirements. For this it is critical that such models are analyzable for the properties of interest. Note that there are typically many different descriptive models serving different purposes, which have to be reconciled eventually – often a non-trivial task[6].

Through a process of gradual refinement of such models and analyses, one or more prescriptive models will emerge. Since the "devil is in the details" sometimes, inconsistencies and conflicts between different design proposals are only detected in such fine-grained models. Naturally, the later such issues are uncovered the greater the task of resolving them, since this might require rework of previously agreed designs.

But, beyond analysis and design, models also serve a fundamental purpose during system maintenance and evolution. To ensure preservation of the architectural integrity of a system, the designers of the new functionality must be sufficiently knowledgeable about the core design principles on which it was based. However, if the designers and developers of the new feature are new to the system, this can be difficult to achieve, particularly if (a) the original design team members are unavailable for consultation, or (b) if there is no trustworthy documentation that can be referenced. In such situations, descriptive models are crucial as teaching aids.

[6] The difficulty with this approach is that the independently derived solutions to the sub-problems may not be independent of each other. This leads to subsequent integration problems. As Michael Jackson noted: "Having divided to conquer, we must reunite if we wish to rule" [5].

4 On Model-Based Software Engineering

There can be no doubt that software is unique among engineering disciplines in a number of regards. The most obvious is the fact that software involves minimal production costs. When implementing software there is no heavy material to be lifted, carried, or bent into shape; no chemicals to be obtained, combined, or processed in some complex manner; no expensive scaffolding to put up. This is, of course, an important benefit, but it can also have disadvantages. Chief among these is that, unhampered by physical production constraints, it is easy to generate complexity in software merely by writing code.

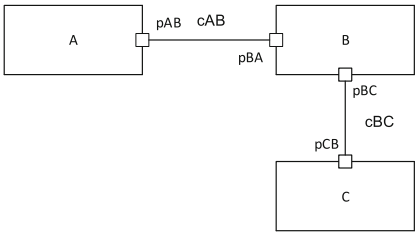
A proven method of reducing complexity, is to use of higher-level computer languages, which are closer to human reasoning and to domain-specific concepts. This was the original motivation behind so-called third-generation programming languages. However, these have not proven effective for descriptive purposes resulting in the emergence of modern *modeling languages*.

4.1 Modeling Languages vs. Programming Languages

There is some debate whether there is a fundamental difference between modeling and programming. After all, a program is a human-readable textual *representation* of the binary data that is actually stored and executed in a computer. Thus, it can be argued that by programming, we are abstracting away the details (i.e., modeling) of the underlying computing instruction set and data representations. Given that, it can be argued that programs *are* models. So, is this conceptually any different from the case where a model written in some computer-based modeling language is used to generate code?

The simple answer to that question is "no"; i.e., programming *is* indeed a kind of modeling. However, that question may be too narrowly focused, since it fails to account for the full range of purposes of models. The fact is that programming languages are intended primarily and almost exclusively for prescriptive purposes, which means that they tend to be more technology facing than human facing. As a result, their constructs and their syntax are designed to be sufficiently precise and detailed to ensure an unambiguous specification of the desired implementation. Note that practically all common programming languages use a strictly textual syntax, which is generally much easier to process by a computer than a graphical syntax. This despite the proven fact that some aspects of a system may be much more naturally expressed using graphical forms.

For example, compare the two representations of a component-based network structure shown in Fig. 6. The diagram on the left uses a typical graphical notation, such as found in a modeling language like UML. The right-hand side shows an equivalent textual specification of the same network as might be expressed in some programming language. Most human readers would agree that the mixed graphical-textual representation of on the left is more intuitive and, therefore, easier to understand.




```
network ABC {
        part A {port pAB};
        part B {port pBA, pBC};
        part C {port pCB};
        connector cAB {A::pAB, B::pBA};
        connector cBC {B::pBC, C::pCB};
        }
```

Fig. 6. A component diagram (left) and its textual representation (right)

Thus, one key distinction between modeling languages and programming languages is that the former have a concrete syntax that is much more oriented to human needs. Furthermore, because the definitions of modeling languages often separate their abstract syntax from their concrete syntax, it is even possible to use multiple different concrete syntaxes for a given modeling language, based on the purpose of a model. This is particularly useful, since it allows us to view a given model using different concrete representations, depending on concerns.

There is another important distinction that can differentiate the two categories of languages: the degree of enforcement of formal syntactical rules. In case of programming languages, there is little flexibility: a compiler will not proceed with code generation until every last syntactical flaw has been removed. That is, before a program can be useful, it must be both complete and syntactically correct.

Most descriptive models, on the other hand, tend to be incomplete and may even be left inconsistent. For example, we may be using a modeling language just to "sketch out" a vague idea in the form of a model, so that it can be discussed by stakeholders. In such situations, we would definitely prefer not to be burdened in placing unnecessary effort in ensuring full conformance to the various syntactical rules, since we are not interested in using such a model for prescription. Ideally, exploration of the design space should be made as lightweight and as efficient as possible, especially in the early phases of development – something that is difficult to achieve if a programming language is used for this purpose because of the need to make even early prototypes complete and correct at all levels of detail. Hence, we are left with less time for exploring different design alternatives.

One alternative, of course, is to use completely informal specifications in natural language text, pseudocode, or informal diagrams to capture a design idea during design space exploration. This, unfortunately, relies exclusively on all-too-fallible human reasoning and is less reliable. Clearly, the optimum seems to lie somewhere in the middle; that is, something that allows us to take advantage of the power of computers to help with correctness, yet does not tie us down too much with a bureaucratic-like formality, at least not until necessary.

There are two ways of achieving this option. One possibility is simply to define fewer syntactic rules in the language, such that it is possible to define models that are incomplete, yet can still be partially checked for syntactic consistency. An example of such a language is standard UML, which has numerous formal syntactic rules

expressed as OCL constraints that can be validated, but still leaves the possibility of incomplete models [9]. Some UML tools provide a refinement of this approach, by allowing modelers to define which of the OCL rules are to be enforced and which ones not. The other strategy is similar: it consists of defining a complete set of formal syntactic rules that a fully valid model must obey, but to group these rules into different levels of strictness. Modelers can then select the degree of strictness that they would like to enforce at a given point. During early phases of development, the level would be set low and increased gradually over time as the design solidifies.

4.2 From Models to Code: A Seamless Thread?

As discussed in the introduction, my colleagues and I had anticipated a "seamless" process, whereby both design and programming (i.e., implementation) would be done using a single high-level domain specific language. In fact, there are a number of successful examples of pragmatic feasibility of this approach in industrial practice (e.g., [2, 16]).

This does indeed represent a unification of modeling and implementation. But, this only applies to *prescriptive models*, which, as we argued, are only one type of model needed in the engineering process.

A key lesson that emerged from our experiences with using an implementation-oriented modeling language is that the resulting prescriptive models are *not* suitable for descriptive purposes. Not only do they contain too much detail, but they are also expressed in a language that is not suitable for the wide variety of different descriptive purposes. Different stakeholders are focused on different concerns and, hence, prefer languages that more directly capture and reflect those concerns. In our experience, implementation models, with their abundance of detail required to specify the infrastructure and primary system functionality, have proven almost as complex and difficult to understand as traditional program code. This means that, while the modeling language can simplify an implementer's task, it does not do much for other stakeholders.

However, if we introduce multiple models specified in a variety of languages, there is the obvious danger of inconsistencies between such models and the implementation. This renders descriptive models as untrustworthy, which greatly diminishes their value. A putative but not fully proven solution to this problem is described next.

4.3 Resolving the Multiple Models Dilemma

Multiple mutually inconsistent sources of information for a given system present a dilemma to the reader: which source is to be trusted, if any? Duplicated information quickly becomes unsynchronized despite the most meticulous procedural strictures designed to prevent that. The only pragmatic solution to this problem is to have *exactly one reference source of information* for an element of the system. This does not mean that such information only appears in one place, which would clearly be too restrictive. Instead, it means that all representations of that information except for the reference itself, whatever their context and concrete form, are formally (i.e., automatically)

derived from the reference source. One approach to achieve this is that the basic source of all information about a system is contained in the final implementation (prescriptive) model itself. If full automated model-to-code generation is used, then the equivalent computer program is a fully derived artifact. Any changes to the model will be accurately reflected in the code. This approach could also be applied in the reverse direction to produce the necessary descriptive models from either the code or the implementation model.

Clearly, this requires sophisticated automatable model-to-model transformations. Particularly challenging are abstraction transformations, that is, transformations that generate abstract representations of detailed models into less detailed ones suitable for descriptive purposes. This means not only that they perform abstraction, but also that they translate from one modeling language to another.

To perform the necessary abstraction transformation for this case requires a precise definition of the various element-to-element mappings. For example, the elements **PeerA**, **FrontEndA**, and **CommEndPointA** in Fig. 4, are all "merged" into element **PeerA** in Fig. 3. To reduce overhead and effort, these types of mappings can often be based on standard abstraction patterns such as those described in [12].

To assist in performing such transformations, one approach that has proven successful in practice is the use of *concern-specific annotations* attached to the implementation model. These are used as "hints" to the transformation engine when constructing the domain-specific model. In case of the UML modeling language a facility that is suitable for this purpose is the *profile mechanism* [9, 10]. A UML profile can be used to provide a domain-specific interpretation of selected elements of a model. Furthermore, this can be supplemented with domain-specific data needed for analysis. For example, when analyzing the timing properties of a proposed software design, it is possible to identify time-consuming elements of a system as well as the amount of time that they consume, by marking them with appropriate annotations defined in the industry-standard MARTE profile of UML [9, 14]. This information can be used by a model-to-model transform program to produce a corresponding timing model of the system expressed in a language suited to that purpose. And, because UML profiles can be dynamically applied to a model (and also "un-applied" subsequently) without affecting the underlying model in any way, it is possible to provide many different domain-specific interpretations for a single implementation model.

Although there are numerous practical *ad hoc* solutions to model transformations, it is still primarily a research topic. Fortunately, there is some excellent work on the theoretical foundations of model transformations that should eventually wind its way into practice and industrial-strength tools (e.g., [3, 15]).

However, one major drawback of this approach is that it is practical only once the implementation model is in place. That is, it cannot be used in the forward direction. For example, numerous high-level concern-specific models might be used during the design process, expressed using domain-specific modeling languages. If accepted, such models must be refined and converted to the modeling language of the implementation and inserted into the overall implementation model. The problem of ensuring that such a transformation is semantics (i.e., design intent) preserving has no general solution.

However, once the conversion has taken place, it may be possible to capture it formally, so that it can be used subsequently to automatically derive the required descriptive view.

5 Conclusions

The ability of modeling languages to be used in the development of complex software systems has been answered in the affirmative in industrial practice numerous times (e.g., [2, 16]). This has raised the prospects of a potentially "seamless" progress from models to code, characterized by a continuous process of refinement starting with high-level models and terminating with model-based implementations. This is an appealing innovation since it avoids some of the most critical error-prone discontinuities that have plagued engineering from time immemorial, since they often lead to failure to accurately reflect design intent in the final implementation.

In this essay, we examined the role of models and modeling in the development of software systems by first analyzing how these are treated in more traditional forms of engineering. Along the way, we also reviewed the role and kinds of models used in software engineering practice. Based on this and despite all the idiosyncrasies and unique features of software relative to more traditional engineering technologies, we are driven to a conclusion that there does not seem to be any compelling reason why software engineering should approach models and modeling any differently in this regard. The only substantive distinction that uniquely characterizes software seems to be that production costs are practically negligible. However, since production occurs once the design is completed, this does not seem to affect what needs to be modeled or how it is done.

Acknowledgement. The author would like to express his gratitude to Prof. Manfred Broy and Dr. Gerard Berry on their very helpful and constructive reviews of the original version of this text. All remaining flaws are solely the responsibility of the author.

References

1. Brooks, F.: The Mythical Man-Month. Addison-Wesley, Reading (1995)
2. Corcoran, D.: The good, the bad, and the ugly: experiences with model-driven development in large scale projects at Ericsson. In: Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA 2010) (2010)
3. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
4. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
5. Jackson, M.: CASE tools and development methods. In: Spurr, K., Layzell, P. (eds.) CASE on Trial, Chap. 8. John Wiley & Sons (1990)
6. Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems, A Cyber-Physical Systems Approach, 2nd edn. (2015). http://LeeSeshia.org. ISBN 978-1-312-42740-2

7. Morgan, M.H. (translator): Vitruvius: The Ten Books on Architecture. Dover Publications, Inc., New York (1914). (An on-line version of this volume can be found in the Project Gutenburg repository at http://www.gutenberg.org/files/20239/20239-h/29239-h.htm)
8. Nuseibeh, B.: Weaving together requirements and architectures. IEEE Comput. **34**(3), 115–117 (2001)
9. Object Management Group (OMG): UML Profile for MARTE™: Modeling and Analysis of Real-time Embedded Systems™, Version 1.1, OMG document no.: formal/2011-06-02 (2011). (http://www.omg.org/spec/MARTE/1.1/PDF)
10. Object Management Group (OMG): OMG Unified Modeling Language™ (OMG UML), Version 2.5, OMG document no.: formal/2015-03-01 (2015). (http://www.omg.org/spec/UML/2.5/PDF)
11. Saltzer, J., et al.: End-to-end arguments in system design. In: Proceedings of the Second International Conference on Distributed Computing Systems, pp. 509–512. IEEE Computer Society (1981)
12. Selic, B., Gullekson, G., Ward, P.: Real-time Object-Oriented Modeling. John Wiley & Sons, Hoboken (1994)
13. Selic, B.: A short catalogue of abstraction patterns for model-based software engineering. Int. J. Inf. **5**(1–2), 313–334 (2011)
14. Selic, B., Gerard, S.: Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-physical Systems. The MK/OMG Press (2013)
15. Syriani, E., Vangheluwe, H., LaShomb, B.: T-Core: a framework for custom-built model transformation engines. J. Softw. Syst. Model. **14**(3), 1215–1243 (2015)
16. Weigert, T., Weil, F.: Practical experience in using model-driven engineering to develop trustworthy systems. In: Proceedings of IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC 2006), pp. 208–217. IEEE Computer Society (2006)