Closing the Gap Between Specification and Programming: VDM^{++} and SCALA

Klaus Havelund

Jet Propulsion Laboratory
California Institute of Technology
klaus.havelund@jpl.nasa.gov

Abstract

We argue that a modern programming language such as SCALA offers a level of succinctness, which makes it suitable for program and systems specification as well as for high-level programming. We illustrate this by comparing the language with the VDM^{++} specification language. The comparison also identifies areas where SCALA perhaps could be improved, inspired by VDM^{++}. We furthermore illustrate SCALA's potential as a specification language by augmenting it with a combination of parameterized state machines and temporal logic, defined as a library, thereby forming an expressive but simple runtime verification framework.

1 Introduction

Having worked with formal methods specification languages, in particular worked with a language such as VDM [9] and participated in the design of RSL [17], creates a wish for better times where programming languages look and feel like these specification languages. In this paper we shall explore this thought by comparing the specification language VDM^{++} and the programming language SCALA. We shall ask the question whether these *better times* have appeared. We will argue that SCALA is suitable for program/system specification as well as for high-level programming. We shall also point out, that there are still steps that can be taken which can improve SCALA, inspired by VDM.

Formal methods generally refer to "mathematically-based techniques for the specification, development and verification of software and hardware systems" [15]. The field covers such topics as specification logics, syntax and semantics, proof systems, industrial strength specification languages, theorem proving, and model checking. Amongst one of the early contributions was VDM (Vienna Development Method) [9, 10, 25, 26, 40] and its associated specification language META-IV [9]. The language and method was a result of work done by Dines Bjørner and Cliff Jones at IBM's Vienna Laboratory in the 1970s. META-IV is a so-called *wide-spectrum* specification language, including as a subset an executable language comparable to the combination of an imperative programming language and a functional programming language; as well as including high-level constructs from logic, such as pre/post conditions and invariants (design by contract as later introduced in the EIFFEL programming language [13]), predicate subtypes, predicate logic (universal and existential quantification) permitted as Boolean expressions, and built-in collections such as finite sets, lists and maps accompanied with a large set of operators and strong support for forming set, list and map comprehensions. VDM is a so-called *model-based* specification language where a model of the desired system is created; in contrast to *axiomatic* specification languages, where properties of the system are specified. Indeed an extremely impressive and forward looking language and associated method.

A VDM language standard was subsequently produced in the form of VDM-SL (VDM Specification Language), which combined the so-called "British style", which focused on using only

pre/post conditions and invariants to specify functions and operations; and the "Danish style", which embraced the executable programming language like subset. Two more derivations were later created in RSL [17], a product of the RAISE project [34], and VDM++ [14]. RSL took its starting point in VDM, but followed an algebraic view, where a module consists of a signature and a set of axioms over the typed names introduced in the signature. Functions can have side-effects and even represent processes communicating on channels. Derived syntax was introduced reflecting the classical VDM model-based style. VDM++ took a less drastic approach, "just" adding object-oriented constructs (classes and objects) as well as concurrency to the more classical VDM notation. In this sense VDM++ is interesting since of these systems, it gets the closest to a modern programming language.

The general experience gained by working with VDM is that of abstraction, elegance and convenient notation. So why are we not all using VDM or one of its derivations? A characteristic of VDM, and most formal methods, is that specification fundamentally is considered different from programming, in spite (in this case) of the great overlap with respect to language constructs. For example, META-IV contains as a subset an executable kernel, which is isomorphic to a full blown programming language. In spite of this overlap, the typical use of VDM is to write a specification, and then subsequently either (re)program the system manually (observing the specification), or translate the specification into a program using a special compiler. Such a translation may, however, be considered a non-optimal indirection.

On the other hand, a modern programming language such as SCALA [35] offers language constructs, which makes it a healthy alternative for writing abstract high-level models. SCALA offers a uniform combination of object-oriented and functional programming. Furthermore, it supports definition of internal DSLs (Domain Specific Languages as APIs) through a set of innovative language constructs. SCALA programs are usually very succinct compared for example to JAVA programs, and have the script-like flavor that PYTHON [33] programs have, but with the static typing that JAVA offers, and compiling to the JVM. Sets, lists and maps are part of the SCALA library, as is the case in JAVA. However, with better notation for manipulating these data structures. We discuss the relationship between VDM++ and SCALA, mentioning some of those SCALA library additions that would be needed in order to write VDM++ like models. Several of these have been suggested elsewhere, for example design by contract for SCALA [31]. This paper does not offer new insights at that level, but rather tries to make a broader argument for the importance of investigating the relationship between specification languages and programming languages, as also argued for in [21].

There are other similar programming languages that one could make a comparison with. One of the more interesting in this context perhaps is FORTRESS [16] (under development at the time of writing), which has many similarities with VDM; but also more mature languages such as SML [29], OCAML [30], and HASKELL [19]. One could also make a comparison with other specification languages similar to VDM/VDM++, for example the more recent ASML [18]. However, SCALA is a rather *mature* and *new* language on the horizon, combining object-oriented programming and functional programming in an interesting way and with support for definition of domain specific languages; and VDM was one of the first wide-spectrum specification languages, and has had followers ever since, including the author. Note that although our comparison is with VDM++ (due to its object orientation), the essence compared to is that of the original VDM. Topics not discussed in this paper are concurrency and testing of models/programs. VDM++ has interesting support for both, as does SCALA. Furthermore, the presentation of VDM++ in [14] illustrates how UML [11] class diagrams can be used to support specification in VDM++, with mappings defined between the two notations. This tight integration between textual and graphical specification is promising and a similar approach could be taken for SCALA, although

this topic will not be discussed further in this paper.

On a slightly different note, there have been several attempts to extend programming languages with specification constructs. EML (Extended ML) [27] is an extension of the functional programming language SML (Standard ML) with algebraic specification. ECML (Extended Concurrent ML) [20] extends CML (Concurrent ML) with a logic for specifying CML processes in the style of EML. EIFFEL [13] is an imperative programming language with design by contract features (pre/post conditions and invariants). SPEC# [36] extends C# with constructs for non-null types, pre/post conditions, and invariants. JML [24] is a specification language for JAVA, where specifications are written in special annotation comments which start with an at-sign (@). Finally, JPF (Java PathFinder) [22, 23] is a model checker for JAVA. One of the original motivations behind JPF was to suggest JAVA as a modeling language, in addition to offer model checking for JAVA and to explore the space between testing and model checking for large programs.

The paper is organized as follows. Section 2 presents a specification of a chemical plant alarm system written in VDM++, adopted more or less unchanged from [14]. Section 3 presents a model of the same problem in SCALA. Section 4 discusses the two languages at a construct-by-construct level, focusing mostly on VDM++'s specification-oriented constructs not found in classical programming languages. Section 5 illustrates a SCALA DSL for writing behavioral properties in a combination of parameterized state machines and temporal logic. Finally, Section 6 ends the paper with a discussion. Throughout the paper VDM++ specifications are shown on grey background, whereas SCALA programs are shown on white background.

2 The Alarm Management System in VDM++

In this section we shall introduce a small example specification of a chemical plant alarm management system taken directly from [14] Chapter 2: *Building a Model in* VDM++*: An Overview*. In [14] this example specification is built together with a corresponding UML class diagram to illustrate how the two techniques can co-exist. We shall not make an emphasis on the UML aspect, except to point out that UML could be used in a similar manner for SCALA.

We shall put some focus on the requirements engineering aspect of this modeling in order to emphasize that VDM++ is used as a specification language. The system to be developed shall manage the calling out of experts to deal with operational faults discovered in a chemical plant. Faulty conditions are detected with sensors positioned throughout the plant. Upon detection of a faulty condition, an alarm is raised, and an expert on duty is assigned to handle the alarm. Each alarm is associated with a specific qualification required to fix the causing problem, and each expert is associated with a set of qualifications. Concretely, Figure 1 lists the requirements that have been formulated for this system. A VDM++ specification is composed of classes. In our case three classes are introduced to model the requirements $\mathbf{R}_1$-$\mathbf{R}_8$ in Figure 1: *Expert*, *Alarm* and *Plant*. Class *Expert* is shown in Listing 1.

The body of this class is divided into three sections: *instance variables*, *types* and *operations*. Only entities declared public are visible from outside the class. Instance variables are mutable variables, like fields in JAVA, which can be updated by operations. In this case we have a variable *quali*, which has the type: **set of** Qualification, meaning that at any point in time it holds as value a set of qualifications, where *Qualification* is defined in a separate type definition. The operator **set of** is a type operator taking a type (in this case *Qualification*) as argument and yielding a new type (in this case the type of all sets of qualifications). VDM++ offers several built-in type operators, the most important of which are sets, lists and maps. The variable *quali* represents requirement $\mathbf{R}_4$ (*Each expert has a set of qualifications*).

$\mathbf{R}_1$	*A computer-based system is to be developed to manage the alarms of this plant.*
$\mathbf{R}_2$	*Four kinds of qualifications are needed to cope with the alarms: electrical, mechanical, biological, and chemical.*
$\mathbf{R}_3$	*There must be experts on duty during all periods allocated in the system.*
$\mathbf{R}_4$	*Each expert has a set of qualifications.*
$\mathbf{R}_5$	*Each alarm reported to the system has a qualification associated with it along with a description of the alarm that can be understood by the expert.*
$\mathbf{R}_6$	*Whenever an alarm is received by the system an expert with the right qualification should be found so that he or she can be paged.*
$\mathbf{R}_7$	*The experts should be able to use the system database to check when they will be on duty.*
$\mathbf{R}_8$	*It must be possible to assess the number of experts on duty.*

Figure 1: Requirements for Chemical Plant Alarm System

Listing 1: Class Expert (VDM++)

```
class Expert
  instance variables

    quali : set of Qualification;

  types

    public Qualification = <Mech> | <Chem> | <Bio> | <Elec>;

  operations

    public Expert: set of Qualification ==> Expert
    Expert(qs) ==
      quali := qs;

    public GetQuali: () ==> set of Qualification
    GetQuali() ==
      return quali;
end Expert
```

Type definitions have the form: *name = type_expression*, and simply defines the *name* to stand for the type denoted by the type expression. In this case type *Qualification* is defined as the union ($t_1 \mid t_2$ denotes the union of types t_1 and t_2) of four types, each of which is a so-called *quote type* containing one element. For example <Mech> is a quote type containing the single constant <Mech>. The definition of type *Qualification* represents requirement $\mathbf{R}_2$ (*Four kinds of qualifications are needed to cope with the alarms: electrical, mechanical, biological, and*

chemical).

Class *Expert* has two operations, *Expert* and *GetQuali*. An operation is an action that accesses the state of an object (read or write), or in the case of *Expert*, which has the same name as the class: creates an object – it is an object *constructor* (similar to JAVA's constructors). Each operation is defined by first giving its type and then its definition. An operation type has the form $t_1 ==> t_2$, where t_1 is the argument type and t_2 is the result type. The ==> arrow indicates that state can be accessed, in contrast to functions which we shall se examples of later. The constructor *Expert* takes as argument a set of qualifications and returns an *Expert* object. It creates a new object, assigning the argument set to the *quali* variable with an assignment statement. Note that VDM++ in addition to various high-level specification constructs contains a programming language like subset. The operation *GetQuali* takes no arguments ('()' represents the empty list of arguments) and just reads the state, returning the contents of the *quali* variable. This is what is usually referred to as a *getter method* in JAVA.

Alarms are objects of the *Alarm* class shown in Listing 2. Since strings are not a built-in type in VDM++, the type *String* is introduced as a sequence of characters[1]. The class defines two instance variables: *descr* is a text description of the alarm, while *reqQuali* contains the qualification that it requires to fix the problem causing the alarm (it is here assumed that there is only one such). The type *Expert'Qualification* refers to the type *Qualification* in class *Expert*. The class has a constructor and a getter method, as before. This class represents requirement $\mathbf{R}_5$ (*Each alarm reported to the system has a qualification associated with it along with a description of the alarm that can be understood by the expert*).

The last class *Plant*, shown in Listing 3 represents the rest of the requirements. It defines two instance variables: *alarms* denotes the set of alarms that can be activated, and *schedule* denotes the current schedule, mapping each time period to the set of experts on duty for that period. A map is like a function, but with a finite domain in which one can look up a key (in the domain) to get an associated value (in the range). The type *Period* is defined in the type section as being a token. Tokens are atomic entities on which only equality is defined. This reflects that we do not care about what exactly time periods are. Tokens are constructed as follows: **mk_token**(*expr*), for example **mk_token("**period−3**")**. The declaration of the two instance variables are followed by the following invariant:

```
inv PlantInv(alarms,schedule);
```

The '**inv** *boolean_expression*' construct defines an invariant that always have to be true. In this case the Boolean expression denoting the invariant is a call of the function *PlantInv* defined in the functions section. The body of this function contains two sub-invariants, each a universal quantification. The first forall-expression states that for all periods in the domain of the schedule map, the schedule of this period should not be empty. This represents requirement $\mathbf{R}_3$ (*There must be experts on duty during all periods allocated in the system*). The second forall-expression states that for all alarms a in the set of alarms, and for all periods p in the domain of the schedule, there exists an expert assigned to p, for which it holds that a's qualification is in the set of qualifications had by the expert p. This represents part of requirement $\mathbf{R}_6$ (*Whenever an alarm is received by the system an expert with the right qualification should be found so that he or she can be paged*). The invariant demonstrates the use of predicate logic.

The *Plant* class defines the three operations *ExpertToPage*, *NumberOfExperts* and *ExpertIsOnDuty* plus the *Plant* constructor. The constructor is associated with the pre-condition (as part of the definition of the operation) that the plant invariant must be true for its arguments.

[1] VDM++ uses the term *sequence* instead of *list*. We shall use the unifying term *list* throughout the remaining part of the paper.

Listing 2: Class Alarm (VDM++)

```
class Alarm
  types

    public String = seq of char;

  instance variables

    descr     : String;
    reqQuali : Expert`Qualification;

  operations

    public Alarm: Expert`Qualification * String ==> Alarm
    Alarm(quali,str) ==
    ( descr := str;
      reqQuali := quali
    );

    public GetReqQuali: () ==> Expert`Qualification
    GetReqQuali() ==
      return reqQuali;
end Alarm
```

The constructor should never be called on arguments that do not satisfy the pre-condition. The operation *ExpertToPage* computes for a given alarm and a given period an expert that has the qualifications to handle the fault causing the alarm. This represents requirement $\mathbf{R}_6$ (*Whenever an alarm is received by the system an expert with the right qualification should be found so that he or she can be paged*). The operation illustrates how it is possible to define an operation with a body (as before), a pre-condition, and a post-condition. The side-effect and returned result of the body should satisfy the post condition. Any of these three concepts can be left out. An operation can for example be defined with just a pre-post condition pair. In this case the operation will not be executable and can therefore not be tested. The pre-condition states that the alarm must exist and the period must be defined (in the domain of the *schedule*). The post-condition states that the resulting expert returned by the operation (represented by **RESULT**) should be assigned to that period, and that the alarm's associated qualification should be amongst the expert's qualifications. A let-expression is used to bind the result of an expression to an non-mutable identifier (defining a constant). The body of the operation looks in this case very much like the post-condition. It uses a non-deterministic choice operator reading: let *expert* be some member in the set *schedule(p)* such that *a.GetReqQuali()* is in the set *expert.GetQuali()* and then return this expert.

The operation *NumberOfExperts* returns the number of experts assigned to a particular period. The **card** operator returns the cardinality of a set. This represents part of requirement $\mathbf{R}_8$ (*It must be possible to assess the number of experts on duty*). The *ExpertIsOnDuty* operation returns for a given expert the periods he or she is assigned to (this is a case where there is no

Listing 3: Class Plant (VDM++)

```
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());

  types
    public Period = token;

  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 NumberOfExperts: Period ==> nat
    NumberOfExperts(p) == return card schedule(p)
    pre p in set dom schedule;

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

    public Plant: set of Alarm * map Period to set of Expert ==> Plant
    Plant(als,sch) ==
      ( alarms := als; schedule := sch)
    pre PlantInv(als,sch);
end Plant
```

Listing 4: Classes Qualification, Period, Expert, and Alarm (SCALA)

```
trait Qualification
case object Mech extends Qualification
case object Chem extends Qualification
case object Bio extends Qualification
case object Elec extends Qualification

class Period(name: String)

case class Expert(quali: Set[Qualification])
case class Alarm(descr: String, reqQuali: Qualification)
```

pre or post condition). This represents part of requirement $\mathbf{R}_7$ (*The experts should be able to use the system database to check when they will be on duty*). The body of this operation is a so-called *set comprehension* reading: the set of periods p where p is in the (domain of the) schedule, and the expert is scheduled for that period. Set comprehensions, as well as list and map comprehensions form a core of the VDM++ language.

3 The Alarm Management System in SCALA

In this section the same set of requirements will be modeled as a SCALA program. Listing 4 shows several class and object definitions corresponding to the VDM++ types *Qualification* and *Period* and the VDM++ classes *Expert* and *Alarm*. The VDM++ union type *Qualification* is in SCALA modeled using an object hierarchy. SCALA has an enumeration type concept, but the here shown approach is more general and supports variant types in general known from functional programming languages. *Qualification* is modeled as an abstract class with no contents (a **trait** is a form of abstract class), and each alternative qualification is modeled as a *case object* that extends this class. SCALA allows to define an object directly without first defining a class and then instantiating it. In addition, defining an object as a case object allows it to be used in pattern matching and to be printed as defined (for example *println(Mech)* will print *Mech*), and not as for example *alarm.Mech$@690da5eb* had the **case** keyword been omitted.

Type VDM++ type *Period* is modeled as a class. Since VDM++ tokens can be created as follows: **mk_token(**"period−3"**)**, we define it as parameterized with a string, hence allowing us to create periods as follows: **new** Period("period−3"). The classes *Expert* and *Alarm* are defined as case classes with parameters. The **case** keyword allows objects to be created without the use of the keyword **new**, allows for pattern matching over terms of the class, allows to access the arguments from outside the generated object, defines equality to include equality of arguments, and supports printing of objects and their arguments in a humanly readable form. For example an expert can be created as follows: Expert(Set(Mech,Bio)). Suppose e denotes such an object we can access its parameter with the notation: *e.quali*.

The VDM++ specification uses the same technique as suggested for JAVA developers, to not expose the internal variables of a class, and to use getters and setters to read from respectively write to these variables. For example, the VDM++ specification of the *Expert* class is rather

lengthy with a constructor defined inside the class as well as a getter method. The corresponding SCALA class definition is just one line. First of all, in SCALA the arguments to a class constructor are given as part of the class header, saving us from an explicit constructor definition inside the body of the class. Second, the reason for using getter and setter methods is that it allows an implementer to change the definition of the internal state, while keeping the promised getter and setter methods. However, SCALA allows to change the public variables of a class, while keeping the same interface. This is done by allowing the implementer that changes the variables to define new methods that when called look like accesses (reads and writes) to the old variables existing before the change. This is possible for example since it is allowed to omit parenthesis as part of the method name. We shall not go into this subject further here.

The main definitions are in class *Plant*, as shown in Listing 5. Recall from above that constructor parameters are written as arguments to the class definition. The VDM++ constructor has the pre-condition that the invariant should be satisfied. This is in SCALA modeled with an assert statement as the first line of the class (all statements at the outermost level of a class are executed when an object is created). The *PlantInv* function is defined starting with the keyword **def**. SCALA does not have universal and existential quantification as part of the language (they are not keywords). However, any collection, such as a set, list or map, has a *forall* and an *exists* method defined on it. For example a Set[A] object defines a method with the following signature:

```
def forall (p: A => Boolean): Boolean
```

The first line of the body of the *PlantInv* function reads as follows. The expression *schedule.keySet* denotes the domain (set of keys) of the the *schedule* map. On this set the method *forall* is called with the following function (predicate) as argument: 'p => schedule(p) != Set()'. This predicate takes as argument a period *p* and returns true iff. the set denoted by that period in the schedule is non-empty. Note that in SCALA a call of a method *m* on an object *o* with an argument *a*, can be written as: *o m a*, as an alternative to the more traditional dot-notation style: *o.m(a)*. With this explanation the rest of the method should be understandable.

The *ExpertToPage* method is defined with a pre-condition, a body, and a post-condition. The definition has the form:

```
def ExpertToPage(a: Alarm, p: Period): Expert = {
  require(pre_condition)
  body
} ensuring {
  result => post_condition
}
```

It uses the two library functions *require* and *ensuring*. The pre-condition is expressed with a call of the SCALA *require* method, which is intended to be used as a pre-condition of a method. It behaves as *assert* except that it throws an *IllegalArgumentException* instead of a *AssertionError* exception. The post-condition is expressed using the *ensuring* method, the application of which has the form:

```
expression ensuring predicate
```

which first evaluates the expression to a value v, and then behaves as: 'assert(predicate(v));v'. We shall in section 4 explain how this really works. The methods *NumberOfExperts* and *ExpertIsOnDuty* are simple. We recall that methods, such as *size* (calculating the cardinality of

Listing 5: Class Plant (SCALA)

```
class Plant(als: Set[Alarm], sch: Map[Period, Set[Expert]]) {
  var alarms: Set[Alarm] = als
  var schedule: Map[Period, Set[Expert]] = sch

  assert(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 NumberOfExperts(p: Period): Int = {
    require(p in schedule.keySet)
    schedule(p) size
  }


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

a set) and *filter* (selecting the members of a set satisfying a predicate), can be called without dot-notation.

4 Comparison by Construct

VDM++ as well as SCALA both offer a combination of object-oriented and functional programming. In SCALA the integration of these two concepts was part of the initial language design and is characterized by all values in SCALA being objects, including numerical values and functions. In contrast, object orientation was added to VDM after its original design to obtain VDM++. Both languages offer the traditional language constructs from these two paradigms; from object-oriented programming: classes and objects, mutable variables, assignment statements, sequential statement composition, conditionals, while loops, for loops over collections[2], exceptions, and threads (concurrency); and from functional programming: higher order functions, currying, lambda abstractions (anonymous functions), algebraic data types and pattern matching. In addition to these concepts VDM++ offers a collection of specification-oriented constructs that we shall discuss in more detail in this section. These are: built-in composite types for sets, lists and maps, a special general concept of union types, type invariants, also occasionally referred to as predicate subtypes, mutable state invariants and pre/post conditions (design by contract), universal and existential quantification (predicate logic) and non-deterministic expressions and statements. VDM++ offers a specification language for threads, which we shall not discuss in this paper. Threads were not part of the original VDM specification language.

4.1 Sets and Lists

One of the original contributions of VDM was the built-in set, list and map data types. We shall refer to these as *collections*. These data types have subsequently made it into standard programming languages today, either as library constructs (JAVA and SCALA) or built-in (PYTHON). Most functional programming languages support lists as a built-in data type. In VDM++ values of these types can be constructed in one of three ways: (i) by explicitly enumerating the elements, (ii) by a comprehension from (iteration over) another collection, and (iii) by applying operators. The following three definitions show these three forms for sets in VDM++:

```
values
  s1 : set of int = {1,2,3};
  s2 : set of int = {x+1 | x in set s1 & x > 2};
  s3 : set of int = s1 union s2;
```

The set *s1* is constructed by explicit enumeration. The set *s2* is a set comprehension and is derived from the set *s1*, equaling {4}. The expression is of the form: {f(x) | x **in set** s **&** p(x)}, for some set s, for some some total function f: **int** +>**int**, and for some total predicate p: **int** +>**bool**. A +>B represents the set of total functions from *A* to *B* while A −>B represents the set of partial functions from *A* to *B*. Sets can furthermore be formed by taking unions, as set *s3* which equals {1,2,3,4}, intersections, etc (a large variety of set operators are available). Set comprehensions can also be formed by ranging over types, as in {x+1 | x : **nat &** x < 3}. Such sets are not used often and they have no counterpart in SCALA. Lists can be constructed

[2]Note that for-loops over collections, such as sets and lists, only has become standard in modern programming languages more recently.

in a similar manner to sets, however using square brackets instead of curly brackets, as in the explicitly enumerated list: [1,2,3] .

SCALA allows to construct values of these types in the same manner: by explicit enumeration, by comprehension, and by applying operators. Concerning comprehension, however, two approaches are possible, one using the functions *filter* and *map*, and one using the **for−yield** construct. The SCALA definitions of the above sets using *filter* and *map* become as follows:

```
val s1: Set[Int] = Set(1, 2, 3)
val s2: Set[Int] = s1 filter (x => x > 2) map (x => x+1)
val s3: Set[Int] = s1 union s2
```

The set *s1* is constructed by explicit enumeration using an object constructor. Note that in SCALA a set is an object – there is no special syntax for sets. The set *s2* is constructed from *s1* using two methods defined on collections: *filter* and *map*. An expression of the form: '*s* filter *p*', where *s* is a collection and *p* is a predicate over the elements in the collection, returns the sub-collection of *s* containing only the elements satisfying *p*. An expression of the form '*s* map *f* ', where *s* is a collection and *f* is a total function f : A =>B, returns the set of elements f(x) for all *x* in *s*. The union of two sets is defined in the same manner as in VDM++. Note that since SCALA allows symbols as operators, many of VDM++'s operator syntax could be modeled if that was desired.

The use of *filter* and *map* is the standard way to define comprehensions in functional programming languages. SCALA has a special **for−yield** construct:
for (x <− s **if** p(x)) **yield** f(x), that corresponds more directly to VDM++'s comprehension: {f(x) | x **in set** s **&** p(x)}. With this construct the set *s2* can be programmed as follows in SCALA:

```
val s2: Set[Int] = for (x <− s1 if x > 2) yield x + 1
```

Lists are modeled in a similar manner to sets, but using a different constructor:

```
val l: List[Int] = List(1, 2, 3)
```

Note that the **for−yield** construct is collection-preserving in the sense that if the base collection ranged over is a set it will return a set, and if it is a list it will return a list. Explicit functions exist for converting between collection types in case this is needed.

4.2 Maps

Maps can be defined in a similar manner. The following VDM++ example defines a set of week days and a set of hours in a day, as well as two calendars mapping day-hour pairs to calendar entries. *cal2* is an update of *cal1* with a meeting scheduled for Tuesday from 8-10.

```
types
  String = seq of char;
  Calendar = map String * int to String;

values
  week : set of String = {"mon","tue","wed","thu","fri","sat","sun"};
  hours : set of int = {1,...,24} ;
```

```
  cal1 : Calendar =
    {mk_(day,hour) |-> "empty" | day in set week, hour in set hours};

  cal2 : Calendar =
    cal1 ++ {mk_("tue",hour) |-> "meeting" | hour in set {8,...,10}};
```

The same definitions in SCALA would become:

```
type Calendar = Map[(String,Int),String]

val week = Set("mon","tue","wed","thu","fri","sat","sun")
val hours = 1 to 24

val cal1 : Calendar =
  for (day <- week; hour <- hours) yield ((day,hour) -> "empty")

val cal2 : Calendar =
  cal1 ++ (for (hour <- 8 to 10) yield (("tue",hour) -> "meeting"))
```

Note that *cal1* is created from the sets *week* and *hour*. Since SCALA's **for−yield** construct, as already mentioned, preserves collection type, the result is a *set* of pairs of the form ((day,hour),"empty") – noting that an expression of the form: a ->b is equivalent to the pair/tuple: (a,b). To turn it into a map type we would have to apply the *toMap* method on this set of pairs. Alternatively, as we have done here, we can define a so-called *implicit function* that converts all sets of pairs into maps:

```
implicit def pairs2Map[A,B](pairs: Set[(A,B)]): Map[A,B] = pairs.toMap
```

An implicit function in SCALA is not meant to be called explicitly in the program. Rather, calls of these functions are inserted by the compiler in places where an expression does not type check, but where it would if one of these functions were applied in that context to one of the sub-expressions. There has to be a unique such implicit function, otherwise the compiler will fail to "resolve" the type error.

4.3 Record and Union Types

VDM++ offers a construct inherited from VDM for constructing records and a very general union type operator. The following example illustrates these two concepts in a VDM++ model of trees of integers, with a function for summing up the elements in a tree.

```
class TreeSum
  types
    Tree = <Empty> | Node;
    Node :: lt : Tree nval : int rt : Tree
  functions
    sum : Tree +> int
    sum(t) ==
      cases t :
        <Empty> -> 0,
        mk_Node(l,n,r) -> sum(l) + n + sum(r)
```

```
      end
end TreeSum
```

The *Tree* type is defined as the union (denoted by '|') of the quotation type <Empty>, which only contains that value, and the type *Node*, which is defined as a record (indicated by the symbol '**::**') consisting of a left tree (named *lt*), a right tree (named *rt*) and a numeric value (named *nval*). The function takes a tree as argument and cases out on the two alternatives, using the **mk_**Node(l,n,r) pattern to match a composite tree value. Elements of type *Node* are created with the **mk_**Node constructor. That is, if *left* and *right* are trees then **mk_**Node(left,4,right) is a node, and consequently a tree. The SCALA version of this model is shown below.

```
class TreeSum {
  trait Tree
  case object Empty extends Tree
  case class Node(lt: Tree, nval: Int, rt: Tree) extends Tree

  def sum(t: Tree): Int =
    t match {
      case Empty => 0
      case Node(l, n, r) => sum(l) + n + sum(r)
    }
}
```

In SCALA there is no general un-tagged union operator, although at the time of writing it is being explored by the SCALA design team. A union type is instead modeled by declaring a type *Tree* (as a trait) and by defining the various alternatives as sub-classes of this, similar to what one would do in for example JAVA. There is also no special notion of record type since classes can be used for this, and in particular **case** classes which allow us to: create objects of the class without using the **new** keyword, do pattern matching over the constructor, rely on equality tests based on arguments to the constructor, get access to the elements via the formal parameter names, and print in a recognizable format. Note that the *Empty* alternative has to be defined as an object since it has no parameters and we are interested in only one instance. The defined function looks very similar to the VDM++ version except for the different syntax for a case construct.

4.4 Type Invariants

VDM++ allows to define a type S as a subtype of another type T, including only members of T that satisfy a certain predicate. For example a type *Degree* can be defined as a subtype of the integers as follows, where we have also defined a function over degrees:

```
types
  Degree = int
  inv x == 0 <= x and x <= 360

functions
  turn : Degree * int +> Degree
  turn(d,x) == (d + x) rem 360
  pre x >= 0
```

The semantics is that *Degree* denotes the set of integers in the interval 0 to 360. Predicate subtypes also exist in other specification languages. For example in RSL [17] one can express the type perhaps more naturally as a set comprehension as follows, with a similar semantics:

```
type Degree = {| x : Int :- 0 <= x /\ x <= 360 |}
```

On application of the function *turn* we will get a type error if it is applied outside its range $0 \ldots 360$ on its first argument, if its second argument violates the pre-condition, or if a value outside the degree range is returned. Note that we could constrain the second argument to be of type **nat** but choose to use a pre-condition for illustration purposes. For example, the following application should result in a type error: turn(−1,20). Of course, such type checking cannot be fully automated in the general case. Type checking would require a call to a theorem prover. However, such subtypes is a centralized and succinct way of expressing constraints compared to for example scattering pre- and post-conditions throughout function definitions, and they can be checked at runtime (with an option for switching off such checks for efficiency purposes).

SCALA does not have predicate subtypes. One can, however, simulate subtypes using implicit conversion functions as suggested in [32]. The resulting "type" definition will consist of three SCALA definitions: a case class, and two implicit function definitions, as follows.

```
case class Degree(x:Int) {
  assert(0 <= x && x <= 360)
}
implicit def convInt2Degree(x:Int):Degree = new Degree(x)
implicit def convDegree2Int(d:Degree):Int = d.x

def turn(d: Degree, x: Int): Degree = {
  require(x >= 0)
  (d + x) % 360
}
```

The case class *Degree* defines the actual type. It contains an assertion in the body asserting the subtype predicate whenever an object of this class is created. Two implicit conversion functions convert respectively from integers to degrees and back. The compiler will based on these conversion functions compile the definition of the *turn* function as follows:

```
def turn(d: Degree, x: Int): Degree = {
  require(x >= 0)
  convInt2Degree((convDegree2Int(d) + x) % 360)
}
```

An application of the function, as as for example: turn(−1,20), will be converted into: turn(convInt2Degree(−1),20), which will cause the assert to fail. The SCALA solution is obviously not very succinct, but offers an interesting solution to the problem.

4.5 State Invariants

VDM++ also supports predicate "sub-typing" of state variables. The example in Listing 3 illustrates this case. We have repeated the relevant piece of VDM++ specification below, adding an additional method for deleting an expert from a period (which first checks the invariant on a copy of the new schedule, and if true updates the schedule - note that this is a specification,

and hence is allowed to be inefficient).

```
class Plant
  instance variables
    alarms   : set of Alarm;
    schedule : map Period to set of Expert;
  inv PlantInv(alarms,schedule);
...
operations
  public DeleteExpert : Period * Expert ==> ()
  DeleteExpert(p,e) ==
    let newSchedule = schedule ++ {p |-> schedule(p) \ {e}} in
      if (PlantInv(alarms,newSchedule)) then schedule := newSchedule
    pre p in set dom schedule;
   ...
  end Plant
```

In case an operation contains more than one statement, the invariant has to hold in between every single statement. If one wants to avoid this, the statements must be wrapped in an **atomic**($assignStmt_1; \ldots; assignStmt_n$) construct.

We note in the SCALA solution in Listing 5 that the invariant is expressed as an assertion that is executed as the first thing when an object of class *Plant* is created. This works as long as the state is never updated once created. However, with the addition of the *DeleteExpert* operation, we need to come up with a better solution. Odersky in [31] suggests the following pattern for defining invariants in SCALA. We define a trait *Invariant* with a variable that contains a list of invariants to be checked, and a method *invariant* for adding new invariants to this list. In addition, we define a function *atomic* that as argument takes a fragment of code, and executes a check of the invariants before as well as after the execution of the code. The definition of *Invariant* in SCALA is as follows.

```
trait Invariant {
  private var invs: List[() => Boolean] = List()

  def invariant(cond: => Boolean) {
    assert(cond)
    invs = (() => cond) :: invs
  }

  def atomic[T](body: => T): T = {
    for (inv <- invs) assert(inv())
    val result = body
    for (inv <- invs) assert(inv())
    result
  }
}
```

Note that the argument types of the methods *invariant* and *atomic* are both of the form: '=>X' for some type *X*. It appears as a function type but without any argument type. Such a "type" represents a call by name argument. When one of these methods is applied to a term, the term

is not evaluated before the method is applied, rather, it is passed on to the body. The *invariant* wraps it inside a lambda abstraction (to form a closure): '() =>cond', and adds it to the list of invariants (:: adds an element to a list). We have modified the method compared to [31] by asserting the condition before insertion in the list. The *atomic* method (named *step* in [31]) executes its argument code piece, with a check of the invariants before as well as after. We can now formulate our plant in SCALA using this trait.

```
class Plant(als: Set[Alarm], sch: Map[Period, Set[Expert]]) extends Invariant {
  var alarms: Set[Alarm] = als
  var schedule: Map[Period, Set[Expert]] = sch

  invariant{PlantInv(alarms, schedule)}
  ...
  def DeleteExpert(p:Period, e: Expert) =
    atomic {
      require(p in schedule.keySet)
      val newSchedule = schedule + (p -> (schedule(p) - e))
      if (PlantInv(alarms,newSchedule)) schedule = newSchedule
    }
  ...
}
```

The *Plant* class adds the invariant, and the body of the *DeleteExpert* method is a call of the *atomic* method with the code as argument.

4.6 Functions and Operations

In SCALA there is no distinction between pure functions (with no side effects) and operations with side effects as there is in VDM++. There is only a notion of functions, with or without side-effects. There is the notion of a method, which is a function that is a member of some class, trait, or singleton object. SCALA allows parametric polymormism (functions defined with a type parameter) as VDM++. In addition SCALA also allows parametric polymorphism at the class and trait level. The perhaps most explicit difference is how pre- and post-conditions are handled. In Listing 5 we saw how pre- and post-conditions are expressed using a pair of library functions *require* and *ensuring*. As already mentioned, the *require* function is like *assert*. The *ensuring* function is more interesting. The expression: '*expr* ensuring *pred*' evaluates *expr* to a value. The predicate *pred* is then applied to this value and if true, the value is returned, otherwise an exception is thrown. The *ensuring* function is defined as follows in SCALA's `Predef` object.

```
final class Ensuring[A](val x: A) {
  def ensuring(cond: A => Boolean): A = { assert(cond(x)); x }
}

implicit def any2Ensuring[A](x: A): Ensuring[A] = new Ensuring(x)
```

Here a class *Ensuring* is defined, parameterized with a value x of a parametric type A. The class defines the *ensuring* method, which takes a predicate on A, checks the assertion and returns x if the assertion has not failed before then. The implicit function *any2Ensuring* converts any value x to an *Ensuring* object. To see how this works, consider the expression:

'*expr* ensuring *pred*'. The SCALA compiler will treat this as '*expr*.ensuring(*pred*)', which will not type check since the method *ensuring* will not be defined on the value returned by *expr*. The compiler will then insert a call of the implicit conversion function *any2Ensuring* as follows: 'any2Ensuring(*expr*).ensuring(*pred*)'.

This SCALA approach to pre- and post-conditions handles the case of pure functions well, but does not handle functions that have side-effects. This is because we in a post-condition cannot refer to the old value of a variable (at function entry point), as in VDM++: $x\tilde{}$ (for old value of x), and as normally allowed in pre-post condition formalisms. We would have to explicitly store such values at function entry point.

VDM++ furthermore allows to define a function or operation only with a pre- and post-condition, without an implementing body. To achieve this in SCALA one can, as suggested in [31], define a function *unimplemented* which throws an exception and call it as part of the body. Using this style, however, we would have to replace the call of *unimplemented* with a real body once we determine what it shall be. If one wants to separate specification from implementation one will have to replace the call of *unimplemented* with a call that delegates to an implementation, for example as also described in [31].

4.7 Quantifiers, Choice

VDM++ offers expressions which have a foundation in logic. These include universal and existential quantification and let-be-such-that expressions. These constructs can quantify/range over infinite types as well as over finite sets, whereas in SCALA their corresponding constructs only can range over finite sets. However, usually this is sufficient in most VDM++ models seen in practice. We saw examples of quantifications in SCALA in the definition of function *PlantInv* in Listing 5. These were just functions (methods) defined on collections.

VDM++ has, as many functional programming languages, let-expressions of the form '**let** x = exp1 **in** exp2'. SCALA does not offer such a construct but allows you to define constants with the **val** construct. The example would become: '**val** x = exp1; exp2'. VDM++ also offers a '**def** x = exp1 **in** exp2' construct in case 'exp1' has side effects. SCALA does not make this distinction just as it makes no distinction between functions and operations.

VDM++ in addition offers a so-called let-be-such-that construct, as illustrated in the definition of the *ExpertToPage* operation in Listing 3. We repeat the expression here:

```
let expert in set schedule(p) be st
  a.GetReqQuali() in set expert.GetQuali()
in
  return expert
```

This expression is of the form: '**let** id **in set** exp1 **be st** exp2 **in** exp3'. That is, choose a value id from the set exp1 such that exp2 is satisfied and then return exp3. There is a non-deterministic choice to be made if more than one element in exp1 satisfies exp2. In the example above, we need to select an expert in the set schedule(p) who's set of qualifications contains the qualification associated with the alarm in question. In SCALA this can be modeled as application of a function to the set and the predicate, which selects an element from the set (for example the first) satisfying the predicate. Our SCALA version becomes (we do not provide the definition of the function *suchthat* here):

```
schedule(p) suchthat {expert =>
  a.reqQuali in expert.quali
```

```
}
```

VDM++ offers non-determinism in the form: '||(stmt1, stmt2, ..., stmtn)'. It represents the execution of the component statements in an arbitrary (non-deterministic) order (not simultaneously). This parallel operator can be defined as follows in SCALA.

```
import java.util.Random

val rand = new Random(System.currentTimeMillis())

def ||[T](statements: (Unit => T)*) {
  var stmts = statements.toList
  while (!stmts.isEmpty) {
    val choice = rand.nextInt(stmts.length)
    val (stmts1,stmtChoice::stmts2) = stmts.splitAt(choice)
    stmts = stmts1 ++ stmts2
    stmtChoice()
  }
}

implicit def convStmt2Lamda[T](stmt: => T): Unit => T =
  ((x: Unit) => stmt)
```

In the scope of these definitions, the following statement will execute the different println statements in random order:

```
||(println("work"), println("rest"), println("sleep"))
```

It is obvious how one would define a non-deterministic choice operator in the same manner.

5 Temporal Specification

In the previous sections we have illustrated how SCALA can be used as a modeling language in a manner comparable with how VDM++ is intended to be used. The modeling approach is the classical VDM approach of defining data structures based on sets, lists and maps, and then define operations and functions on these using programming, pre/post conditions, invariants and elements of predicate logic. In this section we shall illustrate a SCALA DSL named TRACECONTRACT [4, 5] for writing parameterized state machines, which allows anonymous states, thereby allowing a combination of state machines, temporal logic and code. The logic is executable in the sense that specifications can be used for monitoring an application module, either as it executes, or for post-mortem analysis of logs produced by the module during its execution. The solution is an *internal* DSL (a SCALA API) in contrast to our earlier logics for monitoring, which were *external* DSLs (relying on parsers) [2, 7, 6, 8, 3]. Other runtime verification systems include [39, 38, 12, 1]. Some work has been done in the past on extending VDM with temporal logic, for example [28]. Such extensions usually require a substantial amount of theoretic work and implementation effort. The extension shown here has an implementation of less than 120 lines of uncommented code.

As our example application, assume that the plant is instrumented to write events of importance to logs, and that these logs are analyzed after their creation. Assume that the events writ-

$\mathbf{R}_9$	*An alarm must be repaired within one hour by an expert possessing the required qualifications.*
$\mathbf{R}_{10}$	*Two consecutive different alarms without a repair of the first alarm in between must be reported.*

Figure 2: Temporal Requirements for Chemical Plant Alarm System

ten to the logs are of the form *AlarmOn(alarm,time)* and *Repair(alarm,expert,time)*, indicating respectively that an alarm occurred at a certain time, and that a repair of an alarm occurred by an expert at a certain time. Consider the two additional chemical plant alarm system requirements shown in Figure 2. The objective is to write a specification of these two requirements. The first task is to specify the events *AlarmOn(alarm,time)* and *Repair(alarm,expert,time)*, by defining a trait *Event* and defining the two event alternatives as case classes subclassing *Event*:

```
trait Event
case class AlarmOn(alarm: Alarm, time: Int) extends Event
case class Repair(alarm: Alarm, expert: Expert, time: Int) extends Event
```

Note that time stamps are just modeled as data (in milliseconds). The two requirements can now be formalized as the following TRACECONTRACT monitor:

```
class AlarmMonitor extends Monitor[Event] {
  val oneHour = 1000 * 60 * 60

  always {
    case AlarmOn(alarm, time) => AlarmOccurred(alarm, time)
  }

  def AlarmOccurred(alarm: Alarm, time: Int) =
    eventually {
      case AlarmOn(alarm2, _) if alarm != alarm2 => False
      case Repair(`alarm`, expert, time2) =>
        (alarm.reqQuali in expert.quali) && time2 - time <= oneHour
    }
}
```

The class *AlarmMonitor* extends the class *Monitor* which is defined as part of TRACECONTRACT, and which offers all the temporal primitives needed for writing state machines and temporal specifications. The monitor reads as follows: it is always the case, that if an *AlarmOn(alarm, time)* event is observed, the monitor transitions to the *AlarmOccurred(alarm, time)* state - essentially a function call. This function in turn returns an *eventually* state, which has to be exited before the end of the log is reached, otherwise an error is reported. If a *Repair(`alarm`, expert, time2)* is observed (where the quotes around *alarm* indicate the same value as *alarm*), then it is checked that that expert has the right qualifications and that the repair occurs within one hour. If on the other hand an *AlarmOn(alarm2, _)* event is observed before the repair (ignoring the time stamp) where *alarm2* is different from *alarm*, an error is reported.

The *always* and *eventually* constructs are SCALA functions each of which take as argument a so-called partial function (lambda abstraction) as argument. In SCALA such a partial function

can be defined as a block of case statements defining the values for which the partial function is defined. In essence the monitor in these states will wait for an event that matches one of the cases.

To illustrate the flexibility of an internal DSL compared to an external parser-based DSL, observe that the target state *AlarmOccurred(alarm, time)* is a function call. This means that the body of the function can be inserted in place of the call. The result is the following equivalent monitor:

```
class AlarmMonitor extends Monitor[Event] {
  val oneHour = 1000 * 60 * 60

  always {
    case AlarmOn(alarm, time) =>
      eventually {
        case AlarmOn(alarm2, _) if alarm != alarm2 => False
        case Repair(`alarm`, expert, time2) =>
          (alarm.reqQuali in expert.quali) && time2 - time <= oneHour
      }
  }
}
```

The resulting property resembles a temporal logic property, where the intermediate state is un-named. The monitor can be instantiated and applied to a trace as follows:

```
object Test extends Application {
  val alarm1 = Alarm("BrokenPipe", Mech)
  val alarm2 = Alarm("PowerFailure", Elec)
  val expert1 = Expert(Set(Elec, Mech))
  val expert2 = Expert(Set(Chem, Bio))

  val m = new AlarmMonitor

  val trace = List(
    AlarmOn(alarm1, 1000),
    AlarmOn(alarm1, 2000),
    Repair(alarm1, expert1, 5000)
  )

  m.verify(trace)
}
```

6 Discussion

SCALA is quite comparable to a specification language such as VDM++. This is due to the combination of object-oriented and functional programming, with convenient support for processing collections (sets, lists, and maps), together with strong support for defining internal DSLs. At the time of writing several interesting extensions of SCALA, such as macros and union types, are being considered. This movement will further empower the language. One can perhaps

make an argument for writing VDM^{++} and translate to JAVA, as argued in [14], but it is hard to make an argument for writing VDM^{++}, and translate to SCALA. The languages are too close in spirit, albeit not wrt. syntax.

Some of the issues identified in SCALA are the lack of proper support for design by contract. There is indeed support for pre- and post-conditions (*require* and *ensuring*), but there is no support for referring to the value of old variables in post conditions for functions with side-effects. Concerning invariants, [31] suggests a very elegant mechanism for defining such, as we have shown. It has the slight disadvantage that each function body must be wrapped in a special function (*atomic*), which executes the pre- and post-conditions respectively before and after the body's evaluation.

VDM^{++}'s predicate subtypes (a type defined as a subtype of another satisfying some predicate), related to invariants, also appear to be a very convenient concept, which could be adopted in a programming language. They relieve the programmer from scattering assertions throughout the program to check the predicates, and they support conceptual modeling. SCALA's support for defining union types consists of defining an abstract class or trait and then defining all alternative members as sub-classes. The notation for this is a little heavy-handed, as shown in the *TreeSum* specification on page 223. The similar type definition in RSL [17] looks as follows (similar to what it would look like in several functional programming languages such as SML, OCAML and HASKELL):

type Tree == Empty | Node(lt : Tree, nval : **Int**, rt : Tree)

A problem with this notation, however, is that it does not allow methods to be added to the alternatives as is possible when using sub-classes. There is a conflict between object-oriented programming and functional programming [37].

VDM's special syntax for sets, lists and maps is convenient, making them stand out in a specification. Along those lines, one disadvantage of SCALA's approach to define all constructs as library elements is that nothing stands out in a program. User-defined identifiers look like standard library identifiers. A solution to this could be to allow user-defined coloring schemes for library entities in SCALA IDEs.

The small implementation of a very expressive internal DSL for data parameterized state machines and temporal logic illustrates the augmented power that comes with a few language features for defining DSLs. Integration of visualization (static of program structure and dynamic of execution traces) as well as support for static analysis, model checking, proof support, and testing will further strengthen a language such as SCALA. VDM is an impressive specification language, that at its birth was decades ahead of its time. VDM^{++} was an interesting refinement adding object-oriented features. SCALA is an interesting new programming language, that brings the two worlds: specification and programming, closer together.

Acknowledgements Part of the research described in this publication was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Specifically, without the support from JPL's Laboratory for Reliable Software this work would not be possible. The research was also supported in part by AFOSR Grant FA9550-09-1-0481 and NSF Grant CCF-0926190. Thanks to Dines Bjørner and Cliff Jones for numerous discussions of VDM, and to Dines Bjørner for having engaged me in the RAISE project. Thanks to Martin Odersky for discussions of SCALA, and for giving an invited talk at RV 2010 (1st International Conference on Runtime Verification, Malta) on the subject of contracts in SCALA, offering some of the solutions mentioned in this paper. Thanks to Howard Barringer for our continued collaboration on runtime verification,

our discussions of VDM and SCALA, and for having commented on this paper.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittamplan, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*. ACM Press, 2005.

[2] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.

[3] Howard Barringer, Alex Groce, Klaus Havelund, and Margaret Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.

[4] Howard Barringer and Klaus Havelund. TraceContract: A Scala DSL for trace analysis. In *17th International Symposium on Formal Methods (FM'11), Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.

[5] Howard Barringer, Klaus Havelund, Elif Kurklu, and Robert Morris. Checking flight rules with TraceContract: Application of a Scala DSL for trace analysis. In *Scala Days 2011, Stanford University, California*, 2011.

[6] Howard Barringer, Klaus Havelund, David Rydeheard, and Alex Groce. Rule systems for runtime verification: A short tutorial. In *Proc. of the 9th Int. Workshop on Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 1–24. Springer, 2009.

[7] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125. Springer, 2007.

[8] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.

[9] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[10] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982. ISBN 0-13-880733-7.

[11] Grady Booch, Ivar Jacobson, and Jim Rumbaughr, editors. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[12] Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.

[13] Eiffel. `http://www.eiffel.com`.

[14] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

[15] Formal Methods Wikipedia. `http://en.wikipedia.org/wiki/Formal_methods`.

[16] Fortress. `http://java.net/projects/projectfortress`.

[17] Chris George, Peter Haff, Klaus Havelund, Anne Haxthausen, Robert Milne, Claus Bendix Nielsen, Soeren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England, 1992.

[18] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, 2005.

[19] Haskell. `http://www.haskell.org/haskellwiki/Haskell`.

[20] Klaus Havelund. *The Fork Calculus - Towards a Logic for Concurrent ML*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, Denmark, 1994.

[21] Klaus Havelund. RAISE in perspective. In Dines Bjørner and Martin Henson, editors, *Logics of Specification Languages*, pages 611–614. Springer, 2008.

[22] Klaus Havelund and Tom Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer, STTT*, 2(4), April 2000.

[23] Klaus Havelund and Willem Visser. Program model checking as a new trend. *STTT*, 4(1):8–20, 2002.

[24] JML. `http://www.eecs.ucf.edu/~leavens/JML`.

[25] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990. ISBN 0-13-880733-7.

[26] Cliff B. Jones and Roger C. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990. ISBN 0-13-880733-7.

[27] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[28] Cornelis Adam Middelburg. VVSL: A language for structured VDM specifications. *Formal Aspects of Computing*, 1(1):115–135, 1989.

[29] Robin Milner, Mads Tofte, and Robert Harper, editors. *The Definition of Standard ML*. MIT Press, 1997. ISBN 0-262-63181-4.

[30] OCaml. `http://caml.inria.fr/ocaml/index.en.html`.

[31] Martin Odersky. Contracts for Scala. In *Runtime Verification - First Int. Conference, RV'10, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *LNCS*, pages 51–57. Springer, 2010.

[32] David Pollak. Comment on Scala discussion forum.

[33] Python. `http://www.python.org`.

[34] RAISE project. `http://en.wikipedia.org/wiki/RAISE_specification_language`.

[35] Scala. `http://www.scala-lang.org`.

[36] Spec#. `http://research.microsoft.com/en-us/projects/specsharp`.

[37] Daniel Spiewak. Personal communication. March 2012.

[38] Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 109–124. Elsevier, 2006.

[39] Volker Stolz and Frank Huch. Runtime verification of concurrent Haskell programs. In *Proc. of the 4th Int. Workshop on Runtime Verification (RV'04)*, volume 113 of *ENTCS*, pages 201–216. Elsevier, 2005.

[40] VDM. `http://en.wikipedia.org/wiki/Vienna_Development_Method`.