Towards a Unified View of Modeling and Programming (ISoLA 2018 Track Introduction)

Manfred Broy[1], Klaus Havelund[2(✉)], Rahul Kumar[3], and Bernhard Steffen[4]

[1] Technische Universität München, Munich, Germany
[2] Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA
klaus.havelund@jpl.nasa.gov
[3] Microsoft Research, Redmond, USA
[4] TU Dortmund University, Dortmund, Germany

Abstract. The article provides an introduction to the track: *Towards a Unified View of Modeling and Programming*, organized by the authors of this paper as part of ISoLA 2018: the 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. A total of 19 researchers presented their views on the two questions: what are the commonalities between modeling and programming languages?, and should we strive towards a unified view of modeling and programming? The idea behind the track, which is a continuation of a similar track at ISoLA 2016, emerged as a result of experiences gathered in the three fields: formal methods, model-based software engineering, and programming languages, and from the observation that these technologies share a large common part, to the extent where one may ask, does the following equation hold:

$$modeling = programming$$

Keywords: Modeling · Programming · Domain-specific languages
Similarities · Differences · Unification

1 Introduction

Since the 1960s we have seen a tremendous amount of scientific and methodological work in the fields of program modeling and specification, as well as the creation of numerous programming languages. In spite of the very high value of this work, however, this effort has found its limitation by the fact that we do not have a sufficient integration of these languages, as well as of methods and tools that support the development engineer in applying the corresponding techniques

The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11244, pp. 3–21, 2018.
https://doi.org/10.1007/978-3-030-03418-4_1

and languages. A tighter integration between specification and verification logics, graphical modeling notations, and programming languages could have many benefits.

In a (possibly over) simplified view, as an attempt to impose some structure on this work, we will distinguish between three lines of work: formal methods, model-based software engineering, and programming. The first formal methods appeared in the 1970ties, and subsequently have included formalisms such as VDM [8,9,22], CIP [6], Z [59], Event-B [2], ASM [25], TLA+ [41], Alloy [34], and RAISE [24], as well as theorem proving systems such as Coq [5], Isabelle [49], and PVS [54]. These formalisms, usually referred to as specification languages, are based on mathematical concepts, such as functions, relations, set theory, logics etc. A specification typically consists of a signature, which is a collection of names and their types, and axioms over the signature, constraining the values that the names can denote. A specification as such denotes a set of models, each providing a binding of values to the names, satisfying the axioms. Such formal methods usually come equipped with proof systems, such that one can prove properties of the specifications, for example consistency of axioms, or that certain theorems are consequences of the axioms. A common characteristic of these formalisms is their representation as text, defined by context-free grammars, and their formalization in terms of semantics and/or logical proof systems. In parallel, we have seen several model checkers appearing, such as SPIN [30] and UPPAAL [61]. These usually prioritize automated and efficient verification algorithms over expressive specification languages. Exceptions are more recent model checkers for programming languages, including for example Java PathFinder (JPF) [29].

Starting in the 1980s, the model-based software engineering community developed graphical formalisms, most prominently represented by UML [53] and later SysML [52]. These formalisms, usually referred to as modeling languages, offer graphical notation for defining data structures as 'nodes and edge' diagrams, and behavioral descriptions by diagrams such as state machines and message sequence diagrams. These formalisms specifically address the ease of adoption and understanding amongst engineers. It is clear that these techniques have become more popular in industry than formal methods, in part likely due to their graphical and seemingly more light-weight nature. However, these formalisms are complex (the standard defining UML is much larger than the definition of any formal method or programming language), are incomplete (the UML standard for example has no expression-language, although OCL [1] is a recommended add-on), and they lack commonly agreed upon standardized semantics. This is not too surprising as UML has been designed on the basis of an intuitive informal understanding of the semantics of its individual parts and concepts, and not under the perspective of a potential formal semantics ideally covering the entire UML. This leaves users some freedom of interpretation, in particular concerning the conceptual interplay of individual model types, but often leads to misunderstandings. Nevertheless, it has been perceived to be sufficient in practice in order to support tool-based system development, such as, e.g., (partial) code generation.

Historically, programming languages have evolved over time, starting with numerical machine code, then assembly languages, and transitioning to higher-level languages with Cobol and Fortran in the late 1950s. Numerous programming languages have been developed since. The C programming language has since its creation in the early 1970s conquered the embedded software world in an impressive manner. Later efforts, however, have attempted to create even higher-level languages. These include languages such as Java and Python, in which collections such as sets, lists, and maps are built-in, either as constructs or as systems libraries. Especially the academic community has experimented with functional programming languages, such as ML [46], OCaml [50], and Haskell [37], and more recently with the integration of object-oriented programming and functional programming, as for example in Scala [55].

If we view each formalism in the above mentioned formalism classes as a set of abstract language constructs, it is likely that different formalisms will have elements (language constructs) that are not in common. Each formalism has advantageous features not owned by other formalisms. However, what is perhaps more important is that these formalisms for specification and modeling, from now on for simplicity referred to with the common term: modeling languages, and programming seem to have many language constructs in common, and to such an extent that one can ask the controversial two questions: *what are the commonalities between modeling and programming languages?*, and *should we strive towards a unified view of modeling and programming?* It is the goal of the track to discuss the relationship between modeling and programming, with the possible objective of achieving an agreement of what a unification of these concepts would mean at an abstract level, and what it would bring as benefits on the practical level. Note that this discussion is not meant to favor one view (that modeling = programming) over the other (that modeling $\neq$ programming). The track is a continuation of a first track on the same topic, held at ISoLA 2016 [15].

The paper is organized as follows. Section 2 presents arguments for the view that modeling fundamentally differs from programming. Section 3 presents arguments for the opposite view that modeling strongly overlaps with programming. Section 4 discusses the role of domain-specific languages. Section 5 provides an overview of the papers submitted to, and presented at, the track. Finally, Sect. 6 concludes the paper.

2 Differences Between Modeling and Programming

There is clearly a close relationship between formal modeling and programming. Every program can be seen as a formal model, and we can furthermore derive a number of limited perspective models (abstractions) from it, such as data flow descriptions, control flow models, and architecture models [13]. It can be argued, however, that there are a number of very elementary differences between modeling and programming. It is e.g. generally considered a good principle to separate the formalization of problems (what) and their solutions (how), as later expressed

in an implementation, analogous to e.g. what happens in the engineering field. There are indeed some arguments for this separation.

Programming is traditionally algorithm oriented, relying on an operational semantics of the programming language. This means that when programming one has to bring what one wants to express into such an operational form (this is of course to a lesser extent the case for logic programming languages such as Prolog). Modeling can involve looseness, in the form of non-determinism and underspecification. Programming languages usually only support non-determinism indirectly, through concurrency or calls of random-functions. Modeling languages often support some form of first-order (or higher-order) logic, permitting quantification over infinite sets, which of course is not possible in a programming language. Finally, when writing programs, in some cases one has to deal with particularities of the execution platform. A clear example is assembler programs.

Related to this observation is the fact that algorithmic languages need some concept of iteration or recursion, which has to be captured by a fixpoint theoretic semantics. For models we usually do not need fixpoint theory, in general, although there are exceptions. In programming, one cannot avoid to deal with issues of termination, and even worse, of nontermination. This marks the borderline between universal programming languages and pure modeling languages for which execution is not considered.

A particular aspect of the algorithmic focus is that of efficiency and computational complexity. These are usually purely algorithmic notions in relation to programs. When modeling, we can use constructs which are not executable, and even if they are, we might not care very much about the question. It is an accepted view point that one should usually not consider the efficiency of a model. We can only talk about the efficiency of an algorithm.

The essential idea behind programming languages is that they are traditionally meant for communication between humans and the machine. In contrast, most modeling languages are for the communication between humans for the clarification of ideas, to understand a problem and its solution. This is of course a truth with modifications. New programming languages attempt to make programs yet easier to write and read by humans, and some modeling languages focus on efficiency calculations.

In the programming world there are very few accepted programming paradigms. These include procedural programming, object-oriented programming, functional programming, and logic programming. In modeling there seems to be a much larger variety of paradigms. These include e.g. ontologies, class diagrams, state charts, activity diagrams, sequence diagrams, timed automata, model-based formal specification languages (where one uses collection types such as sets, lists, and maps to build other types), algebraic specification (using equations between terms for specifying semantics), differential equations, etc. The playing field seems much larger. An important distinction here is between discrete systems (e.g. state machines) and continuous systems (e.g. speed and acceleration), e.g. modeled with differential equations, as encountered in cyber physical systems.

An interesting observation is that in the model-based engineering community, where formalisms are mostly graphical, there is less emphasis on concrete syntax, and more emphasis on abstract syntax. However, since abstract syntax is often itself represented as diagrams, it becomes somewhat of a challenge to precisely define what the 'modeling language' is. Although we do see this as an issue, we also recognize that the focus on abstract syntax rather than concrete syntax, as is done in the programming language community, may have advantages.

3 Similarities Between Modeling and Programming

Programming languages are indeed meant for description of data and algorithms in a way that machines can execute. However, programming languages have evolved over six decades since the conception of Fortran in the mid 1950ties, and today's high-level programming languages provide language constructs that can be used for modeling and not just implementation. Let's take a simple example. When Algol 60 was defined as the first committee programming language, the members of the committee decided not to standardize input and output. At that time, input and output was considered as an unimportant technical detail. Today, however, many applications are interactive. Therefore, the flow of input and output between different distributed programs is of a completely different and of a much more important nature. What was considered as unimportant in Algol during its initial design, is important today. Support for interactive programming is today supported in most newer programming languages, e.g. through the notion of actors, and is important for modeling as well. Other evolving programming concepts include object-oriented programming, functional programming, and advanced type systems, and specifically the combination of these concepts.

This point can in particular be illustrated by the large similarity between the modern programming language Scala [55], first appearing in 2004, and the long standing tried and proved VDM specification language [8,9,22], developed three decades earlier in the mid 1970ties, and in particular its subsequent object-oriented version VDM^{++} [22]. There are in fact very few language constructs in VDM^{++}, which one will not find in Scala, largely concerned with infinite structures, namely existential and universal quantification over infinite sets (e.g. $\forall x : \mathbb{Z} \cdot P(x)$), and set comprehensions over infinite (e.g. $\{f(x) \mid x : \mathbb{Z} \cdot P(x)\}$). In our experience, however, practical applications of VDM existential/universal quantifications are usually over finite sets (e.g. $\forall x \in S \cdot P(x)$ for some finite set S), and similarly for set comprehension (e.g. $\{f(x) \mid x \in S \cdot P(x)\}$). Such finite quantifications and comprehensions over collections exist also in a language such as Scala (e.g. `S. forall (x ⇒ P(x))` and **for** `(x ← S)` **yield** `f(x)`). VDM also supports *design by contract*, meaning pre/post conditions on functions and class invariants. However, such concepts have found their way into programming languages, e.g. in Eiffel [19]. VDM finally supports predicate subtypes (e.g. **type** $\mathbb{N} = \{x : \mathbb{Z} \mid x \geq 0\}$). This kind of construct is now seen in programming languages supporting dependent types, such as Agda [3] and Idris [32].

If we consider UML/SysML, we can notice that an important part of UML/SysML is class diagrams, which essentially are class definitions with declarations of variables and methods specified with pre/post conditions, and occasionally code, plus constraints, typically written in OCL, which is a functional programming equivalent. These concepts can easily be represented in a programming language. Similar observations can be made about state charts, which fundamentally is a programming concept. It is not clear why we call the description of an algorithm by a state machine modeling and the description of the same algorithm by a program not necessarily modeling. Sequence diagrams are not directly representable as an executable programs. However, a sequence diagram can be considered as a property that a program execution has to obey. In that sense such a sequence diagram can be turned into a monitor of the executing system once built (a temporal assertion).

We have above argued that programming languages can handle finite data structures, and that these are useful and very common in modeling. However, so-called wide spectrum languages have been developed supporting a continuum, from models independent from any computational or algorithmic nature, to programs. In such systems one can establish and prove a refinement relation between a description at a higher level and a description at a lower level. We already mentioned VDM, which is an example of such a wide-spectrum language. Another example is the CIP-L language of the CIP system [6], where a full fledged programming language, comprising different programming styles such as functional as well as procedural programming, is integrated with non-executable constructs from set theory and predicate logic.

In summary, it seems worthwhile for the modeling community to benefit from the long chain of developments in programming languages, most of which have been tried and tested in the field. Not only past developments but also new developments, such as integrating programming, specifications, and proofs as is done in type theoretic languages such as Agda and Idris, and other systems such as Dafny [42] and Why3 [10]. Likewise, in the opposite direction, programming language design probably already have been and will be influenced by specification languages. Furthermore, it seems that program visualization techniques (of static structure as well as of executions) could help bringing modeling and programming closer together. Finally, extensible programming languages supporting the development of domain-specific language (DSL) constructs in addition to or restricting a programming language seems to be an important topic. The next section goes into more detail on the topic of DSLs.

4 Domain-Specific and Aspect-Oriented Languages

The perhaps major difference one could identify between programming and modeling languages is the level of abstraction: modeling languages explicitly support the focus on a specific aspect while ignoring others. Section 2 mentions computability, complexity and performance as examples. This difference essentially vanishes when looking at aspect-oriented [40] and domain-specific programming

[23,39]. In particular, aspect-oriented programming aims at a modular treatment of (so-called crosscutting) concerns, whereas domain-specific languages (DSLs) can be considered a means to generalize this form of modularity, both conceptually and technically:

- conceptually, one can consider a certain aspect as a particular domain, e.g., the domain of a specific kind of security, dependability, or traceability.
- technically, weaving can be considered as a very specific feature of a code generator that, e.g., merges a domain-specific/aspect program into code of the overall system.

In this sense, DSLs are much more than a way for supporting efficient programming by, e.g., factoring out boilerplate code. E.g., the *Language-Oriented Programming* [18,64] approach (LOP) as followed by the Racket team [21] is based on DSLs to support what they call the ultimate goal of programming language research, namely to deliver *software developers tools for formulating solutions in the languages of problem domains."* (cf. Fig. 1).

```
01 #lang video
02
03 (image "splash.png" #:length 100)
04
05 (fade-transition #:length 50)
06
07 (multitrack (blank #f)
08             (composite-transition 0 0 1/4 1/4)
09             slides
10             (composite-transition 1/4 0 3/4 1)
11             presentation
12             (composite-transition 0 1/4 1/4 3/4)
13             (image "logo.png" #:length (producer-length talk)))
14
15 ; where
16 (define slides
17   (clip "slides05.MTS" #:start 2900 #:end 80000))
18
19 (define presentation
20   (playlist (clip "vid01.mp4")
21             (clip "vid02.mp4")
22             #:start 3900 #:end 36850))
23
24 (fade-transition #:length 50)
25
26 (image "splash.png" #:length 100)
```

Fig. 1. A script in the Racket-based *Video language* (reprinted from [4]).

Clearly, the racket team addresses programmers, or even super-programmers, capable of mastering various (programming) languages. This requirement is a little bit relaxed in the projectional editing approach [62] as most prominently provided by JetBrains' Meta Programming Systems (MPS) [35], which allows one to integrate DSLs that are not purely textual, e.g., spreadsheets and tables.

Language-Driven Engineering (LDE) goes even further by considering DSLs as a new way to impose a new kind of modularity which enables the cooperative development even of non-programmers with different mindset and education [60]. These people can be enabled to participate in the development process using adequate DSLs perhaps designed as enrichments of well-known application-level modeling languages, like P&ID diagrams, timing diagrams, process models, electrical wiring diagrams, timed automata, Markov chains, or whatever such users wish to use to support full code generation. Figure 2 displays a few of the languages we used in our industrial projects.

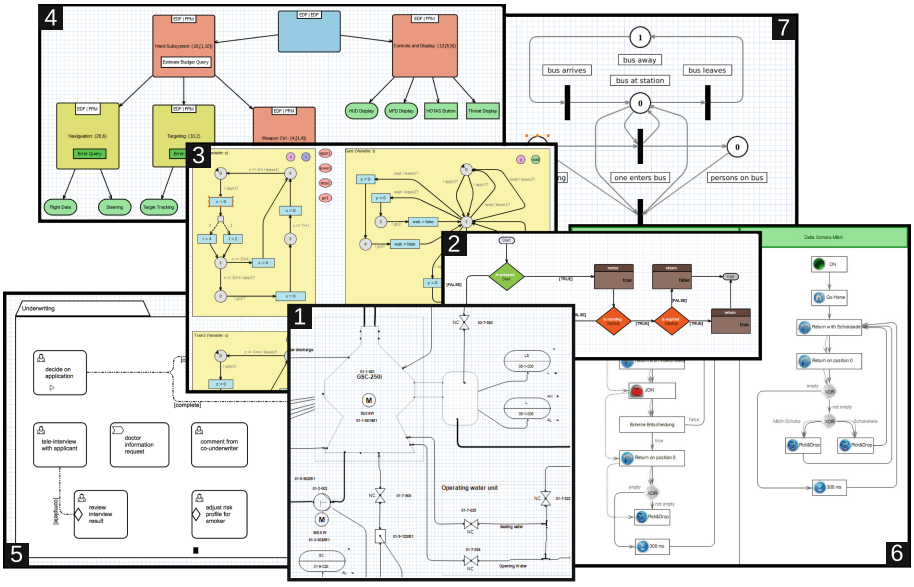

Fig. 2. Examples of DSLs: (1) Piping & Instrumentation Diagram [66], (2) Flow Graph [66], (3) Probabilistic Timed Automata [48], (4) Hierarchical Scheduling Systems [16], (5) OMG's Case Management CMMN [65], (6) EasyDelta Pick and Place DSL [7], and (7) Place/Transition Net [47] (reprinted from [60]).

While the LDE approach aims at enriching typically graphical domain languages[1], like the ones shown in Fig. 2, in order to define an external DSL for which full code can be generated, the LOP approach, as presented in [21], aims at capturing domain-specific features by establishing tailored internal domain-specific languages (there called *embedded DSLs* or *eDSLs*) on top of LISP/Racket

[1] Which are very popular in practice, as *"pictures are (often) worth a thousand words"*.

(see, e.g., Fig. 1)[2]. As a consequence, the addressed software developers are clearly programmers, while it is the goal of LDE to provide tailored (graphical) languages that allow application experts without programming knowledge to act themselves as software developers. In this sense, the Racket approach appears as a programming approach, LDE as a modeling approach, and the projectional editing approach as a hybrid. This illustrates the flexibility of DSLs to support the system development both at the modeling and the programming level. The work presented in [12] goes even at step further by considering DSLs as a means for transforming typical programming tasks into modeling activities which blurs the difference between modeling and programming.

5 Contributions to ISoLA 2018

The papers submitted to the UVMP track are introduced below, grouped into subsections according to the sessions of the track. Within each session the papers are ordered to provide a natural flow of presentations. Section 5.1 (*On Modeling and Programming*) provides an overview of the concepts of modeling and programming, and presents a wide spectrum of views of their relationship. Section 5.2 (*Formal Methods and Proofs*) focuses on the role of proofs, which establish the formal relationship between modeling and programming. Section 5.3 (*Modeling as Programming*) examines more closely the degree to which modeling can be considered as a programming activity. Section 5.4 (*The Application Perspective*) relates the discussion of modeling versus programming to real world phenomena. Section 5.5 (*Tailoring Languages*) discusses the role of domain-specific languages.

5.1 On Modeling and Programming

Jones [36] (*On Modeling and Programming*), argues that the term 'model' is used in several very different ways in computer science: analytic, in fields like physics to explain observed natural phenomena to reproduce results, experiments and insights; synthetic, as in computer science and engineering addressing constructed artifacts built to satisfy problem specifications; and in mechanization of established hand procedures. He argues that all three views are defensible and productive, but lead to very different ways of thinking. He focuses on modeling as used in the analytic and synthetic contexts. The paper introduces the concept of programming and different types of modeling but then concentrates very much on classical models related to formal systems and to programs. It treats issues of computability and complexity and discusses also the paradigms of computer science including the empirical, the mathematical, and the engineering paradigm. It concludes by saying that modeling has several meanings and purposes.

[2] The difference between internal and external DSLs can be sketched as follows: an internal DSL is added (e.g. via API functionality) to a host language, which is usually a general-purpose programming language, while an external DSL comes with its own syntax that is completely independent of already existing languages.

Elaasar [20] (*Definition of Modeling vs. Programming Languages*), explains how mainly graphical modeling languages and programming languages have originated in different communities, with different requirements. This has lead to differences in how modeling languages and programming languages are defined. This discussion is centered around the concepts of abstract syntax, concrete syntax, semantics, and software APIs, and the point is e.g. made, that the differences have lead to different tooling. A main observation is that while programming language developers usually focus on concrete syntax, modeling language developers focus on an abstract syntax, which may have numerous concrete syntaxes, such as a textual syntax and a graphical syntax. The points are illustrated with a case study, the definition of an ontology modeling language. It is finally argued, that modeling and programming languages seem to move towards a common point, with interesting perspectives what concerns e.g. common tooling.

Hallerstede, Larsen, and Fitzgerald [26] (*A Non-unified View of Modeling, Specification, and Programming*), argue that modeling and programming serve different purposes, and that care should be taken to distinguish them during development. They argue that a unified notation and method would become overly complex. Especially with many stakeholders it would be unrealistic to impose a unified set of methods and languages. The view is presented that executability is in tension with specification abstraction, and that using specification abstractions in programs makes them inefficient and limits their usefulness. Specific features mentioned, that are seen in specification languages but not explicitly in programming languages, include looseness (allowing many implementations) and quantification over infinite sets. It is mentioned, that formal methods tools with advantage can interact with traditional programs, e.g. a program can call a constraint solver.

Lethbridge and Algablan [43] (*Using Umple to Synergistically Process Features, Variants, UML Models and Classic Code*), describe a methodology for modeling variants such as product lines, and features, using the same master syntax as design models that are used for modeling classes, states, and composite structures. The extension to Umple is achieved by introducing mixset, that allows for creation of mixins composed from multiple locations in a textual codebase. Impressively, this approach allows for multiple programming languages to be embedded and generated from the design models. This work enables improved analysis, documentation generation, and reviewing/testing of models, design and code. It is particularly impressive that the work presented also allows for separation of concerns between various aspects of models to exist, while maintaining benefits of modeling, analysis, and code generation.

5.2 Formal Methods and Proofs

Börger [11] (*Why Programming Must Be Supported by Modeling and How*), argues that including abstract modeling concepts in terms of high-level programming language constructs into programming environments is not sufficient to bridge the numerous abstraction levels that software development typically passes on its way from requirements to code. Rather, an appropriate modeling

framework (a design and analysis method and a language) is required that allows one to successively refine *ground models* comprising the user-level requirements in order to bridge the gap between descriptions understandable by the main stakeholders to executable code realizing the expected behavior. This is concretized in the realm of Abstract State Machines.

Huisman [31] (*On Models and Code - A Unified Approach to Support Large-Scale Deductive Program Verification*), points out that despite substantial progress in the area of deductive program verification over the last years, it still remains a challenge to use deductive verification on large-scale industrial applications. The classical reasons for why this is the case are mentioned, including the size of applications, and the need for users to provide loop invariants. However, in addition to these issues, problems are mentioned such as the need to reason about missing components, and the need for other specification formalisms than traditional pre-postcondition-style specifications. The suggestion is an approach based on a provable refinement relation defined between different levels in a model/program. Amongst important research topics are mentioned code generation from higher level models, support for optimization refinement, derivation of models from code, and support for compositional modeling and programming.

Ionescu, Jansson, and Botta [33] (*Type Theory as a Framework for Modelling and Programming*), propose type theory as a suitable framework for both modeling and programming. They show that it meets most of the requirements put forward in [14]. First and foremost, type theory supports specifying program properties as types, and programming and proving that (functional) programs meet their types. Type theory is compared to ZFC set theory, which is recalled to be a problematic foundation for computer science. Examples mentioned of systems based on type theory include NuPRL, Coq, Agda, Idris, and Lean. Type theory is not only considered as a foundation for programming but for mathematics in general, and as such can be used for example to encode continuous mathematics, useful for modeling of cyber physical systems. It is emphasized that type theory is particularly well suited for meta-programming, including definition of embedded DSLs.

O'Connor, Chen, Susarla, Rizkallah, Klein, and Keller [51] (*Bringing Effortless Refinement of Data Layouts to COGENT*), states that the COGENT systems programming language has enabled modeling of certain aspects of operating systems very effectively, but the gap between the current implementations and modeling capabilities/approaches is vast. The work attempts to solve an extremely difficult, and relevant problem with modeling for operating systems by narrowing the gap between the C data structures that are used profusely in operating systems, and the algebraic data types of COGENT. The data description language presented enables the programmer and modeler to effectively model the system and then verify properties about the system. The work presented is not only combining the various aspects of modeling, programming, and most importantly, verification, but, it is also paving the way for potentially creating operating systems using a waterfall design methodology, which, has mostly been a holy grail for system engineers and designers.

5.3 Modeling as Programming

Cleaveland [17] (*Programming is Modeling*), argues that programming is, in particular in the domain of embedded systems, a modeling activity, as it typically happens at a quite high level of abstraction, far away from the physical level. This tendency is supported by development languages that increasingly provide domain-specific features further abstracting from the physical reality. On the other hand he argues that modeling is much more general than programming, here seen as merely addressing the operational behavior, emphasizing that he regards the 'is' in the title as clearly asymmetric. The paper closes with discussing the implications of viewing programs as models, programming languages as meta models, and abstraction as a way to enforce structure.

Sestoft [57] (*Programming Language Specification and Implementation*), is presenting two examples concerned with programming language specification and implementation, illustrating the differences and similarities between modeling and programming. The first example is that of spreadsheets, and the evaluation of cell formulas. An operational+axiomatic semantics is presented, and it is shown how the operational semantics can be programmed in F#. It is shown that non-determinism in the specification may reflect run-time non-determinism in the implementation as well as under-specification. A cost semantics (specification) of spreadsheets is then presented, which would be difficult to represent in F#. The second example is a semantics of Ada written in VDM in the 1980ties, which is shown to be representable in F#, thus making the point that what was considered a specification in VDM in 1980 now looks much like an implementation in a functional language.

Havelund and Joshi [28] (*Modeling in Scala*), present two examples in using the Scala programming language for modeling. The first example is a reformulation in Scala of a conceptual model of what a relational database is, first formalized four decades ago in the VDM specification language. The similarity between the two formalizations is used as an argument that a modern programming language today has the a large intersection with what considered a formal specification language then. The second example is a reformulation of a spacecraft controller, first formalized two decades ago in the Promela language of the SPIN model checker. The modeling illustrates the use of an internal DSL for hierarchical state machines, and a randomized scheduler written in 50 lines of code, that detects the same four errors detected by SPIN. The argument is made that a high-level programming language can be used for modeling, and that further integration of modeling and programming is desirable, with support for DSL development, visualization, and verification.

Madsen and Møller-Pedersen [44] (*This is Not a Model*), argue for merging modeling and programming within the same language, and mention the object-oriented (modeling and) programming languages SIMULA and Beta as examples of languages designed with this objective. It is pointed out that one of the original advantages of object-orientation, introduced with SIMULA, was that the same concepts and language mechanisms could be used for analysis, design, and programming. This is contrasted to mainstream modeling and

programming approaches where different languages are used for modeling and for programming. The paper defines a model as being the execution of a program, where the program itself is the model description. This is in contrast to traditional modeling languages such as UML, where the collection of diagrams is considered the model. It is advocated that more focus should be on tool support for viewing program executions, including visual techniques such as e.g. sequence diagrams.

5.4 The Application Perspective

Hatcliff, Larson, Belt, Robby, and Zhang [27] (*A Unified Approach for Modeling, Developing, and Assuring Critical Systems*), present an architecture-centric approach for development of embedded real-time systems, that emphasizes the use of a formally specified architecture as the 'scaffolding' through which different modeling and programming activities are organized. An open-source medical device, a Patient-Controlled Analgesic (PCA) infusion pump, is used as a concrete example. The distinction between 'models', 'specifications', and 'programs' is blurred. The approach is specifically based on the Architecture and Analysis Definition Language (AADL). Behaviors of components can be expressed and verified, either in the state machine notation BLESS, or programmed in conventional style using Slang, a dialect of the Scala programming language supported by verification. BLESS state machines are translated into Slang. Slang is translated into C and C^{++}.

Smyth, Schulz-Rosengarten, and Hanxleden [58] (*Towards Interactive Compilation Models*), describe the impact of considering compilation between hierarchies of implementation languages as a domain that deserves its dedicated domain-specific development environment: Modeling the entire development process itself on a meta-model level extends the possibilities of the model-based approach to guide the developer not only by supporting the refinement of tools for model creation, but also debugging, optimization, and prototyping of new compilations. The paper reports on experiences gathered while working on the model-based reference compiler of the KIELER SCCharts project which, in particular, illustrates the impact of considering meta modeling as part of the program development.

Margaria [45] (*From Computational Thinking to Constructive Design with Simple Models*) argues that the most important aspect of the educational revolution imposed by Computational Thinking is the "doing" part in the sense of creating a habit of designing the logic of any project or endeavor in terms of simple models. The advocated modeling-oriented teaching approach is based on years of experience with middle and high school students, beginner students in computer science, and with students of other disciplines. They all have been introduced successfully to CS or programming via constructing simple, yet executable models in the form of short courses, bootcamps, and semester-long courses in various locations and settings. Unlike coding, the model-oriented approach promises to be scalable, and adequate to provide the general public of professionals with the kind of familiarity with computational concepts that can be a game changer

for the societal diffusion of basic computing-related comprehension and design skills. This perspective identifies dissemination of Computational Thinking as a new criterion for separating programming from modeling.

5.5 Tailoring Languages

Selić [56] (*Design Languages: A Necessary New Generation of Computer Languages*) argues that with the increased demand for so-called 'smart' systems required to interact with the physical world in ever more complex ways, we are witnessing a corresponding growth in the complexity of their embedded software. The first part of this paper examines in detail the primary inadequacies of current mainstream programming technologies, which renders them unsuitable for addressing modern software applications. This is followed by a discussion of emerging trends in computer language development, which point to a new generation of programming languages, referred to herein as design languages.[3] The primary technical requirements for these new languages are explained. The paper tackles an important problem, namely that of the future development of programming languages in a world full of cyber-physical systems and distributed computer applications.

Karsai [38] (*From Modeling to Model-based Programming*), starts with contrasting the limitations of 'classical' model-based design, e.g., in the UML-style, with the strong support domain-specific modeling frameworks like Matlab/Simulink provide, in particular, to their non-IT users. Karsai then addresses the question why 'truly' domain-specific software development which enables application experts to participate in the development process is still far from being (widely) accepted. The two main reasons given are the typically enormous effort for developing domain-specific development environments and lack of corresponding educations. The author proposes to address the first problem by enhancing the corresponding tooling and the second by adapting the software engineering curricula. The paper focuses on concretizing the corresponding vision by reporting on first experiences and successes.

Voelter [63] (*Fusing Modeling and Programming into Language-Oriented Programming: Our Experiences with MPS*) argues that modeling and programming, considered from the model-driven perspective, where models are automatically transformed into the real system, cannot be categorically distinguished. However, the two have traditionally emphasized various aspects differently, making each suitable for different use cases. After introducing 10 criteria and weighting to what extent they apply to either direction (modeling or programming), language-oriented programming with JetBrains MPS is presented as a hybrid approach, whose projectional editing and language modularity features provide powerful means to building domain-specific modeling tools. The main body of the paper presents discussions and examples from various projects for how those 10 criteria are addressed in MPS. As MPS itself is largely bootstrapped

[3] Design languages are essentially DSLs, as discussed in Sect. 4.

(i.e., built with itself), the very same criteria also apply to the meta level, explaining the choice of acronym which stands for Meta *Programming* System.

Bosselmann, Naujokat, and Steffen [12] (*On the Difficulty of Drawing the Line*) discuss the relationship between modeling and programming as a continuously evolving entity. It is a general tendency that structures and categorizations considered obvious in the past often get blurred in the cause of deeper investigation. E.g., the separating line between control and data path, traditionally clearly defined, is today often profitably moved by changing the level of interpretation, and even the gender classification has recently moved from a binary to a continuous spectrum. Domain-Specific Languages (DSLs), assumed here to come with corresponding rich tooling, are considered as a driver for a similar tendency when it comes to distinguishing between descriptive and prescriptive models, between model and program, or even between developer and user. Conceptual underlying key is to view the system development as a decision process which increasingly constrains the range of possible system implementations, and DSLs as a means to freeze taken decisions on the way towards a concrete realization. This way naturally comprises programming and modeling aspects. In fact, considering all interactions that influence the behavior of the system as 'development' turns GUIs into DSLs and users into developers. The pragmatics of this approach is illustrated in the light of the development of the Equinocs system, Springer's new editorial service.

6 Conclusion

We provided an introduction to the ISoLA 2018 track: *Towards a Unified View of Modeling and Programming*, discussing the possible unification of modeling and programming, the arguments against it, the arguments for it, and the role of domain-specific languages versus general purpose languages. Finally, we provided a summary of the 19 contributions to the track. The arguments against a unification of modeling and programming focus on certain features that cannot be implemented, are hard to implement, or are usually not seen in programming languages, such as under-specification, non-determinism, quantification over infinite sets, or continuous mathematics as found in cyber physical systems. An important argument is, that many interest groups may have different views on what formalisms are useful, and that designing a 'silver bullet' will not work. The arguments for a unification center around the observation that high-level programming languages tend to get closer and closer to modeling languages due to their abstractions, and that support for domain-specific extensions of programming languages will address some of the concerns raised against a unification.

Whichever way one sees this question, one can probably agree that more unification is possible than what can be observed in current practice, as formalisms in the different communities – viewed at an abstract level – already share many language constructs. However, the question remains, whether a single unified approach or just a unification of concepts should be strived for. After all, there are many (potentially conflicting) concerns that need to be taken into account:

- Allow for high-level as well as low-level programming.
- State properties of programs, as predicates, or as refinement relations between levels of abstraction, supported by formal proofs and testing.
- Textual as well as graphical syntax for programs/models.
- Visualization of executions.
- Support for meta-programming and design of domain-specific languages.
- Harmonize tooling technologies used in the different communities.

Finding a good balance between all those aspects without overloading individual solutions clearly provides lots of challenges for future research.

References

1. Documents associated with Object Constraint Language (OCL), Version 2.4. http://www.omg.org/spec/OCL/2.4
2. Abrial, J.R.: Modeling in Event-B. Cambridge University Press, Cambridge (2010)
3. Agda. http://wiki.portal.chalmers.se/agda/pmwiki.php
4. Andersen, L., Chang, S., Felleisen, M.: Super 8 Languages for Making Movies (Functional Pearl). In: Proceedings of the ACM on Programming Languages 1(ICFP) (2017)
5. Barras, B., et al.: The Coq Proof Assistant Reference Manual: Version 6.1 (1997)
6. Bauer, F., Broy, M., Gnatz, R., Hesse, W., Krieg-Brückner, B.: Towards a wide spectrum language to support program specification and program development. In: Alber, K. (ed.) Programmiersprachen. Informatik - Fachberichte, vol. 12, pp. 73–85. Springer, Heidelberg (1978)
7. Berg, A., et al.: PG 582 - Industrial Programming by Example. Technical report, TU Dortmund (2015). http://hdl.handle.net/2003/34106
8. Bjørner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language. LNCS, vol. 61. Springer, Heidelberg (1978). https://doi.org/10.1007/3-540-08766-4
9. Bjørner, D., Jones, C.B.: Formal Specification and Software Development. Prentice Hall International (1982). ISBN 0-13-880733-7
10. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, pp. 53–64. Wrocław, Poland, August 2011
11. Boerger, E.: Why programming must be supported by modeling and how. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 89–110. Springer, Cham (2018)
12. Bosselmann, S., Naujokat, S., Steffen, B.: On the difficulty of drawing the line. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 340–356. Springer, Cham (2018)
13. Broy, M.: On architecture specification. In: Tjoa, A.M., Bellatreche, L., Biffl, S., van Leeuwen, J., Wiedermann, J. (eds.) SOFSEM 2018. LNCS, vol. 10706, pp. 19–39. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73117-9_2
14. Broy, M., Havelund, K., Kumar, R.: Towards a unified view of modeling and programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 238–257. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_17

15. Broy, M., Havelund, K., Kumar, R.: Towards a Unified View of Modeling and Programming (Track Summary). In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, part 2. LNCS, vol. 9953, pp. 3–10. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_1
16. Chadli, M., Kim, J.H., Larsen, K.G., Legay, A., Naujokat, S., Steffen, B., Traonouez, L.M.: High-level frameworks for the specification and verification of scheduling problems. Softw. Tools Technol. Transfer (2017)
17. Cleaveland, R.: Programming is modeling. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 150–161. Springer, Cham (2018)
18. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. JetBrains onBoard Online Magazine 1 (2004). http://www.onboard.jetbrains.com/is1/articles/04/10/lop/
19. Eiffel (2015). http://www.eiffel.com
20. Elaasar, M.: Definition of modeling vs. programming languages. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 35–51. Springer, Cham (2018)
21. Felleisen, M.: A programmable programming language. Commun. ACM **61**(3), 62–71 (2018)
22. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs For Object-oriented Systems. Springer, Santa Clara (2005). https://doi.org/10.1007/b138800
23. Fowler, M., Parsons, R.: Domain-Specific Languages. Addison-Wesley/ACM Press (2011). http://books.google.de/books?id=ri1muolw_YwC
24. George, C., et al.: The RAISE Specification Language. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead (1992)
25. Gurevich, Y., Rossman, B., Schulte, W.: Semantic Essence of AsmL. Theor. Comput. Sci. **343**(3), 370–412 (2005)
26. Hallerstede, S., Larsen, P.G., Fitzgerald, J.: A Non-unified view of modelling, specification and programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 52–68. Springer, Cham (2018)
27. Hatcliff, J., Larson, B.R., Belt, J., Robby, Zhang, Y.: A unified approach for modeling, developing, and assuring critical systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 225–245. Springer, Cham (2018)
28. Havelund, K., Joshi, R.: Modeling in Scala. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 184–205. Springer, Cham (2018)
29. Havelund, K., Visser, W.: Program model checking as a new trend. STTT **4**(1), 8–20 (2002)
30. Holzmann, G.: The SPIN Model Checker. Addison-Wesley, Boston (2004)
31. Huisman, M.: On models and code - a unified approach to support large-scale deductive program verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 111–118. Springer, Cham (2018)
32. Idris. https://www.idris-lang.org
33. Ionescu, C., Jansson, P., Botta, N.: Type theory as a framework for modelling and programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 119–133. Springer, Cham (2018)
34. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2012)
35. JetBrains: Meta Programming System. https://www.jetbrains.com/mps
36. Jones, N.D: On modeling and programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 22–34. Springer, Cham (2018)
37. Jones, S.L.P.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)

38. Karsai, G.: From modeling to model-based programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 295–308. Springer, Cham (2018)
39. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press, Hoboken (2008)
40. Kiczales, G., et al.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0053381
41. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Pearson Education Inc., London (2002)
42. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
43. Lethbridge, T.C., Algablan, A.: Using umple to synergistically process features, variants, UML models and classic code. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 69–88. Springer, Cham (2018)
44. Madsen, O.L., Møller-Pedersen, B.: This is not a model. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 206–224. Springer, Cham (2018)
45. Margaria, T.: From computational thinking to constructive design with simple models. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 261–278. Springer, Cham (2018)
46. Milner, R., Tofte, M., Harper, R. (eds.): The Definition of Standard ML. MIT Press (1997). ISBN 0-262-63181-4
47. Naujokat, S., Lybecait, M., Kopetzki, D., Steffen, B.: CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. Softw. Tools Technol. Transfer (2017)
48. Naujokat, S., Traonouez, L.-M., Isberner, M., Steffen, B., Legay, A.: Domain-specific code generator modeling: a case study for multi-faceted concurrent systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8802, pp. 481–498. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45234-9_33
49. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
50. OCaml. http://caml.inria.fr/ocaml/index.en.html
51. O'Connor, L., Chen, Z., Susarla, P., Rizkallah, C., Klein, G., Keller, G.: bringing effortless refinement of data layouts to COGENT. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 134–149. Springer, Cham (2018)
52. OMG: SysML. http://www.omg.org/spec/SysML/1.3
53. OMG: UML. http://www.omg.org/spec/UML/2.5
54. PVS. http://pvs.csl.sri.com
55. Scala. http://www.scala-lang.org
56. Selić, B.: Design languages: a necessary new generation of computer languages. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 279–294. Springer, Cham (2018)
57. Sestoft, P.: Programming language specification and implementation. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 162–183. Springer, Cham (2018)
58. Smyth, S., Schulz-Rosengarten, A., von Hanxleden, R.: Towards interactive compilation models. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 246–260. Springer, Cham (2018)

59. Spivey, J.M.: The Z Notation - a Reference Manual. International Series in Computer Science, 2nd edn. Prentice Hall, Hemel Hempstead (1992)
60. Steffen, B., Gossen, F., Naujokat, S., Margaria, T.: Language-driven engineering: from general-purpose to purpose-specific languages. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 546–564. Springer, Cham (2018)
61. UPPAAL. http://www.uppaal.org
62. Voelter, M., Siegmund, J., Berger, T., Kolb, B.: Towards user-friendly projectional editors. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 41–61. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_3
63. Voelter, M.: Fusing modeling and programming into language-oriented programming. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 309–339. Springer, Cham (2018)
64. Ward, M.P.: Language oriented programming. Softw. Concepts Tools **15**(4), 147–161 (1994)
65. Weckwerth, J.: Cinco Evaluation: CMMN-Modellierung und -Ausführung in der Praxis. Master's thesis, TU Dortmund (2016)
66. Wortmann, N., Michel, M., Naujokat, S.: A fully model-based approach to software development for industrial centrifuges. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 774–783. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_58