Modeling with Scala[⋆]

Klaus Havelund and Rajeev Joshi

Jet Propulsion Laboratory, California Inst. of Technology, USA
{klaus.havelund,rajeev.joshi}@jpl.nasa.gov

Abstract. The activities and the associated formalisms for modeling and programming have many commonalities. In this paper we emphasize this point of view by modeling two examples in the programming language Scala, which have previously been modeled in the VDM specification language, and the Promela modeling language of the SPIN model checker respectively. The latter Scala model uses an internal DSL for hierarchical state machines, and a simple randomized testing framework exposing the same errors as found with SPIN. We believe, as the examples illustrate, that this use of a modern programming language for modeling is promising, especially if utilizing internal DSLs.

1 Introduction

Numerous formalisms exist for modeling systems before their development (prescriptive modeling) or as they exist (descriptive modeling). These formalisms have either a textual form or a graphical form, or both. Graphical formalisms can sometimes be grounded in a corresponding user friendly textual formalism, but not always, as is the case for example for UML and SysML in their standardized versions (Xml cannot be considered user friendly). Meanwhile, modern high-level programming languages have evolved in recent years with several features that make them suitable for modeling, especially if supported by visualization, as we argue in this paper, and illustrate with two examples. Similar arguments were presented in [4, 5]. One such high-level programming language is Scala [19], which combines object-oriented and functional programming. We present two formal modeling activities performed twenty years apart: one in 1979 using the Vdm specification language [3, 6], and the other in 1997 using the Promela modeling language of the Spin model checker [14]. We show how these models can be formalized in Scala with little impact on size or readability.

The Vdm specification is for a relational database model, formalized in [2] in a functional subset of Vdm. The modeling in Scala is almost one-to-one. For a more detailed comparison between Vdm and Scala we refer the reader to [8]. It is interesting to note that Vdm was originally used just for describing systems on paper, with no support for execution or even parsing and type checking.

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

The Promela model is that of the Remote Agent spacecraft controller, first formalized in Promela in [13]. This latter example is interesting in two respects. First, several errors were detected in the original Promela model, one of which later caused an actual deadlock in flight, and which also was detected in the Scala model using a simple testing approach. Second, the Scala model uses a Domain-Specific Language (DSL) - an internal Scala DSL - for modeling with Hierarchical State Machines (HSMs), as a larger effort to explore how such an HSM DSL can be used for modeling flight software for modern spacecraft.

The paper is organized as follows. Section 2 presents the relational database model in Vdm and Scala. Section 3 presents the Remote Agent model in Promela and Scala, and describes its testing. Finally, Section 4 concludes the paper.

2 Relational Databases

The example presented here is the concept of relational databases, as modeled in Vdm by Bjørner in [2], and reproduced here. We shall not illustrate how a database is updated, but will, as in [2], focus on how they are queried. This is not a description of how a database is implemented, but rather the definition from a user's abstract point of view. It is a conceptual model as argued by Bjørner in [2], meant to convey the concept of a relational database to a reader.

2.1 An Informal Description

A *relational data base* consists of an unordered collection of distinctly *named relations*. Each relation consists of an unordered collection of unnamed *tuples*. A tuple is a sequence of *fields*, the data of the database, identified by their position in the tuple[1]. All tuples of a particular relation are of the same length, and for a particular position, the fields in that position in all tuples of a relation are of the same type. The fields in tuples are of primitive type, such as integers, floating point numbers, Booleans, and strings.

The query language offers a collection of *commands*, all of which from the database will produce an unnamed relation, which is stored in a particular *working space*, which also can be referred to in the commands. Relations are referred to by their name, or if no name is provided, the reference is to the relation in the working space. The following four commands must be supported[2]. *Selection*: operates on a single relation (the origin relation) and delivers the relation containing all those tuples from the origin relation, whose field in a given position stands in a given relation to a given value. *Projection*: operates on a single relation, and delivers the relation of tuples each of which is a sub-tuple of a tuple in the origin relation, containing only selected fields. *Join*: operates on two relations. It forms the concatenation of those tuples from the two relations which in

[1] The paper [2] also presents a model where fields of a tuple are named by character strings, probably a more correct model.
[2] We have for space considerations omitted the division command from [2].

respective positions have fields which stand in a certain relation to each other. *Store*: stores the unnamed relation in the working space as a named relation. A *session* is a sequence of commands.

2.2 Relational Database Model in VDM

In the following, VDM specifications, Figures 1, 2, and 3, are shown in boxes with rounded corners, whereas SCALA programs, Figures 4, 5, and 6, are shown in boxes with square corners. The first set of VDM definitions are shown in Figure 1. These are the essential types of the specification[3]. The type RDB is that of a relational database, consisting (a cartesian product) of a set of relations RELS and a working space WS. The RDB type is defined as a constructed type using the :: symbol. The elements of this type are constructed by calls of an implicitly defined constructor mk−RDB(rels,ws), which in turn can be referred to in pattern matching. The type RELS represents the named relations as a mapping (function with finite domain) from relation names (type Rid) to relations of type REL. A workspace WS is a relation REL, which itself is a set of tuples TPL. A tuple is a finite non-empty list of fields of type Field. The type Field is the union of various primitive types, such as integers.

$$
\begin{array}{l}
\mathsf{RDB} :: \mathsf{RELS}\ \mathsf{WS} \\
\mathsf{RELS} = \mathsf{Rid} \xrightarrow{\mathbf{m}} \mathsf{REL} \\
\mathsf{WS} = \mathsf{REL} \\
\mathsf{REL} = \mathsf{TPL}\text{-}\mathbf{set} \\
\mathsf{TPL} = \mathsf{Field}^{+} \\
\mathsf{Field} = \mathsf{INTG} \mid \ldots
\end{array}
$$

Fig. 1: Domains in VDM.

$$
\begin{array}{l}
\mathsf{Cmd} = \mathsf{Sel} \mid \mathsf{Proj} \mid \mathsf{Join} \mid \mathsf{Sto} \\
\mathsf{Sel} :: [\mathsf{Rid}]\ N_1\ \mathsf{ROp}\ \mathsf{Field} \\
\mathsf{Proj} :: [\mathsf{Rid}]\ {N_1}^{+} \\
\mathsf{Join} :: ([\mathsf{Rid}]\ N_1)\ \mathsf{ROp}\ ([\mathsf{Rid}]\ N_1) \\
\mathsf{Sto} :: \mathsf{Rid} \\
\mathsf{ROp} = \underline{\mathsf{EQ}} \mid \underline{\mathsf{NEQ}} \\
\\
\mathsf{Sess} = \mathsf{Cmd}^{*}
\end{array}
$$

Fig. 2: Commands in VDM.

The next set of VDM definitions are shown in Figure 2, and model the commands that can be issued against a relational database. The Cmd type is defined as the union of a collection of constructed types, each representing a kind of command: selection, projection, join, and storing the workspace as a named relation. Each of these commands carry arguments.

The Sel command carries an optional relation identifier ($[\mathsf{Rid}] = \mathsf{Rid} \cup \{\,\mathbf{nil}\,\}$), a positive (non-zero) natural number, comparison operator of type ROp, and a field value. The semantics is: select those tuples from the relation where the field identified by the positive number is related to the field argument as indicated by the relational operator. When the relational identifier is **nil** the relation

[3] It is very common in VDM to approach a problem by starting defining such type definitions.

```
1   E−Sess(cl)(mk−RDB(rs,ws)) =
2     if cl = ⟨ ⟩
3       then mk−RDB(rs,ws)
4       else
5         (let (rs', ws') =
6           cases hd cl :
7             mk−Sel(r,i,o,f) →
8               (rs, {t | t ∈ ((r=nil) → ws, T → rs(r)) ∧ t[i] = f}),
9             mk−Pro(r,il) →
10              (rs, { ⟨ t[ il [i]] | 1 ≤ i ≤ len il ⟩ |
11                          t ∈ ((r=nil) → ws, T → rs(r))}),
12            mk−Join(($r_1$,$i_1$),o,($r_2$,$i_2$)) →
13              (rs, { $t_1 \frown t_2$ |
14                $t_1$ ∈ (($r_1$=nil) → ws, T → rs($r_1$)) ∧
15                $t_2$ ∈ (($r_2$=nil) → ws, T → rs($r_2$)) ∧
16                cases o : (EQ  → $t_1[i_1] = t_2[i_2]$,
17                           NEQ → $t_1[i_1] \neq t_2[i_2]$)}),
18            mk−Sto(r) →
19              (rs + [r ↦ ws], ws)
20        in
21          E−Sess(tl cl)(mk−RDB(rs',ws')))
22
23  type: Cmd* ⥲ (RDB ⥲ RDB)
```

Fig. 3: Evaluation function in VDM.

referred to is that in the unnamed working space. The relational operator type is the union of two singleton types containing the constants respectively EQ (for *equal*) and NEQ (for *not equal*). The format of the other commands should now be clear: projection maps a set of tuples in a relation to a set of tuples only containing a certain subset of columns indicated by a list of column numbers. A join combines tuples from two relations where one column value in one relation is related to another column value in the other relation in a certain way. Storing stores the working space as a named relation. Finally, a session Sess is a (possibly empty) list of commands.

Finally, the recursively defined function E−Sess in Figure 3 interprets a session on a relational database, resulting in a new relational database, as indicated by the type of the function at the bottom of the figure ($A \overset{\sim}{\rightarrow} B$ is the set of partial functions from A to B). We shall not go into much details of this function definition, except for some notation explanation: line 8 contains a set comprehension of tuples t which belong to either the working space if the relation id is **nil** or to the relation denoted by the relation id: rs(r), and which satisfy the equation[4]: t[i] = f. The set comprehension in lines 10-11 is the set of tuples, each of which itself is generated by a tuple comprehension expression projecting

[4] This is actually wrong as will be discussed later.

to only those tuple elements identified by column ids in the list il . The term $\ldots + [\ \ldots \mapsto \ldots]$ in line 19 is a finite map update. This definition is written in a functional style. However, VDM also supports an imperative style where expressions can have side effects.

2.3 Relational Database Model in Scala

The SCALA model (a program) of the relational database concept is shown in Figures 4 (types), 5 (commands), and 6 (applicative evaluation function). We have written the functions as closely as reasonable to the VDM version, except that we introduced a couple of auxiliary functions for looking up a relation id and for comparing fields. Commands in Figure 5 are defined using class inheritance, and using so-called **case** classes that allows for pattern matching over objects of the classes. We see in Figure 6 the **for**−**yield** construct, the general form of which is:

for $(x_1 \leftarrow exp_1; \ldots; x_n \leftarrow exp_n$ **if** $\mathsf{p}(x_1, \ldots, x_n)$ **yield** $\mathsf{f}(x_1, \ldots, x_n)$

This construct is used to model the set and list comprehensions in the VDM specification. It denotes a collection of values. The expressions $exp_1, \ldots, exp_n$ must themselves denote collections (lists, sets, maps, ...). For each $x_1 \in exp_1$, $x_2 \in exp_2$, etc, where the Boolean expression $\mathsf{p}(x_1, \ldots, x_n)$ is true, the value denoted by the expression $\mathsf{f}(x_1, \ldots, x_n)$ is added to the resulting collection. The type of the resulting collection is that of the first expression exp_1. So if it is a list, the result will be a list, if a set, the result will be a set. For example, if exp_1 is a set, it corresponds to VDM's set comprehension:

$$\{\ \mathsf{f}(x_1, \ldots, x_n) \mid x_1 \in exp_1 \wedge \ldots \wedge x_n \in exp_n \wedge \mathsf{p}(x_1, \ldots, x_n)\ \}$$

```
type Rid
type RDB = (RELS, WS)
type RELS = Map[Rid, REL]
type WS = REL
type REL = Set[TPL]
type TPL = List[Field]
type Field = Any
```

Fig. 4: Domains in Scala.

It should be clear that the VDM model and the SCALA program are very similar. One might prefer the more mathematical notation of the VDM specification. The topic of displaying programs with a mathematical appearance was addressed elegantly in the FORTRESS programming language [7]. The VDM specification in [2] was written before the appearance of syntax and type checkers for VDM. We did indeed find six errors, including two syntax errors, and four type checking/static

```
 trait Cmd
case class Sel(rid: Option[Rid], fieldNr: N1, o: ROp, field: Field) extends Cmd
case class Proj(rid: Option[Rid], fieldNrs: List1[N1]) extends Cmd
case class Join(l: (Option[Rid], N1), o: ROp, r: (Option[Rid], N1)) extends Cmd
case class Sto(rid: Rid) extends Cmd

 trait ROp
case object EQ extends ROp
case object NEQ extends ROp
```

Fig. 5: Commands in Scala.

```
1  def E_Sess(cl: List[Cmd])(rdb: RDB): RDB = {
2    val (rs, ws) = rdb
3    if (cl == Nil) rdb
4    else {
5      val (rs_, ws_) =
6        cl.head match {
7          case Sel(r, i, o, f) ⇒
8            (rs, for (t ← getRel(r, rdb) if comp(o)(t(i),f)) yield t)
9          case Proj(r, il) ⇒
10           (rs, for (t ← getRel(r, rdb)) yield (for (i ← il) yield t(i)))
11         case Join((r1, i1), o, (r2, i2)) ⇒
12           (rs,
13             for (t1 ← getRel(r1, rdb); t2 ← getRel(r2, rdb)
14                  if comp(o)(t1(i1),t2(i2))) yield (t1 ++ t2)
15           )
16         case Sto(rid) ⇒
17           (rs + (rid → ws), ws)
18       }
19     E_Sess(cl.tail)(rs_, ws_)
20   }
21 }
```

Fig. 6: Evaluation function in Scala.

analysis errors. All of these were fixed in our presentation of the VDM model, except for line 8 in Figure 3, where the test t[i] = f ignores the operation o in the pattern mk−Sel(r,i,o,f). This is changed in the SCALA version.

One interesting observation, however, would not be highlighted by a syntax or type checker, static analyzer, or theorem prover. This concerns the Join command, which simply concatenates the tuples from each relation, see line 13 in Figure 3 and line 14 in Figure 6. This means that common columns are duplicated, which is not the standard "natural join" operator[5], where such columns

[5] https://en.wikipedia.org/wiki/Relational_algebra.

are merged into one. We noticed this discrepancy only upon executing the model. It may be a minor issue, but it illustrates how executable models can reveal otherwise undiscovered properties.

3 The Remote Agent

In this section we shall demonstrate a Scala model of a space software module previously (in 1997) modeled in the Promela language of the Spin model checker [14]. Spin supports verification of finite state asynchronous process systems communicating via message passing and/or shared variables. Models are formulated in the Promela language, which has similarities to a simple programming language, although without much support for regular programming with data structures. Efficient verification has been given priority over convenient modeling language features in some cases. Properties to be verified are stated as assertions in the model, or in the linear temporal logic Ltl. The Spin *model checker* automatically determines whether a model satisfies a property, and generates a counter example in the form of an error trace if this is not true.

The particular system being modeled is the multi–threaded *plan execution module* of the *Remote Agent* (RA) [16], which itself was programmed in Lisp. The Remote Agent was an artificial intelligence based spacecraft control system architecture. In addition to the plan execution module modeled in this section, it also contained a *planning module*, which generated plans based on goals received from Earth, sending these plans to the plan execution module for execution. A third module, the *mode identification and recovery module*, constantly monitored the state of the spacecraft and would attempt to recover in case of anomalies. The Remote Agent was one of 12 technologies tested on the Deep-Space 1 (DS-1) spacecraft, launched October 1998. The Remote Agent itself was initiated during May 1999, demonstrating the complete control of a spacecraft by artificial intelligence based software for the first time in NASA's history.

The plan execution module is a classic multi-threaded application vulnerable to classic concurrency errors such as data races and deadlocks. The Spin effort, described in [13], consisted of hand translating parts of the plan execution Lisp code into a model in the Promela language of Spin, and then verifying two properties formulated by the Remote Agent programmers. Both properties turned out to be broken in the model, revealing a total of 4 errors. The effort was at the time considered very successful before flight. It, however, further gained reputation since one of the errors, after having been fixed in the code, was later re-introduced in a different part of the plan execution module by a different programmer through a copy-and-paste operation, but without copying the fix (a critical section). This caused a data race that lead to a deadlock during flight. Because of the deadlock, thrusting did not turn off when required, and the spacecraft was unable to recover by itself. The craft was put in stand-by mode by the ground crew until a repair was made.

3.1 The Remote Agent in Promela

In this section we shall present the Remote Agent plan execution module in more (although not full) detail, as well as its modeling in PROMELA. The full description of the Remote Agent can be found in [13]. The Remote Agent Executive, Figure 7, supports execution of *tasks*. A task may be, for example, to operate a camera. A task often requires that specific *properties* hold during its execution. For example, the camera-operating task may require the camera to be turned on throughout task execution. When a task is started, it first tries to *achieve* the properties on which it depends, whereafter it starts performing its main function. E.g. the camera-operating task may try to turn on the camera before running the camera. Properties may, however, be unexpectedly broken (e.g., the camera may turn off) and tasks depending on such broken properties must then be informed about this (aborted).

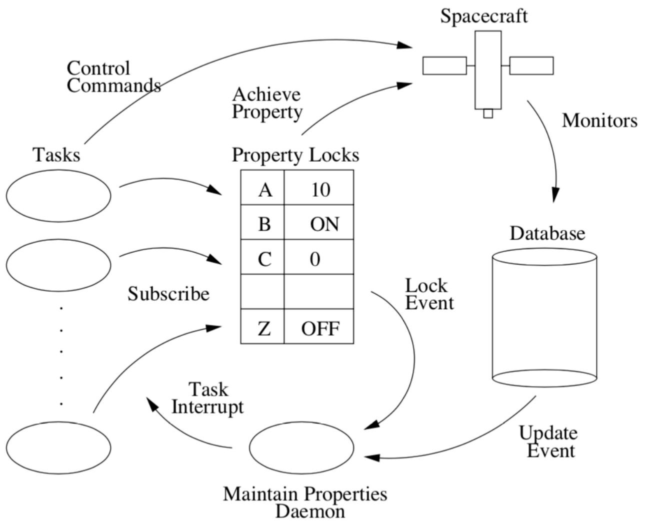


Fig. 7: The Remote Agent executive.

The Database The state of the spacecraft at any particular point can be considered as an assignment of values to a set of variables, each corresponding to a component sensor on board the spacecraft. As an example, the variable CAMERA may have one of the values ON or OFF. A particular assignment of a value to a variable is called a *property*, where the variable is called the *property name* and the value is called the *property value*. The actual state of the spacecraft is constantly monitored, and stored in a *database*.

The Lock Table When a task requires a certain property to hold, e.g. (RADIO, ON), it adds the property in a *lock table*. During this locking, other tasks with incompatible properties, e.g. (RADIO, OFF), cannot execute. Two properties are incompatible if they require different observed values of the same variable. The lock table in addition for each locked property stores which tasks rely on it (there can be multiple), and in addition contains a flag *achieved*, which is set to true when the property has been achieved to hold in the database.

The Daemon Executing concurrently with the tasks is a *property maintenance daemon* that monitors the lock table and the database. If there is an *inconsistency* between the database and the locks – meaning that a locked property no longer holds in the database while the *achieved* flag is true – the daemon aborts all tasks *subscribing* to the property, and subsequently it tries to re-achieve the property (if a task has not already done so). The daemon remains inactive unless certain *events* occur, such as a change of the database (a DB_EVENT) or lock table (a LOCK_EVENT). Once awakened, the daemon examines all locks in the property lock table. For each lock where the achieved field is *true*, it checks whether the property holds in the database. If this is not the case, all tasks in the lock's subscribers list are aborted and a recovery procedure is initiated to re-achieve the property. After examining all locks, the daemon goes into sleep mode again by waiting for another DB_EVENT or LOCK_EVENT event.

The Tasks Before a task executes its main job, it will try to achieve the properties that the execution depends on. The first step is to lock the properties in the lock table. Locking a property will only succeed if it is compatible with the existing locks; otherwise, the task aborts. If there are no conflicting locks and the lock does not already exist, the task will create it. Note that some other task may have already locked the same property, which is not defined as a conflict. If it succeeds, the task also puts itself into the subscribers list of the lock, indicating that the task depends on this property.

The creator task of a lock is called the *owner*, in contrast to tasks that subscribe later to the same property. The owner is responsible for achieving the property, resulting in the database being updated. Upon successful achievement, the achieved-field in the lock is set to *true* by the task. If the achievement fails, the task aborts. Other tasks that subscribe later than the owner must wait for the owner to achieve the property. This is done by simply waiting for a DB_EVENT, and the indication that the property was successfully achieved.

Once a task has first locked and then achieved its required properties, it executes its main job, relying on the properties to be maintained throughout job execution. Before a task terminates, it releases its locks. That is, it removes itself from the subscribers list, and if the list then becomes empty (no other subscribers), it removes the lock completely. In case there are other subscribers, the lock must of course be maintained.

Modeling in Promela The modeling in PROMELA required some ingenuity due to the lack of modern programming language concepts. E.g. PROMELA's concept of asynchronous communication channels (introduced with keyword **chan**) was used to model lists, the basic data structure of LISP. All communication between processes in this model takes place via *shared variables*, reflecting the LISP implementation. Figure 8 shows the top level task process and Figure 9 shows the top level daemon process. Each of these in turn call functions that perform further operations. The PROMELA code in total is 332 uncommented lines of code. The LISP module it was modeled after was 3000 lines of code. The model only deals with a limited number of tasks and properties in order to limit the search space the SPIN model checker has to explore. Abstractions were made in an informal manner.

A task (Figure 8) will want to achieve a property p before executing its main task, here named closure. Before that, it locks the property in the lock table. During execution of the closure, it may get aborted in case the daemon discovers an inconsistency between the lock and the database. The task is thrown to the program point in line 10, whereafter it will release its lock.

```
 1  proctype Achieving_Task(TaskId this)
 2  { Property p;
 3    ...
 4    bool err = 0;
 5    {
 6       lock_property (this, p, err);
 7       achieve_lock_property (this, p, err);
 8       closure ()
 9    } unless {err || active_tasks [this].state == ABORTED};
10    active_tasks [this].state = TERMINATED;
11    { release_lock (this, p)} unless { active_tasks [this].state == ABORTED}
12  }
```

Fig. 8: Promela model of tasks.

The daemon (Figure 9) will initially (first_time == **true**) check all locks and then (line 17) wait for the database or lock table to be updated. Upon being awakened, it will first check all locks and interrupt tasks subscribing to violated locks (line 6). The daemon maintains a counter event_count holding the value of the sums of two counters being increased when respectively the database is updated and the lock table is updated. It will execute a conditional statement (lines 11-18) which will repeat the procedure if these counters have changed, otherwise the daemon will wait for new changes.

The Properties to be Verified The PROMELA model was verified against the following two properties (delivered to us after the model had been created):

```
proctype Daemon(TaskId this) {
  bit  lock_violation ;
  byte event_count = 0;
  bit  first_time  = true;
  do
  :: check_locks( lock_violation );
     if
     ::  lock_violation  → do_automatic_recovery()
     :: else
     fi ;
     if
     :: (! first_time  &&
          Ev[DB_EVENT].count + Ev[LOCK_EVENT].count ≠event_count ) →
            event_count = Ev[DB_EVENT].count + Ev[LOCK_EVENT].count
     :: else →
           first_time  = false;
           wait_for_events ( this ,  DB_EVENT,LOCK_EVENT)
     fi
  od
};
```

Fig. 9: Promela model of the daemon.

Release property: *A task releases all of its locks before it terminates.*

Abort property: *If an inconsistency occurs between the database and an entry in the lock table, then all tasks that rely on the lock will be terminated, either by themselves or by the daemon.*

The Release property was stated as an assertion after line 11 in Figure 8. The Abort property was stated as the Linear Temporal Logic (LTL) formula, focusing on just one of the tasks (task 1):

```
[](task1_property_broken -> <>task1_terminated).
```

The definitions of the terms `task1_property_broken` and `task1_terminated` are not shown here, see [13] for details. The property states that it is always (`[]`) the case that if task 1's property is broken, then eventually (`<>`) task 1 is terminated.

The model, consisting of two tasks, each attempting to lock the same variable but with different conflicting values, a daemon, and an environment that randomly can damage a database entry, was verified exhaustively by Spin. It turned out that both properties were violated in the model as well as in the Lisp code according to the programmer. The error numbering below follows the numbering in [13].

– *Error 1* : The Release property was violated since the task in Figure 8 may not only get aborted by the daemon during its main task, but also during releasing the lock, which causes the lock releasing to be abandoned. This error is somewhat obvious from the code, but was not detected by the programmer, nor by us during modeling since we did not know the properties at that point.

– *Error 2* : The ABORT property was violated since the daemon in Figure 9 in line 15 makes the decision to wait for new events if the event counters have not changed. However, if the environment corrupts the database in between this decision has been taken and the actual wait in line 17, the daemon will not wake up to detect the damage.
– *Error 3* : The ABORT property was violated since the daemon in Figure 9 in the function check_locks contains two pieces of sequentially composed code: one where tasks depending on violated properties are aborted, and one where the daemon repairs the database. In case the environment damages the database in between these two sections of code, the tasks will not get aborted.
– *Error 4* : The ABORT property was violated since when a task achieves a property and then subsequently sets the *achieved* flag to true, the environment may damage the database in between these two statements, and hence the daemon may not detect the damage since the *achieved* flag his false (a lock is only defined as violated if this flag is true).

All these errors were classical concurrency errors where the environment damages the database in between two sections of code not protected by a critical section. The pattern of Error number 2 was the one causing a deadlock during flight in a sibling module to where the code was copied without copying the fix.

3.2 The Remote Agent in Scala

In this section, we show our model of the Remote Agent using a SCALA DSL for Hierarchical State Machines (HSMs), using the same naming conventions as in [13]. HSMs [18] are an extension of traditional state machines: they allow declaration of mutable state variables (which can be used in transition guards, and updated in transition actions), hierarchical nesting of states (a child state inherits all transitions from its parent, but can override any subset), entry and exit actions (which are triggered when control enters or leaves a state), and support for orthogonal regions (multiple child states executing in parallel - not currently supported by the DSL). Figure 10 shows a graphical depiction of the HSM for a Remote Agent task, automatically generated from the SCALA program using the SCALAMETA [20] and PLANTUML [17] frameworks. Following standard notation, the filled out black circles indicate the initial child substate that is entered whenever a parent state is entered. Associated with each state are also optional code fragments called the *entry* and *exit* actions, which are executed whenever the HSM transitions into or out of a state respectively. A transition between two states is shown using a labeled arrow, where the label is of the form EVENT if guard / code, which indicates that the HSM reacts to the given EVENT only if the given guard holds, and then it executes the given code and transitions

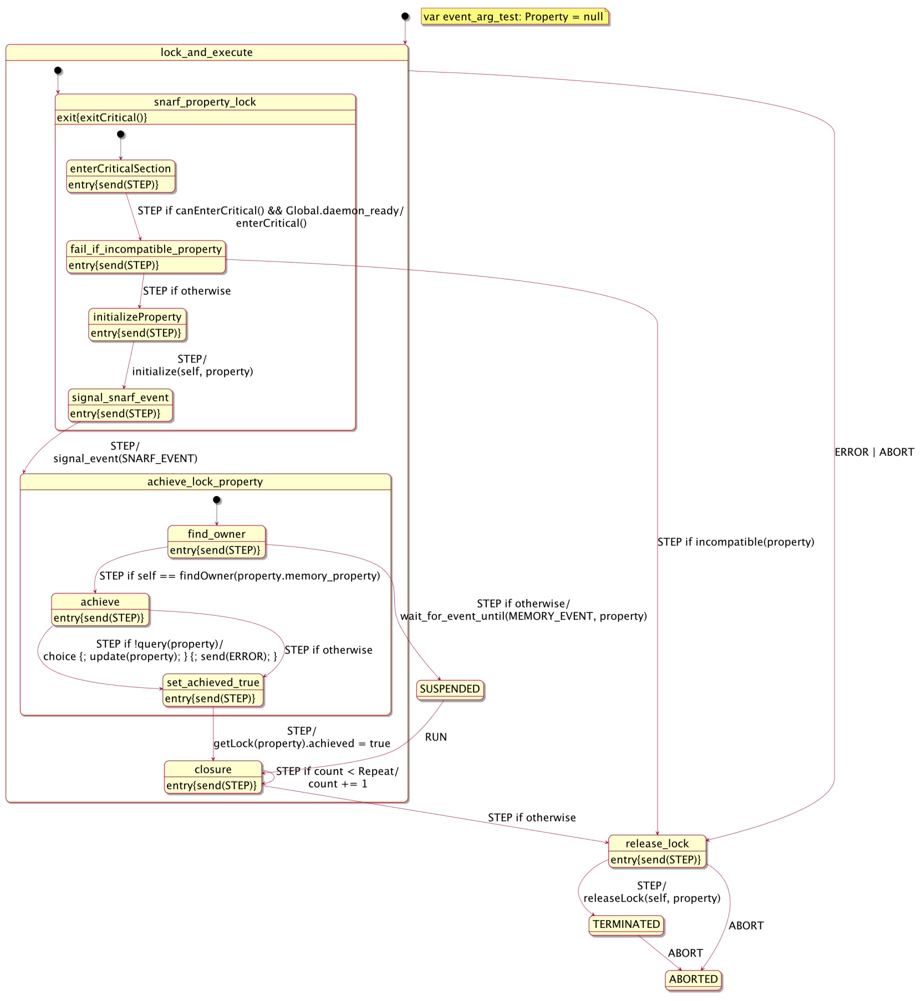
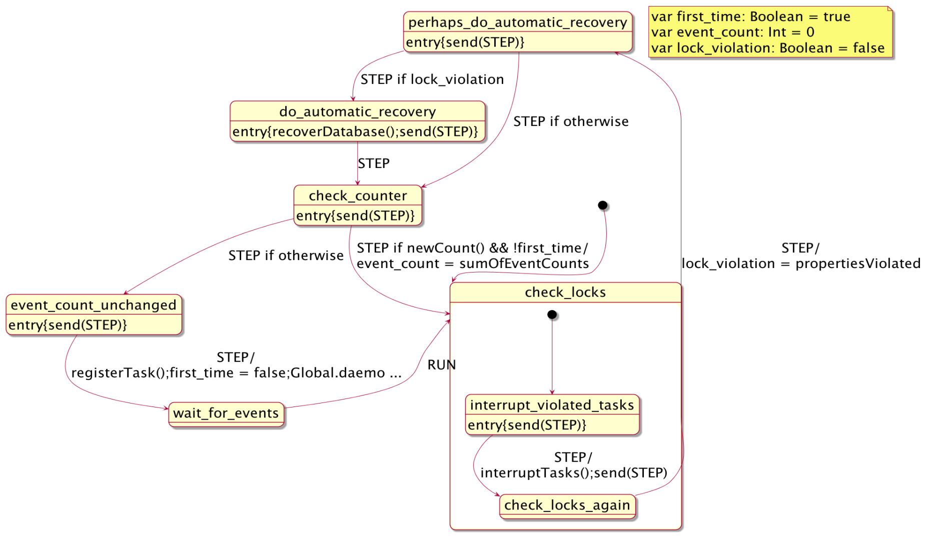


Fig. 10: Task HSM.

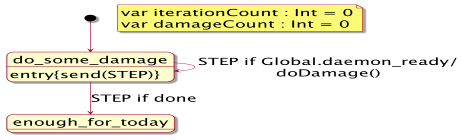
to the target state. The hierarchical nature of the state machine means that a child state inherits transitions from its parent state, unless it explicitly overrides the transition. For instance, all substates of the lock_and_execute state inherit the transitions on ABORT and ERROR events that take the HSM into the state release_lock. Thus, when a remote agent task is first started, it starts in the state enterCriticalSection, and executes the entry action send(STEP) which sends the STEP event to itself[6]. In state enterCriticalSection, the HSM responds to a STEP event only if the associated guard condition is true. When this condition

[6] The idiom of an HSM sending an event to itself is commonly used in HSMs to implement sequential behavior, and is used extensively in our modeling. Breaking

is true, and the HSM is scheduled for execution, it executes the enterCritical() code block, and transitions to the state fail_if_incompatible_property.


perhaps_do_automatic_recovery
entry{send(STEP)}
var first_time: Boolean = true
var event_count: Int = 0
var lock_violation: Boolean = false
STEP if lock_violation
do_automatic_recovery
entry{recoverDatabase();send(STEP)}
STEP if otherwise
STEP
check_counter
entry{send(STEP)}
STEP if otherwise
STEP if newCount() && !first_time/
event_count = sumOfEventCounts
STEP/
lock_violation = propertiesViolated
check_locks
event_count_unchanged
entry{send(STEP)}
STEP/
registerTask();first_time = false;Global.daemo ...
RUN
wait_for_events
interrupt_violated_tasks
entry{send(STEP)}
STEP/
interruptTasks();send(STEP)
check_locks_again

(a) Daemon


var iterationCount : Int = 0
var damageCount : Int = 0
do_some_damage
entry{send(STEP)}
STEP if Global.daemon_ready/
doDamage()
STEP if done
enough_for_today

(b) Environment

Fig. 11: Daemon and environment HSMs.

Figure 11 shows the HSMs for the daemon and the environment (which can inflict damage by corrupting the database). An attractive feature of modeling HSMs in Scala is that transition guards and actions can be written using a full-fledged programming language. For instance, in state check_counter, the daemon calls the Scala function newCount() to test if the sum of the counts of updates to the database and the lock table differs from its own local count. If they are different, it updates its event count and transitions to check_locks; but if they are the same, it transitions to event_count_unchanged, and subsequently waits in the next step. As we describe later (see Section 3.3), this logic is flawed as the daemon can miss a race condition that can lead to database corruption.

up a sequence of steps using this idiom allows us to check system executions where task behaviors are interleaved with each other.

Figure 12 shows how the task state machine is expressed in our SCALA HSM DSL (in space saving format). Each HSM kind is implemented as a class, which extends the RaHSM class, which itself extends the HSM trait defined in [12]. The RaHSM class adds e.g. prioritized event queues. The Task class is parameterized with a monitor, which checks properties of interest as the system is running. Each state in the HSM is a SCALA object that extends the underlying state class, optionally passing it the name of its superstate (if any), and an optional Boolean value true if it is the initial state of the superstate. Entry and exit blocks are defined as shown, e.g. lines 11 and 8, as calls of functions that each can take an arbitrary block of SCALA code (call-by-name) that is executed on entry or exit. HSM transitions are defined using the when construct, with each event transition defined using SCALA pattern matching. Each transition has the form when E **if** grd $\Rightarrow$ S exec code , which denotes a transition that executes on receiving event E if condition grd is true, and then the given code is executed, and the HSM changes to state S. For instance lines 18 and 19 show two HSM transitions on the STEP event, depending on whether a guard condition is true or not. The form '**if** otherwise', meaning negation of all other guards, makes diagrams more readable. The use of pattern matching allows compact representation of many transitions where different events have the same target state and the same behavior, see for instance line 5 where the ERROR and ABORT events are handled in one statement.

The SCALA model is 556 lines of uncommented code (assuming all definitions are in one file as in the PROMELA case), compared to the 332 lines of uncommented PROMELA model and the 3000 lines of original LISP code. The additional code compared to the PROMELA model reflects a more detailed and natural programming of the data structures, such as the lock table, and the modeling of the Remote Agent processes as HSMs is somewhat verbose.

3.3 Monitoring and Randomized Scheduling

In this section we describe how the SCALA model is tested.

Monitoring The properties to be monitored are formulated in the DAUT SCALA DSL [9, 10], supporting a formalism combining temporal logic and programming. DAUT is a part of an effort defining monitoring DSLs in SCALA, including also the works described in [1, 11, 15]. A DAUT monitor is a SCALA class extending the class Monitor[E], parameterized with the type of events E it can monitor. The Monitor class offers a method verify (event: E), which updates the monitor for a new event, issuing an error if the monitor is violated on a safety property, and a method end() terminating the monitor, and issuing an error if a liveness property is violated (some event did not occur that should have occurred). The events we shall monitor here are called EVRs (EVent Reports), and are defined by the type definitions in Figure 13. There are four kinds of EVRs, all subclassing the EVR trait. There is an EVR reporting an HSM entering a state, exiting a state, sending a message to another HSM, and finally an action reporting any other kind of event.

```
class Task(...) extends RaHSM(monitor)
{  initial (lock_and_execute)

   object lock_and_execute extends state() {
      when { case ERROR | ABORT ⇒ release_lock } }

   object snarf_property_lock extends state(lock_and_execute, true) {
      exit {  exitCritical ()  } }

   object enterCriticalSection  extends state(snarf_property_lock,  true)  {
      entry { send(STEP) }
      when {
         case STEP if canEnterCritical() && Global.daemon_ready ⇒
            fail_if_incompatible_property   exec {  enterCritical ()  } } }

   object  fail_if_incompatible_property   extends state(snarf_property_lock) {
      entry { send(STEP) }
      when {case STEP if incompatible(property) ⇒ release_lock
             case STEP if otherwise ⇒ initializeProperty  } }

   object  initializeProperty   extends state(snarf_property_lock) {
      entry { send(STEP) }
      when {case STEP ⇒ signal_snarf_event exec {initialize( self ,property)}}}

   object  signal_snarf_event   extends state(snarf_property_lock) {
      entry { send(STEP) }
      when {case STEP ⇒ achieve_lock_property exec {signal_event(LOCK_EVENT)}}}

   object achieve_lock_property extends state(lock_and_execute)

   object find_owner extends state(achieve_lock_property, true) {
      entry { send(STEP) }
      when {case STEP if self == findOwner(property.memory_property) ⇒ achieve
             case STEP if otherwise ⇒ SUSPENDED exec {waitfor_event(DB_EVENT,property)}}}

   object achieve extends state(achieve_lock_property) {
      entry { send(STEP) }
      when {case STEP if !query(property) ⇒ setachieved_true exec {
                choice  { update(property) } { send(ERROR) }}
             case STEP if otherwise ⇒ setachieved_true  }}

   object setachieved_true  extends state(achieve_lock_property) {
      entry { send(STEP) }
      when {case STEP ⇒ closure exec {getLock(property).achieved = true}}}

   object closure extends state(lock_and_execute) {
      val Repeat = 2; var count: Int = 0
      entry {send(STEP)}
      when {case STEP if count < Repeat ⇒ closure exec {count += 1}
             case STEP if otherwise ⇒ release_lock}}

   object release_lock  extends state() {
      entry { send(STEP) }
      when {case ABORT ⇒ ABORTED
             case STEP ⇒ TERMINATED exec { releaseLock(self, property) }}}

   object ABORTED extends state()

   object TERMINATED extends state() {
      when { case ABORT ⇒ ABORTED }}

   object SUSPENDED extends state() {
      when { case RUN ⇒ closure }}
}
```

Fig. 12: The task HSM in Scala.

```
trait EVR
case class EnterState(task: RaHSM, state: String) extends EVR
case class ExitState(task: RaHSM, state: String) extends EVR
case class Message(sen: RaHSM, event: EventId, rec: RaHSM) extends EVR
case class Action(name: Any*) extends EVR
```

Fig. 13: The monitored event reports.

We define two monitors, shown in Figure 14. Class RaMonitor defines some utility functions specific for the Remote Agent scenario. The ReleaseMonitor implements the RELEASE property from Section 3.1. It reads: *"it is always the case, that when an EnterState(task, state) is observed, i.e. the task enters state, and this state is either the TERMINATED state or the ABORTED state, then the task has released the property it was locking (a function call on the task)"*. As can be seen, this is a temporal logic formula of the form $\Box(e \rightarrow p)$, where e is an event and p is a state predicate.

The AbortMonitor implements the ABORT property. A variable abortedTasks is introduced to store all tasks that have received an ABORT message, and is updated upon each observed ABORT message (the first **case** statement). The property (the second **case** statement) then reads: *"it is always the case, that when an Action(DB_EVENT) or an Action(LOCK_EVENT) is observed, i.e. either the database or the lock table is modified, then for each violated task (i.e. relying on a property that has been broken in the database), for which we have not in the past observed a Message(_, ABORT, target), where target denotes that task, we must observe such an abort message in the future"*. This is effectively the equivalent to a temporal logic formula of the form: $\Box(e_1 \rightarrow \forall t \in S\ .\ (\neg \blacklozenge e_2(t) \ \rightarrow \ \Diamond e_2(t)))$, where e_1 is an event and e_2 is an event parameterized with a task t, universally quantified over the finite set S (the violated tasks). Here $\blacklozenge e_2(t)$ means that $e_1(t)$ occurred in the past, and $\Diamond e_2(t)$ means that $e_2(t)$ will occur in the future. In DAUT a *hot* state must match a future event, otherwise the end() method will issue an error. The SCALA construct **for** (t ← S **if** P(t)) **yield** hot {...} results in a list containing a hot state for each t in S for which $P(t)$ holds. As can be seen, DAUT does not directly support past time logic, thus requiring the auxiliary variable abortedTasks.

Randomized Scheduling The four HSMs (two tasks, the daemon, and the environment) execute by sending each other messages, including the STEP messages to themselves. A scheduler will in each step have to pick a state machine which is enabled (e.g. there are messages in its queue) and execute the state machine on the first message in the queue. We shall use the SCALA concept of an *iterator* to select enabled machines to execute. The MachineSelector class in Figure 15 extends Iterable [RemoteAgentHSM], which requires us to define an iterator method, which in turn returns an Iterator [RemoteAgentHSM] object, which itself defines the hasNext and next methods. Each call of next(), assum-

```
class RaMonitor extends Monitor[EVR] { ...}

class ReleaseMonitor extends RaMonitor {
  always {
    case EnterState(task, state) if state == "TERMINATED" ||
                                    state == "ABORTED" ⇒
      task.asInstanceOf[Task].hasReleasedProperty()
  }
}

class AbortMonitor extends RaMonitor {
  var abortedTasks: Set[RaHSM] = Set()

  always {
    case Message(_, ABORT, target) ⇒
      abortedTasks += target
    case Action(DB_EVENT) | Action(LOCK_EVENT) ⇒
      for (task ← violatedTasks() if !abortedTasks.contains(task)) yield hot {
        case Message(_, ABORT, target) if target.hsmName == task.hsmName ⇒ ok
      }
  }
}
```

Fig. 14: Monitors for the Release and Abort properties.

ing that hasNext is true, returns an enabled machine that can execute one step. SCALA's (effectively JAVA's) random number generator is used to pick a machine randomly. In addition, a user-defined function canRun parameterized with a machine and all the machines, can be used to select which machine to run. This function is useful for steering the scheduler around errors already detected, in order to detect different errors, as illustrated in the next subsection.

Given an instance of the MachineSelector, we can iterate over the machines it generates, calling the run() method on each machine making it potentially perform one transition, as shown in the scheduler in Figure 16. This class resets and re-executes the state machines from their initial state numerous times, defined by the upper limit maxResets provided as parameter to the class. The second class parameter is the function reset, which is user defined, and which generates a new list of machines and a new instance of the monitor (so we can call the method end() on it). The third parameter is the user defined canRun method used to control the machine selection.

Test Results The monitors were executed with a flag causing them to terminate on the first error encountered. For each error an error trace is produced. For example for error number 2, the error trace in Figure 17 is produced after 1.1 seconds on reset number 2,453 of the HSMs to their initial state. It shows that

```
class MachineSelector(
    machines: List [RemoteAgentHSM],
    canRun: (RemoteAgentHSM, List[RemoteAgentHSM]) ⇒ Boolean)
  extends Iterable [RemoteAgentHSM]
{
  val random = new scala.util.Random

  override def iterator : Iterator [RemoteAgentHSM] =
    new Iterator [RemoteAgentHSM] {
      def isEnabled(machine: RemoteAgentHSM): Boolean = {
        machine.enabled() && canRun(machine, machines) && ...
      }

      override def hasNext: Boolean = machines.exists(isEnabled(_))

      override def next(): RemoteAgentHSM = {
        val enabledMachines = machines.filter(isEnabled(_))
        val nextMachine = random.nextInt(enabledMachines.size)
        enabledMachines(nextMachine)
      }
    }
}
```

Fig. 15: The machine selector.

```
class Scheduler(
  maxResets: Int,
  reset : () ⇒ (List [RemoteAgentHSM], Monitor[EVR]),
  canRun: (RemoteAgentHSM, List[RemoteAgentHSM]) ⇒ Boolean)
{
  def run(): Unit = {
    for (i ← 0 until maxResets) {
      val (machines, monitor) = reset()
      val generator = new MachineSelector(machines, canRun)
      for (machine ← generator) { machine.run() }
      monitor.end()
    }
  }
}
```

Fig. 16: The scheduler.

task 1 first (step 12) locks the property and achieves it in the database. Then the daemon (step 14) wakes up. The daemon later (steps 24 and 25) determines that there are no new events, and it gets ready to wait for new events. The environment (step 26) then destroys the database. Finally (step 29) the daemon goes to sleep, as a result of the previous counter check, and therefore misses the database corruption that happened in between.

```
...
12. Task1  : achieve -STEP-> set_achieved_true
...
14. Daemon : wait_for_events -RUN-> interrupt_violated_tasks
...
24. Daemon : perhaps_do_automatic_recovery -STEP-> check_counter
25. Daemon : check_counter -STEP-> event_count_unchanged
26. Env    : do_some_damage -STEP-> do_some_damage
...
29. Daemon : event_count_unchanged -STEP-> wait_for_events
```

Fig. 17: Extract from counter example for error 2 containing 29 transitions, detected after 1.1 seconds on reset number 2,453.

We ran the test harness with one property at a time. For each found error, we had to route the scheduler around that error, in order to find the next one (as an alternative to fixing the errors which we did not). This was done by defining the canRun functions (see Figure 15). For example, the function avoidError2 in Figure 18 defines a thread as scheduable if either it is not the environment, or if it is, the daemon is not in state event_count_unchanged, where it has taken the *decision* to wait but not waited yet. All four errors mentioned in Section 3.1 were

```
def avoidError2(machine: RaHSM, machines: List[RatHSM]): Boolean = {
  !machine.isInstanceOf[Env] || {
    val daemon = machines.find { case machine ⇒
        machine.isInstanceOf[Daemon]
      }.get.asInstanceOf[Daemon]
    !daemon.inThisState("eventcountunchanged")
  }
}
```

Fig. 18: User defined scheduler control method avoiding error 2.

detected using the monitors and the randomized scheduling. The approach was also useful in getting the Scala model and monitors correct. Figure 19 shows the data for the different verifications, comparing the verification performed with Spin in [13] in 1997. The testing of the Scala model was performed on a MacBook Pro 15 inch laptop running macOS Version 10.13.5, with a 2.8 GHz Intel Core i7, and 16GB of memory. A Sun workstation was used for the Spin verification.

For each of the four errors, we indicate the property violated, the number of states explored by Spin, the memory consumption in Mb used by Spin, the time spent by Spin (seconds), the number of resets of the Scala model (and average over 10 runs in parentheses), and the time spent testing the Scala model (and average over 10 runs in parentheses). The last two rows show data for a "correct" model, i.e. after fixing errors and/or proper re-routing the scheduler around errors 1-4. Here the Promela model was proved correct by Spin. For the Scala model, no further errors were found during 1 million resets. This does of course not exclude the possibility of further errors in the Scala model, due to the randomness of the approach, in contrast to Spin, which performs an exhaustive exploration of the given model's state space. However, since both are models, abstracting the real 3000 line Lisp program, both approaches may have missed errors. Note finally, that Spin in its current form (year 2018), which has evolved considerably since 1997, on a modern multi-core machine would be much faster than indicated in Figure 19. The comparison is not intended to illustrate any speed advantages of the Scala scheduler, only that one in a high-level programming language quickly can write a relatively effective test engine.

Error nr.	Kind	States Spin	Memory Spin (Mb)	Time Spin (sec)	Iterations Scala	Time Scala (sec)
1	Release	2,963	2.6	0.3	8,283 (26,682)	1.9 (2.88)
2	Abort	49,038	3.7	5.3	2,453 (122,338)	1.1 (8.44)
3	Abort	45,705	3.6	4.9	3,357 (63,264)	1.2 (5.08)
4	Abort	48,858	3.7	5.4	283,899 (329,039)	16.1 (20.31)
✓	Release	222,840	7.1	21.2	1,000,000	57.7
✓	Abort	107,479	5.0	11.6	1,000,000	54.5

Fig. 19: Verification data.

4 Conclusion

We have shown how a high-level programming language can be used for modeling. The modeling of the Remote Agent used an internal DSL for modeling HSMs, supported by automated visualization of these from their text representation. Such an approach should furthermore be supported by formal verification, possibly through a refinement relation between levels of abstraction, with mathematical specifications over infinite domains at the top level. Note, that internal DSLs, in providing the expressiveness of the host language, require the user to be a programmer in the host language, in contrast to external DSLs. This conflict between expressiveness of internal DSLs, versus notational convenience of external DSLs, can be a dilemma for DSL implementers. Note finally, that the systems modeled here are discrete systems, in contrast to continuous systems such as cyber-physical systems, which seem more challenging.

References

1. H. Barringer and K. Havelund. Tracecontract: a Scala DSL for trace analysis. In *Proc. of the 17th international conference on Formal methods*, pages 57–72, Berlin, Heidelberg, 2011.
2. D. Bjørner. Formalization of data base models. In D. Bjørner, editor, *Abstract Software Specifications, 1979 Copenhagen Winter School Proceedings*, volume 86 of *LNCS*, pages 144–215. Springer, 1979.
3. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
4. M. Broy, K. Havelund, and R. Kumar. Towards a unified view of modeling and programming. In *7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*, volume 9953, pages 238–257, 2016.
5. M. Broy, K. Havelund, R. Kumar, and B. Steffen. Towards a unified view of modeling and programming (track summary). In *7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*, volume 9953, pages 3–10, 2016.
6. 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.
7. Fortress. https://en.wikipedia.org/wiki/Fortress_(programming_language).
8. 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 *Easy-Chair Proceedings*, December 2011. Manchester, UK.
9. K. Havelund. Data automata in Scala. In *Proc. of the 8th International Symposium on Theoretical Aspects of Software Engineering (TASE'14)*, 2014.
10. K. Havelund. Monitoring with data automata. In *Proc. of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, volume 8803, pages 254–273, 2014.
11. K. Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, 17:143–170, 2015.
12. K. Havelund and R. Joshi. Modeling and monitoring of hierarchical state machines in Scala. In *9th International Workshop on Software Engineering for Resilient Systems (SERENE 2017)*, volume 10479, pages 21–36. Springer, 2017.
13. K. Havelund, M. R. Lowry, and J. Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Trans. Software Eng.*, 27(8):749–765, 2001.
14. G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.
15. S. Kauffman, K. Havelund, and R. Joshi. nfer - a notation and system for inferring event stream abstractions. In *Runtime Verification - 6th Int. Conference, RV'16*, volume 10012, pages 235–250. Springer, 2016.
16. B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith. Plan execution for autonomous spacecrafts. In *Proceedings of the International Joint Conference on Artificial Intelligence*, August 1997. Nagoya, Japan.
17. PlantUML. http://plantuml.com.
18. M. Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes, MA, USA, 2 edition, 2009.
19. Scala. http://www.scala-lang.org.
20. Scalameta. https://scalameta.org.