Towards a Unified View of Modeling and Programming

Manfred Broy[1], Klaus Havelund[2*], and Rahul Kumar[3]

[1] Technische Universität München, Germany
[2] Jet Propulsion Laboratory, California Inst. of Technology, USA
[3] Microsoft Research, USA

Abstract. In this paper we argue that there is a value in providing a unified view of modeling and programming. Models are meant to describe a system at a high level of abstraction for the purpose of human understanding and analysis. Programs, on the other hand, are meant for execution. However, programming languages are becoming increasingly higher-level, with convenient notation for concepts that in the past would only be reserved formal specification languages. This leads to the observation, that programming languages could be used for modeling, if only appropriate modifications were made to these languages. At the same time, model-based engineering formalisms such as UML and SysML are highly popular in engineering communities due to their graphical nature. However, these communities are, due to the complex nature of these formalisms, struggling to find grounds in textual formalisms with proper semantics. A unified view of modeling and programming may provide a common ground. The paper illustrates these points with selected examples comparing models and programs.

1 Introduction

Over the last several decades we have observed the development of a large collection of specification and modeling languages and associated methodologies, and tools. Their purpose is to support formulation of requirements and high-level designs before programming is initiated. Agile approaches advocate to avoid explicit modeling entirely and suggest to go directly to coding. Other approaches advocate avoiding manual coding in a programming language entirely and suggest instead the generation of code directly from the models. This way modeling languages replace programming languages. We can divide modeling languages into formal specification languages (formal methods), usually focusing on textual languages based on mathematical logic and set theory, and associated proof tools (theorem provers, model checkers, etc.), and on the other hand model-based engineering languages (UML, SysML, Modelica, Mathematica, ...), focusing more

[*] 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.

on visual descriptions, code generation, and simulation. Many of these languages have similarities with programming languages.

In parallel, and frankly seemingly independent, we have seen the development of numerous new programming languages. Few languages have had the success of C, which still today is the main programming language for embedded systems. The success is so outstanding that nearly no progress wrt. praxis has been made in this domain (embedded programming) since the 1970ties, although some richer languages appeared soon after C in this domain, such as C++, Ada, and Eiffel. These later languages for example all have module systems, which C does not. We have seen several high-level languages appear that target the softer side of software engineering (such as web-programming, user interfaces, scripting), including languages such as Java, JavaScript, Ruby, Python and Scala. More academic languages include Haskell and the ML family, including OCaml.

There is seemingly a strict difference between a modeling language and a programing language. For a programming language we always assume a notion of *executability* and *computability*. Programming languages are restricted to concepts that can be executed. Put differently, programming languages put emphasis on the *"how"*, the algorithms for solving problems. A specification and modeling language in principle should rather focus on the *"what"*. A mathematical way of phrasing this is that specifications should ideally be *predicates* on solutions (executions for example). Intuitively, one may also argue that there are modeling tasks which do not directly aim at programing, for instance if we model a business process independent of the question which parts should be carried out by machines. This is modeling, which seems far away from programming. It might be interesting to bring it into a form which is closer to programming if we want to simulate or automatically analyze such models. But here there seems to be a boundary. Programming means computability. Modeling can be more general. Finally, at a more technical level, in programming, at least when we work in general purpose programming languages, we have to deal with non termination and the concept of undefined [15]. In a number of modeling approaches such concepts like undefined are avoided. Here again there is an interesting challenge in a unifying view of modeling and programming. We would have to manage to introduce the concepts of undefined into modeling, representing nonterminating expressions in programming. Some attempts have been made in this direction though, for example 3-valued logic as found in VDM [31].

In spite of these perceived differences, the similarities between modeling languages and programming languages are obvious, which suggests a unifying view. For example, many logics support the notions of local variables with bounded scopes and syntactic expressions similar to programming languages. Many modeling languages even offer programming constructs, such as mutable variables, assignment statements, and looping (while) statements, and of course recursion. Furthermore, some of the modeling languages, such as UML, are deeply influenced by programming languages wrt. how models are structured. In particular, the idea of object-oriented modeling is taken from the concept of object-oriented programming. It is even considered one of the strong sides of object-orientation, that one can have a unified view of object-oriented specification, object-oriented design, and object-oriented programming. In summary, the concepts that are used in modeling and the concepts that are used in programming are so closely related that it is beneficial to attempt a unified view.

In this paper we attempt to argue for such a unified view of modeling and programming. This view can in the extreme be considered a call for a single universal formalism for modeling and programming any form of system. This is done by high-lighting some trends in modeling and programming, and by programming some example models in the Scala programming language, a high-level formalism suited for this purpose. However, we fully understand that such a unification faces many obstacles, some of which are non-technical. What we intend is to fuel an *effort* to at least consider merging efforts to the extent feasible. We believe that the model-based engineering community can learn from the formal methods and programming language communities, and vice versa. Note that even if a single formalism would appear, there will always be alternatives, just like there are multiple programming languages (evolution continues).

The paper is organized as follows. In Section 2 we give a brief overview of some of the trends in modeling and programming, that we consider important. In Section 3 we illustrate with examples how modeling can be perceived as programming. Finally Section 4 outlines brief discussion points to be reflected on when considering a unified approach, as well as a conclusion.

2 Trends in Modeling and Programming

In this section we briefly survey some trends in the fields of formal methods, model-based engineering, and programming, that we find worthwhile highlighting.

2.1 Formal Methods

Early work on formal methods include the work of John McCarthy (*Recursive Functions of Symbolic Expressions and Their Computation by Machines* [34] and *Towards a Mathematical Science of Computation* [35]), Robert Floyd (*Assigning Meanings to Programs* [19]), Edsger Dijkstra (*A Discipline of Programming* [16]), Tony Hoare (*An Axiomatic Basis for Computer Programming* [29]), and Dana Scott and Christopher Strachey (*Towards a Mathematical Semantics for Computer Languages* [36]), to mention a few. These ideas were theoretic in nature and deeply influential. They brought us the ideas of annotating programs with assertions, such as pre- and post-conditions, and invariants, correct by construction development (refinement), and giving semantics to programming languages.

These ideas were subsequently the basis for several, what we could call, second generation formal specification languages such as VDM [13,14,31], VDM^{++} [18], Z [37], B and Event-B [12], CIP [38], TLA [32], RAISE [22], and OBJ [21], to mention just a few. Each of these languages are full specification languages, most with rich type systems and detailed rules (grammars) for what constitutes a valid specification. These languages were ahead of their time wrt. language features in the sense that many of these features have found their way into modern programming languages of today. A particular example of this is collections (sets, lists, and maps).

The VDM language for example is a wide-spectrum specification language offering a combination of high-level specification constructs and low level programming like constructs. The methodology consists in part, as in CIP, of refining a high-level specification into a low-level program like specification in a stepwise manner. The language offers concepts such as the combination of imperative (procedural and later object-oriented in VDM^{++}) and functional programming; exceptions; algebraic data types and pattern matching; functions as values and lambda abstractions; built-in collection types such as sets, lists and maps, with mathematical notation for creating values of these types, such as for example set comprehension; design-by-contract through pre- and post conditions and invariants; predicate subtypes (so one for example can define natural numbers as a subset of the integers); and predicate logic including universal and existential quantification over any type as Boolean expressions. VDM and Z are so-called *model oriented* specification languages, meaning that a specification is an example model of the desired system. This means that such specifications are somewhat close to high-level programs. This is in contrast to so-called *property oriented* (algebraic) specification languages, such as OBJ, where a specification denotes a set of models[4].

A different branch of formal methods includes theorem proving and model checking. In theorem proving we have seen specification languages, which resemble functional programming languages, including for example Isabelle [10], PVS [11], and Coq [9]. In model checking, early work, such as Spin [30], focused on modeling notations. However, recent research has focused on software model checking, where the target of model checking is code, as for example seen in the Java PathFinder model checker (JPF) [27,28], and in Modex [30] (for C). JPF was created due to the observation that a powerful programming language might be a more convenient modeling language than the traditional model checker input languages. Numerous model checkers now target C.

As can be seen from the above discussion, formal specification languages have for a long time been flirting with programming language like notations, and vice versa. However, the two classes of languages have by tradition been considered as belonging to strictly separate categories. VDM for example was always, and still is, considered a specification language, albeit with code generation capabilities. It has never, in spite of the possibility, been named a programming language, which one may consider being as one of the reasons it is not more wide spread. Writing specifications in VDM and generating code in Java, for example, has not become popular. Programmers feel uncomfortable working with two languages (a specification language and a programming language) when the two languages

[4] This characteristic of the difference is somewhat simplified since a VDM specification in fact also can denote more than one model.

are too similar. This is an argument for merging the concepts into a specification, design and implementation language.

2.2 Model-based Engineering

Model-based engineering includes modeling frameworks that are usually visual/-graphical of nature. One of the main contributions in this field is UML [8] for software development, and its derivatives, such as SysML [7] for systems development. The graphical nature of the UML family of languages has caused it to become rather popular and wide-spread in engineering communities. Engineers are more willing to work with graphical notations, such as class diagrams and state machines, than they are working with sets, lists and maps and function definitions. It seems clearly more accepted than formal methods as described in the previous section.

One of the important notations in UML/SysML is class diagrams. Class diagrams are, just like E/R-diagrams, really a simple way of defining data, an alternative to working with sets, lists and maps as found in VDM and modern programming languages. For example, to state that a person can own zero or more cars one draws a box for Person, and a box for Car, and draws a line between them. It is an idea that quickly can be picked up by a systems engineer, quicker than learning to use programming language data structures. Another notation is that of state machines, a concept that interestingly enough has not found its way into programming languages, in spite of its usefulness in especially embedded programming. UML and SysML also support requirements (as special comments), a concept that usually is not embedded as a first citizen in programming.

The above observations are rather positive. However, UML and SysML are very complexly and weakly defined formalisms. For example, the (human unreadable) abstract syntax for UML (including OCL, in a different document) is 11605 lines of XML, whereas the typical (human readable) concrete syntax (grammar) for a programming language is between 500 and 2500 lines. The UML/SysML standards are long and complex documents. Furthermore, the connection between models and code is fragile, relying on the correctness of translators from for example UML state machines to code. Finally, a discussion about semantics (what do two boxes with a line in between mean?) can turn a project meeting into chaos.

2.3 Programming

Several new programming languages have emerged over the last decades, which include abstraction mechanisms known from the formal specification languages mentioned above. Such languages include SPARK Ada, Eiffel, Java, Python, Scala, Julia, Fortress, C#, Spec#, F#, D, RUST, Swift, Go, Dafny, and Agda. Some languages support design-by-contract with pre-post conditions, and in some cases with invariants. These languages include for example SPARK Ada, Eiffel, Spec#, Dafny, and to some limited extent Scala. Java supports contracts through JML, which, however, is not integrated with Java, but an add-on comment language (JML specifications are comments in a Java program). Most of the languages above support abstract collections such as sets, lists and maps. It is interesting to observe that SUNs Fortress language (which unfortunately was not finished due to lack of funding) supports a mathematical notation for collections very similar to VDM. The systems Dafny and Why3 are amongst the newest branches of work, interesting since these languages are developed specifically with verification in mind.

A trend on the rise is the combination of object-oriented and functional programming, as seen perhaps most prominently in Scala, but also in the earlier Python, and now in Java which got closures in version 1.8. Ocaml is a similar earlier attempt to integrate object-oriented and functional programming, although in a layered manner, and not integrated with the standard module system. As in many other aspects, Lisp was early out with this combination with the Common Lisp Object System (CLOS). Some interesting new directions of research include dependent types as found in Agda (to some extent related to predicate subtypes in VDM) and session types. Session types are temporal patterns that can be checked at compile time. They are much related to temporal logic as used within the formal methods community to express properties of concurrent programs. At the same time there are also attempts to move away from C, but without losing too much efficiency. Examples include the languages Go, D and RUST. However, as stated earlier, C has an impressive staying power, and none of such attempts have yet become main stream.

3 Modeling as Programming

In this section we shall attempt to explore the argument that modeling can be perceived as programming. We will do this through a small collection of examples, illustrating how what is normally considered as modeling can be perceived as programming. Models are encoded in the Scala programming language, which is sufficiently high-level to illustrate the point. We start with class diagrams, as found in UML and SysML, then move on to a classical formal specification language such as VDM^{++}, and finally discuss Domain-Specific Languages (DSLs).

3.1 Modeling of Class Diagrams

A commonly used part of UML and SysML is the class diagram. The class diagram is a visualization of data structures as nodes and edges. Nodes represent data elements and edges represent the relationships between data elements. To take an example, consider the class diagram in Figure 1 (the example is adopted from [6]). This diagram models libraries of books. In this diagram a box (node) denotes a type, a set of objects of that type. Hence for example the top node $\lfloor$Library$\rfloor$ (references to text, for example names, in models are enclosed in $\lfloor$...$\rfloor$) denotes the type of libraries: a set of library objects each representing a library. A library has a name, which is a string. Note that such data of primitive types (strings, integers, reals, Booleans, ...) are represented as so-called *attributes* and are declared inside the boxes instead of as edges, although in principle they can be perceived edges to boxes representing primitive types[5]. A library consists of (left arrow) a collection of books (zero or more represented by the *multiplicity* $0..*$), reachable from a library object via the field ⌊books⌋. In the other direction: a book is related to zero or one (0..1) libraries. Similarly, a library (right arrow) has associated a collection of members. Books and members have names. In addition each book has as attribute the number of books on shelf. Finally, a loan is a connection between a book and a member, and a library has associated a collection of (current) loans.

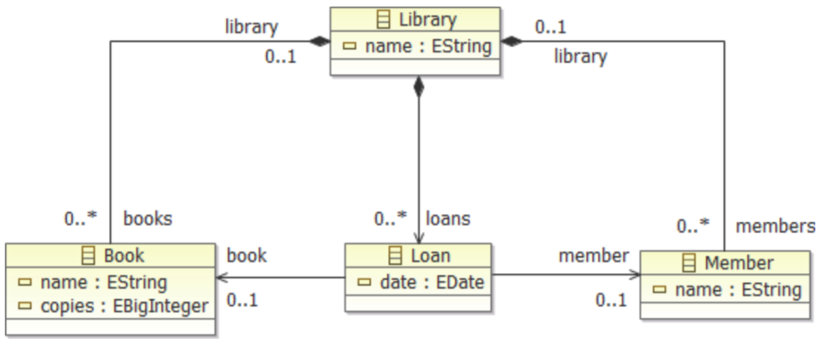

Fig. 1. The book library (from [6])

Book

$name : \mathrm{seq}\ CHAR$

$copies : \mathbb{Z}$

$copies > 0$

Fig. 2. Z model of books in a library

In many modeling situations such diagrams form the core of the modeling effort. Constraints can be added to such diagrams. For example one constraint could be that the number of copies of a book should be positive. Such a constraint (not shown) can be added inside a special constraint box on the diagram in Figure 1, attached to the ⌊Book⌋ box with a dotted line. It is interesting to

[5] This is an example of a discussion about semantics that can throw a project meeting off its course.

note, that a box with an associated constraint (written in another box and linked with a line) conceptually is very similar to the idea of a Z schema [37], as shown in Figure 2[6]. This schema represents the fundamental concept of a model: a signature (the declaration of ⌊name⌋ and ⌊copies⌋ above the line with their types) and then zero or more axioms (below the line). Attempts have been made to provide textual versions of UML and SysML diagrams. An example is the K specification language [26], that was developed at JPL. The expression language of K as well as Z (what is written in constraints) is predicate logic. Both languages support datatypes such as sets, including advanced set expressions such as set comprehension. K is object-oriented and is inspired by Z, as well as by other languages, such as VDM [13,14,31,18] and RAISE [22].

Another textual notation coming out of the model-based engineering community itself is OCLInEcore [5], which is an attempt to define a textual language combining the structure oriented Ecore meta-model of the Eclipse Modeling Framework (EMF) [2] with the OCL constraint language (Object Constraint Language) [1]. OCL is a declarative expression language that is now part of the UML standard. OCL descended from Z, but is based on chained method calls read from left to right, starting from *finite* collections, in contrast to predicate logic. For example OCL does not have general universal and existential quantification over infinite sets. In predicate logic we would write a universal quantification over a set/type S as follows: $\forall\, x : S \bullet P(x)$, meaning: for all x in the set S, $P(x)$ is true. In OCL one would write this as: ⌊S→ **forAll**(x | P(x))⌋. However, OCL requires S to be finite, in contrast to predicate logic, where S can be infinite. This is the major distinction between OCL and predicate logic, in addition to the alternative syntax. OCL is executable, given a model instance.

In order to illustrate OCLInEcore we expand our example by adding the following requirement: "*The number of loans that a book is part of should be less than or equal to the number of copies of the book*". The OCLInEcore model in Figure 3 formalizes this requirement. For this purpose, in addition to the two attributes ⌊name⌋ and ⌊copies⌋, two *properties* are defined. In contrast to an attribute, which has a primitive type, a property is linked to one or more objects of another user-defined type (those drawn as boxes in class diagrams). The property ⌊library⌋ links a book to the library it is part of, and is the "*opposite property*" of the ⌊books⌋ property of the ⌊Library⌋ (expressed using the ⌊#⌋-notation), meaning that if a book is in the ⌊books⌋ set (technically a bag) of a library, then the library is also in the ⌊library⌋ of the book. The '⌊?⌋' represents 0 or 1.

The property ⌊loans⌋ denotes a collection of ⌊Loan⌋ objects and is derived (meaning its value depends on other values), with the formula defining its value provided as an OCL expression. The expression reads as follows: from this book (referred to as ⌊**self**⌋ later in the expression), retrieve the library it is part of, retrieve the loans of this library, and select those for which the book is equal to ⌊**self**⌋. For a given collection ⌊S⌋, the notation ⌊S→ M(...)⌋ means calling the

[6] Note that the constraint can actually be avoided in Z by defining the type of ⌊copies⌋ to be $\mathbb{N}_1$, the natural numbers starting from 1.

```
class Book {
  attribute name : String;
  attribute copies : Integer;

  property library#books : Library[?];

  property loans : Loan[∗] { derived }
  {
    derivation: library.loans→ select(book=self);
  }

  operation isAvailable() : Boolean[?]
  {
    body: loans→ size() < copies;
  }

  invariant CopiesPositive:
    copies > 0;

  invariant SufficientCopies:
    loans→ size() ≤ copies;
}
```

Fig. 3. OCLInEcore model of books in a library (from [6], modified)

method ⌊M⌋ on the set ⌊S⌋. Hence in this case the ⌊**select**(predicate)⌋ method is defined on sets and returns the subset of elements of the set satisfying the predicate. The two invariants can now be formulated, and their explanation should at this point be straight forward. The *operation* ⌊isAvailable⌋ is defined to illustrate that one can also define such, here with an OCL expression as body. One can also define operations with side-effects specified with pre/post conditions. No code with side-effects, however, is allowed in bodies of operations, which seems to be a limitation, and a sign of an attempt to move towards a programming language, but not all the way.

The main point we are trying to make here is that the OCLInEcore model, which in reality is very similar to a Z specification (signature + axioms), can (for the most part) be elegantly expressed in the Scala programming language. This is shown in Figure 4. The class ⌊Book⌋ extends the class ⌊Model⌋, which we have programmed to offer various methods for writing models, including the ⌊invariant⌋ method used to define invariants. What in the OCLInEcore model was the property ⌊loans⌋ and the operation ⌊isAvailable⌋, are here modeled as methods (using the ⌊**def**⌋ keyword). Multiplicities such as ⌊Loan[∗]⌋ are modeled using Scala's collection libraries, in this case ⌊Set[Loan]⌋. The Scala definitions should be somewhat obvious. It is clear that Scala in this case can model this problem in a manner comparable to OCLInEcore. In addition, Scala offers so much more than OCLInEcore, such as an actual programming language.

```
trait Book extends Model {
  var name: String
  var copies: Int
  var library: Library

  def loans: Set[Loan] =
    library.loans.filter(_.book eq this)

  def isAvailable(): Boolean =
    loans.size < copies

  invariant("CopiesPositive") {
    copies > 0
  }

  invariant("SufficientCopies") {
    loans.size <= copies
  }
}
```

Fig. 4. Scala program modeling books in a library

The only code that has to be written to provide support for writing class invariants is the definition of the class ⌊Model⌋, which is shown in Figure 5. Without going into details, the class defines a method ⌊invariant⌋, which as argument takes a Boolean call-by-name argument. The argument is not evaluated before the method body is executed, rather, it is only evaluated whenever referred to. In this case it is stored, still unevaluated, in a list of invariants, all of which can then be verified on an object of this class with a call of ⌊verify⌋. Note that such invariants (specifications) in addition can be the target of more formal analysis, just as they can in a formal specification language.

3.2 VDM^{++} Specifications

As another example, we shall consider a chemical plant alarm management system, first modeled in VDM^{++} in [18] and also later modeled in Scala in [23], which goes into further detail comparing VDM^{++} and Scala. We show here a slight modification of the VDM^{++} specification as well as the corresponding Scala program. In [18] the example specification was associated with a corresponding UML class diagram to illustrate how the two techniques can co-exist. Here we shall put emphasis on VDM^{++} and its relationship to Scala.

The system shall manage the calling out of experts to deal with operational faults discovered in a chemical plant. Two operations must be provided. ⌊ExpertToPage⌋: Upon detection of a faulty condition, an alarm is raised, and this operation must find an expert on duty able to handle the alarm. Each alarm is associated with a specific qualification required to fix the causing problem, and

```
trait Model {
  type Constraint = Unit ⇒ Boolean

  var constraints: List[(String, Constraint)] = Nil

  def invariant(name: String)(c: ⇒ Boolean) {
    constraints ::= (name, (Unit ⇒ c))
  }

  def verify() {
    for ((n, c) <- constraints) assert(c(), n)
  }
}
```

Fig. 5. Support for defining invariants in Scala

each expert is associated with a set of qualifications. Upon an alarm, an expert must be found, and paged, that is on duty during the corresponding period and with the right qualification. ⌊ExpertIsOnDuty⌋: returns the periods during which an expert is on duty. In addition to providing these two operations, the state of the system must satisfy the following *invariant*: (i) There must be experts on duty during all periods allocated in the system. (ii) For any alarm and for any period, there should exist an expert assigned to that period that has the qualification required to fix the source problem of the alarm.

The VDM^{++} class ⌊Plant⌋ in Figure 6 is part of the model of this system (other classes/types shown in [18] have been left out here: ⌊Alarm⌋, ⌊Period⌋, and ⌊Expert⌋). The body of this class is divided into three sections: *instance variables* (mutable variables), *functions* (with no side-effects), and *operations* (with side-effects). An invariant defined by the function ⌊PlantInv⌋ is imposed on the instance variables. The corresponding Scala program modeling the plant is shown in Figure 7. We shall not go into the further details, except for mentioning the use of the ⌊suchthat⌋ method in the Scala program, defined in the ⌊Model⌋ class, which from a finite set selects an element satisfying a predicate provided as argument.

3.3 Domain-Specific Languages

We consider the ability to define domain-specific languages (DSLs) an essential part of a modeling/programming framework. This form of activity is supported within the UML/SysML community through meta-modeling and profiles. Programming languages have been slower to pick up this concept, although an early language such as Lisp supported macros from its birth. A modern programming language such as Scala supports definition of so-called *internal* DSLs with a collection of a few elegant language features. In this section we shall illustrate this with an example DSL for monitoring event sequences. The example was also

```
class Plant
  instance variables
    alarms : set of Alarm;
    schedule : map Period to set of Expert;

    inv PlantInv(alarms,schedule);

 functions
    PlantInv: set of Alarm * map Period to set of Expert → bool
    PlantInv(as,sch) ==
      (forall p in set dom sch & sch(p) <> {})
        and
      (forall a in set as &
         forall p in set dom sch &
           exists expert in set sch(p) &
             a.GetReqQuali() in set expert.GetQuali());

  operations
    public ExpertToPage: Alarm * Period ⇒Expert
    ExpertToPage(a, p) ==
      let expert in set schedule(p) be st
        a.GetReqQuali() in set expert.GetQuali()
      in
        return expert
    pre a in set alarms and p in set dom schedule
    post let expert = RESULT in
      expert in set schedule(p) and
      a.GetReqQuali() in set expert.GetQuali();


    public ExpertIsOnDuty: Expert ⇒set of Period
    ExpertIsOnDuty(ex) ==
      return {p | p in set dom schedule & ex in set schedule(p)};
end Plant
```

Fig. 6. VDM^{++} model of plant

```
trait Plant extends Model {
  var alarms: Set[Alarm]
  var schedule: Map[Period, Set[Expert]]

  invariant{PlantInv(alarms, schedule)}

  def PlantInv(alarms: Set[Alarm], schedule: Map[Period, Set[Expert]]): Boolean =
    (schedule.keySet forall {p ⇒ schedule(p) != Set()})
      &&
    (alarms forall { a ⇒
      schedule.keySet forall { p ⇒
        schedule(p) exists { expert ⇒
          a.reqQuali in expert.quali
        }
      }
    })

  def ExpertToPage(a: Alarm, p: Period): Expert = {
    require((a in alarms) && (p in schedule.keySet))
    schedule(p) suchthat { expert ⇒
      a.reqQuali in expert.quali
    }
  } ensuring { expert ⇒
    (a.reqQuali in expert.quali) &&
      (expert in schedule(p))
  }

  def ExpertIsOnDuty(ex: Expert): Set[Period] =
    schedule.keySet filter { p ⇒ ex in schedule(p) }
}
```

Fig. 7. Scala program modeling plant

listed in [25]. An *internal* DSL is an extension of the programming language, in this case effectively an API, however, expressed in such a way that use of this API has the flavor of new syntax added to the language.

```
class NoLockCycles extends Monitor {
  "r1" -- 'acquire('t,'l) |-> 'Locked('t,'l)
  "r2" -- 'Locked('t,'l) & 'release('t,'l) |-> remove('Locked)
  "r3" -- 'Locked('t,'l1) & 'acquire('t,'l2) |-> 'Edge('l1,'l2)
  "r4" -- 'Edge('l1,'l2) & 'Edge('l2,'l3) & not('Edge('l1,'l3)) |-> 'Edge('l1,'l3)
  "r5" -- 'Edge('l1,'l2) |-> { if (get('l1) == get('l2)) fail() }
}
```

Fig. 8. Scala program written in the rule DSL, monitoring lock operations

The DSL illustrated here is LogFire [24], created for rule-based programming, and specifically for writing temporal trace properties. We shall not go into the details of LogFire (the reader is referred to [24]), but only show a model written in this DSL, and briefly explain how the DSL is defined. A monitor is specified as a set of rules, each of the form:

$$name \;{-}{-}\; condition_1 \;\&\ldots\&\; condition_n \;\longmapsto\; action$$

A rule consists of a left-hand side: a list of conditions, and a right-hand side: an action. The rules operate on a set of facts, the *fact memory* (implemented as a Rete network [20]) where conditions check the presence or absence of certain facts, and the action adds or deletes facts, or executes any Scala code inserted as part of the action. Figure 8 illustrates a monitor, which monitors $acquire(thread, lock)$ and $release(thread, lock)$ events emitted from an instrumented multi-threaded application, that uses locks to protect against data races. The monitor attempts to determine whether any group of threads access locks in a cyclic manner, which potentially can lead to deadlocks (the classical dining philosopher problem). A cycle is detected if for any lock l there is an edge from l back to itself.

The class contains five rules. Beyond the monitored events `acquire` and `release`, the monitor adds and deletes `Locked(thread,lock)` facts (the thread holds the lock), and adds `Edge(lock1,lock2)` facts, representing that there is an edge from `lock1` to `lock2`, indicating that a thread holds `lock1` while acquiring `lock2`. The class extends the `Monitor` class which contains the DSL definitions that allow us to write rules in this manner. Specifically the functions: `--`, `&`, `|->`, `remove`, and `fail`.

The implementation of this DSL relies on Scala's (1) allowance for methods having symbol names, such as `--` and `|->`, (2) allowance for method calls of the form `obj.method(arg)` to be written as `obj method arg`, and (3) automated insertion (by the compiler) of user-defined *implicit* functions that can lift a value of one type to a value of another type in places where the compiler's type checker fails to type check an expression. For example part of the DSL implementation are the definitions shown in Figure 9. The implicit function ⌊liftRuleName⌋ gets invoked by the compiler automatically when a string is followed by the symbol ⌊−−⌋. It lifts the string (rule name) to an anonymous object, which provides the method ⌊−−⌋, which when applied to a condition returns an ⌊RuleDef⌋ object, which provides the methods ⌊&⌋ and ⌊|−>⌋. This way method calls can be chained together.

```
implicit def liftRuleName(name: String) =
  new {
    def −−(c: Condition) = new RuleDef(name, List(c))
  }

class RuleDef(name: String, conditions: List[Condition]) {
  def &(c: Condition) = new RuleDef(name, c :: conditions)

  def |−>(stmt: ⇒ Unit) {
    addRule(Rule(name, conditions.reverse, Action((x: Unit) ⇒ stmt)))
  }
}
```

Fig. 9. Scala rule DSL implementation

We do notice, however, some drawbacks of the Scala DSL. Notice the quoted names: ⌊'acquire⌋, ⌊'Locked⌋, ⌊'t⌋, etc. These are Scala symbols (elements of the type ⌊Symbol⌋). It is not possible in this form of DSL to avoid these quotes. Likewise, the name of the rule is a string. Furthermore, a monitor in this DSL cannot be type checked without actually running the monitor. That is to say, Scala's DSL defining features are not optimal, although they do make defining internal DSLs somewhat easier.

4 Discussion and Conclusion

A unified modeling and programming framework has to satisfy quite different and contradicting goals. First of all, it has to represent the concepts of the application domain at an adequate level of abstraction such that the specialities of the applications are directly represented and not covered by awkward implementation concepts. Second it has to address the structuring of algorithms and data structures in a way such that programs stay understandable, modular and support the most important methods of structured program development. And finally it has to allow addressing specific implementation properties of execution machines including their operating systems, such that it can be controlled how the implementation uses resources and exploits the possibilities of the execution platform and its hardware. An obvious problem here of course is to what extent the particular application domain influences the programming language, and to what extent this is true for the execution platform and efficiency concerns as well. In the following we shall briefly mention some of the elements to consider when imagining a unified approach to modeling and programming.

Target domains: We can observe three major domains of interest, namely modeling; programming of non-embedded systems, such as web applications, including scripting; and finally programming of embedded and cyber-physical systems. It is clear that these three domains till date have been addressed by different communities and different languages. The question is to what extent these quite different domains could be targeted with the same formalism. Note that the different modeling and programming languages used on different targets have an overwhelming number of language constructs in common, to an extent where this question at least needs to be answered in a scientific manner rather than in an opinionated emotional manner.

Predicate specifications: A formalism must generally support specifying properties as predicates rather than only as algorithms. Predicate-oriented techniques include design-by-contract, including pre- and post-conditions, as well as state invariants. Such can for example be found in Eiffel as well as in SPARK. This concept can be carried further to for example include behavioral sequence specification, such as temporal logics, sequence diagrams, etc. Note, however, that many models are very operational in nature, and hence can very well best be formulated as state machines, or programs (data structures and algorithms).

Programming in the large: A formalism must support programming in the large, and in general provide good modularization and component-based development. One cannot discuss components without discussing concurrency. Concurrency is an essential part of modern programming, especially considering the emergence of multi-core computers. However, concurrency is important at the modeling level as well, where it can serve as a natural way to describe interacting agents. Important concepts include agent systems, message passing based communication, parallel data structures (programming concurrent without knowing it), and distributed programming.

High-level programming: A formalism must support high-level programming as found in modern programming languages. The elegance of functional programming has been praised many times. Nevertheless its breakthrough is only recent. In contrast, object-oriented programming, which in particular addresses encapsulation and reuse, has been very popular for decades now. A language such as Scala integrates the two paradigms nicely, as even early versions of Lisp (CLOS) did. Functional programming means for example functions as values (lambda abstractions) and pattern matching, and of course reliance on recursion. Functional programming is by some considered the best approach to use multi-core systems due to no shared state updates. A key feature of VDM was the introduction of elegant syntax for collections, such as sets, lists and maps. These days such concepts are introduced in languages mostly as libraries. Fortress has built-in syntax for these very similar to VDM. There should be easy ways of iterating through collections to avoid indexing problems, as well as support for parallel computation over such. A formalism should be statically typed, although with type inference, and with allowance for going type less in clearly defined regions to support scripting. Decades of experience in strong type systems should be harvested, including more recent topics such as dependent types, session types, and units.

Low-level programming: A formalism must support low-level programming. Embedded programing often means: no dynamic memory allocation after initialization, no garbage collection, some knowledge of memory layout, even to the point where computation with addresses is used to improve speed. This again means use of low level programming languages such as C. C, however, allows for memory errors and makes programmers less effective as they would otherwise be were they allowed to program in higher-level languages. We need to satisfy the needs encountered by typical C programmers, including offering comparable speed and memory control. This includes support for hardware control and targeting specific execution platforms

Continuous mathematics: A formalism can support modeling of cyber-physical systems. That is: physical systems controlled by computer programs. To model (not program) a cyber-physical system, there is a need for describing continuous behavior using continuous mathematics, including for example differential equations, as supported by for example Modelica [17,4] and Mathematica [3]. Modelica is an object-oriented language for modeling systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents. Models can be simulated. Mathematica is a more broadly scoped symbolic mathematical computation program. A closely related continuous mathematics topic is real-time analysis, as for example performed by real-time model checkers such as UPPAAL [33]. Whether continuous mathematics should be part of a programming solution is a controversial topic.

Domain-specific languages: A formalism must support definition of domain-specific languages. A key to modeling and programming is to capture the relevant concepts of the application domain. UML for example supports meta-modeling and profiles, used for defining (graphical) domain-specific languages. The programming language community is still somewhat behind in this respect. It should be easy for developers to define new domain-specific languages, either external stand-alone, or internal extending the modeling/programming formalism.

Visualization: A formalism must be visualizable. Visualization is of extreme importance, as demonstrated by the relative success of formalisms such as UML and SysML in the engineering community, when compared to formal methods textual notations. People generally are more at ease looking at a two-dimensional illustration when they encounter a problem specification for the first time, and it eases communication. Whether visualization is useful as well for entering and maintaining specifications is another matter. We do believe that visualization and text should match up, such that modification of one leads to immediate change in the other. Visualization is normally static, rendering static models as pictures. However, visualization can also be dynamic, of program executions, illustrating how behavior evolves.

Analysis: A formalism must be analyzable. A key concept in a combined modeling and programming environment is the support for advanced analysis of models/programs, including, but also beyond, what is normally supported in standard programming environments. This ranges from basic built-in support for unit testing, over advanced testing capabilities, including test input generation and monitoring, to concepts such as static analysis, model checking, theorem proving and symbolic execution. A core requirement, however, must be the practicality of these solutions. The main emphasis should be put on automation. The average user should be able to benefit from automated verification, without having to do manual proofs. However, support for manual theorem proving should also be possible, for example for core critical algorithms. Integration of static and dynamic analysis will be desirable: verify what is practically feasible, and test (monitor) the remaining proof obligations.

What modelers do that programmers don't: A central question is how a model/program is represented. Within the formal methods and programming communities this is simple: specifications/programs are represented as text, exactly as typed in by a user. Any tools such as analyzers and compilers read in the text and produce results. The specification language or programming language is defined by a grammar, which succinctly formalizes what is the language of syntax correct texts. In the case of a compiler it produces binary code/byte code, which of course can be stored and used by other programs. However, at the core, the program text is the main reference, from which other formats can be generated. A compiler will produce an abstract syntax tree (AST) from the text, but it only lives as long as the compiler runs. The situation is different in the model-based engineering community. Here the syntax is mostly graphical, and it is not the main representation (and not of main importance). Instead, abstract syntax is the key representation, often stored in XML, from which everything else is generated. In support of collaborative environments there is even a push for storing models in databases, which can be accessed by multiple users simultaneously, hence a more sophisticated approach than the text-based source code repositories often used by programmers. Modelers furthermore have the habits if querying models, transforming models, and generally consider models as data, in contrast to the programming community where data usually are separated from programs. That is, from within a program one can usually not get access to the entire AST of the program itself, although often limited forms of reflection are possible. These different views of representations are worthwhile investigating.

Conclusion We have in this paper outlined some views on the potential in combining modeling and programming, supported by analysis capabilities such as static analysis, model checking, theorem proving, monitoring, and testing. We believe that the time is right for the formal methods/modeling and programming language communities to join forces. To some extent this is already happening in the small. However, we believe that we are standing in front of a major wave of research creating a united foundation for modeling, programming and verification. A cynical argument is that this is all obvious, which may very well be true.

References

1. Documents associated with Object Constraint Language (OCL), Version 2.4. http://www.omg.org/spec/OCL/2.4. Accessed: 2016-06-29.
2. EMF. http://www.eclipse.org/modeling/emf/. Accessed: 2016-07-6.
3. Mathematica. https://www.wolfram.com/mathematica. Accessed: 2016-06-29.
4. Modelica - A Unified Object-Oriented Language for Systems Modeling. Language Specification Version 3.3. https://www.modelica.org/documents/ModelicaSpec33.pdf. Accessed: 2016-06-29.
5. OCLInEcore. https://wiki.eclipse.org/OCL/OCLinEcore. Accessed: 2016-06-28.
6. OCLInEcore online tutorial. http://goo.gl/wR2HvP. Accessed: 2016-06-28.
7. OMG Systems Modeling Language (SysML). http://www.omgsysml.org. Accessed: 2016-07-12.
8. OMG Unified Modeling Language (UML). http://www.omg.org/spec/UML. Accessed: 2016-07-12.
9. The Coq Theorem Prover. https://coq.inria.fr. Accessed: 2016-07-12.
10. The Isabelle Theorem Prover. https://isabelle.in.tum.de. Accessed: 2016-07-12.
11. The PVS Theorem prover. http://pvs.csl.sri.com. Accessed: 2016-07-12.
12. J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.
13. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
14. D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982. ISBN 0-13-880733-7.
15. M. Broy. From chaos to undefinedness. In K. Futatsugi, J. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of his 65th Birthday*, volume 4060 of *LNCS*, pages 476–496. Springer, 2006.
16. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
17. H. Elmqvist, M. Otter, D. Henriksson, B. Thiele, and S. E. Mattsson. Modelica for embedded systems. In *Proceedings of the 7th Modelica Conference, Como, Italy*, pages 354–363, September 2009.
18. J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

19. R. W. Floyd. Assigning meanings to programs. In J. Schwartz, editor, *Mathematical Aspects of Computer Science, Proc. of Symp. in Applied Math. American Mathematical Society, Rhode Island, USA*, pages 19–32, 1967.
20. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
21. K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *POPL*, 1985.
22. C. George, P. Haff, K. Havelund, A. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language.* The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England, 1992.
23. K. Havelund. Closing the gap between specification and programming: VDM^{++} and Scala. In M. Korovina and A. Voronkov, editors, *HOWARD-60: Higher-Order Workshop on Automated Runtime Verification and Debugging*, volume 1 of *EasyChair Proceedings*, December 2011. Manchester, UK.
24. K. Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, 17:143–170, 2015.
25. K. Havelund and R. Joshi. Experience with rule-based analysis of spacecraft logs. In C. Artho and C. P. Ölveczky, editors, *Formal Techniques for Safety-Critical Systems: Third International Workshop (FTSCS 2014), November 2014, Luxembourg*, volume 476 of *Communications in Computer and Information Science*, pages 1–16. Springer International Publishing, April 2015.
26. K. Havelund, R. Kumar, C. Delp, and B. Clement. K: A wide spectrum language for modeling, programming and analysis. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Rome, Italy*, pages 111–122. Scitepress digital library, February 2016.
27. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer, STTT*, 2(4), April 2000.
28. K. Havelund and W. Visser. Program model checking as a new trend. *STTT*, 4(1):8–20, 2002.
29. C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12:567–583, October 1969.
30. G. J. Holzmann. *The Spin Model Checker – Primer and Reference Manual.* Addison-Wesley, 2004.
31. C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 1990. ISBN 0-13-880733-7.
32. L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, 1994.
33. K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *STTT*, 1:134–152, 1997.
34. J. McCarthy. Recursive functions of symbolic expressions and their computation by machines, part I. *Communications of the ACM*, 3:184–195, 1960.
35. J. McCarthy. Towards a mathematical science of computation. In C. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.
36. D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. *Computers and Automata, Microwave Research Inst. Symposia*, 21:19–46, 1971.
37. J. M. Spivey. *The Z Notation - a Reference Manual.* International Series in Computer Science (2nd ed.). Prentice Hall, 1992.
38. The CIP Language Group. *The Munich Project CIP Volume I: The Wide Spectrum Language CIP-L.* Springer, 1985. Volume 183 of LNCS.