On the Integration of Formal Methods: Events and Scenarios in PVS and VDM

Georg Droschl*
Austrian Research Center Seibersdorf
Seibersdorf, Austria
and
Institute for Software Technology, Technical University of Graz
Graz, Austria

Abstract

Tool support is known to be one of the success factors in formal specification based analysis and -program development. This paper investigates tool support in the context of a case study where a wide range of tool features is required: For an access control, C++ code has to be developed based on the user's requirements expressed in natural language.

The access control has been classified a mixed data-control problem. This paper discusses (1) why VDMTools and PVS have been selected and (2) how they can be used together. Another aspect is the use of VDM as a framework for modeling event based systems. In our approach to tool integration, two specifications are considered to share a common part. For the present application this part consists of the scenario of all possible events.

1 Introduction

1.1 An Access Control as a Case Study

CSS is a security system which has been developed by ARCS (the Austrian Research Center at Seibersdorf [32]). CSS includes features from digital video recording to automatic door control. In a case study investigating the benefits of formal methods[1], one part of CSS is being *re-developed* based on the original requirements. This part is called SSD-0e. For brevity, we will refer to SSD-0e simply as SSD. Essentially, SSD provides an interface between the *guards on duty*, *devices* like *door-controllers*, and the *operator*.

1.2 The Roles of VDM and PVS

The activities of this project include the *formalization of SSD's user's requirements* as well as the *development of C++ code*. The aim of the first phase of the project was to gain a basic understanding of the application domain and the problem. At the end of this phase, VDM-SL has been selected as a specification language and IFAD's VDMTools for tool support.

In the second phase, a formal VDM-SL specification has been developed. It has been analyzed using the *animation* and *testing* facilities of the IFAD Toolbox. However, these features turned out not to be sufficient for our purpose. Consequently, a more in-depth analysis has been performed using theorem prover PVS.

*droschl@ist.tu-graz.ac.at

[1]This work is supported by a PhD scholarship provided by the Austrian Research Center in Seibersdorf.

1.3 Problem Definition

Our analysis confirmed that we needed to use two tools; this paper addresses the problem of how to integrate these tools. In the present case study, some of the challenges have been tackled using IFAD VDM and some using PVS. Based on the given application, this paper aims at answering the following two questions:

1. Why is it necessary to use two different tools, such as PVS and the IFAD Toolbox for VDM-SL ?
2. These tools serve different purposes. Thus, the developer has to to cope with the fact that their underlying formalisms are rather different. How has this transition from one formalism to the other been made ?

1.4 A Key Concept of the Access Control: Events and Scenarios

One of the key concepts of SSD is that of an *event*. In SSD, there are about 30 events, modeled in VDM by so-called *interface functions*. Essentially, an event is *triggered* when the *guard* or the *operator* interacts with the system.

In the analysis phase using PVS the concept of a *scenario* has been introduced. We define a *scenario* as the *set of all valid sequences of events*. It can be seen as a graph. By a *valid* sequence of events we mean those that are *covered* by the specification. The scenario of SSD will be used as an example throughout this paper.

1.5 Maintaining VDM-SL and PVS specifications

In order to be able to exploit both the features of PVS and the ones of the IFAD Toolbox, the two formalisms need to be linked. We have to deal with two *system artifacts* of SSD: a VDM-SL specification and a PVS specification. In this project, first a VDM specification has been developed. Then, it turned out that part of SSD's functionality required to be investigated more closely. Thus, part of the requirements have been formalized in the specification language of PVS. One of the results of the analysis using PVS was that the requirements (and the VDM-SL specification !) were contradictory. Thus, changes had to be made in both the requirements document and all specifications involved (in VDM-SL and PVS). The motivation of our approach is to facilitate these changes. Of course, having at one's disposal an effective means of going *back and forth between the two formalisms* is essential.

Our solution to this problem consists of dividing the specification into two parts, called LOG and ENV (for *logic* and *environment*). The key idea is that those parts of the specification which are likely to change are assigned to LOG. We will refer to this separation in the remainder of this paper as *LOG-ENV*.

The aim of *LOG-ENV* is to ease the transition between the VDM specification $SPEC_{VDM}$ and the PVS specification $SPEC_{PVS}$. Both of these specifications consist of a LOG and an ENV part. Altogether, there are four pieces of specification, LOG_{VDM}, ENV_{VDM} and LOG_{PVS}, ENV_{PVS}, respectively. Thus, we may want to write $SPEC_{VDM} = LOG_{VDM} + ENV_{VDM}$ and $SPEC_{PVS} = LOG_{PVS} + ENV_{PVS}$. $SPEC_{VDM} = LOG_{VDM} + ENV_{VDM}$ means a conjunction of specifications.

We would like to point out that the transition between the two formalisms is not automatic. Rather, by making the two LOG parts as identical as possible, we aim at facilitating the transition from one tool to the other, once the environments ENV_{VDM} and ENV_{PVS} have been set up.

1.6 Related Work

Interest in merging formal methods is growing [4, 8, 20, 23, 24, 34, 35]. Our work is closely related to [34] which deals with composing partial specifications, that are written in different specification languages. All of the specifications are given semantics are translated in a common style: predicate logic. However, [34] is very general, and gives little guidance on constructing actual multiparadigm specifications. The authors of [34] have followed up that work by developing a specific multiparadigm technique in which Z is supplemented. The technique is explained in [35]. Compared to [34] and [35] this paper focus on the tool aspect.

In [23] a meta-method of method-integration is presented, considering both formal and semi-formal notations. [23] uses Z as a "heterogeneous basis".

Facilitating the transition of core concepts from one specification language to another is an extension of previous work describing the mapping from one specification language to another [1, 19]. This work is different from [1], since in our approach, only for part of the VDM-SL specification there is a mapping to its PVS counterpart.

Concerning the tools used in the present case study, the IFAD Toolbox is being extended with facilities for theorem proving [1, 2, 3]. Recently, PVS has been complemented with features for model checking [29, 31]. In [10] the scenario of SSD has been used to build a tool for automatically generating test cases for analysis using the IFAD Toolbox. Due to this tool, the reach of the Toolbox has been extended.

In previous work reported in [11], another part of SSD's functionality has been investigated using PVS. There, the transition is based on the same principle as in this paper. This is where the idea for building the *LOG* and *ENV* parts has emerged from.

1.7 Overview of this Paper

A presentation of the access control will be given in Sec. 2. First, the basic concepts will be discussed. Parts of the VDM specification are shown for the interested reader. SSD includes interfaces to the operator and the guards on duty. The role of the operator is discussed in Sec. 2.1.1 followed by the role of the guard in Sec. 2.1.2. After a brief overview of advanced features within SSD, the notions of events and scenarios are introduced in Sec. 2.2. Section 2 concludes with a discussion of the system size of SSD. Section 3 discusses the integration of VDM and PVS. Sections 3.1, 3.2 and 3.3 argue why these tools are required for such a project. Section 3.4 gives a general presentation of the $LOG - ENV$ principle, which is illustrated on an example in Sec. 3.5. Section 3.5 contains pieces of specification and an example theorem in Sec. 3.5.6 to prove a property that is part of SSD's user's requirements. In Sec. 4 the paper is concluded by a discussion and an outlook on future work.

2 An Access Control called SSD

This section begins with a general presentation of an access control called SSD. In a fragmentary manner, parts of the VDM-SL specification will be given. It is assumed that the reader has a basic understanding of the VDM terminology [14]. Section 2.2 introduces *events* and *scenarios* in the context of SSD. Section 2 concludes with a brief discussion of the size of the application and the case study.

2.1 General Presentation

CSS is a comprehensive security system which has been developed by the Austrian Research Center [32]. SSD is one module of CSS and essentially deals with access control issues. This module is being re-developed at the Technical University of Graz in order to investigate the benefits of formal methods in software development.

SSD is an *access control* and essentially provides an interface to the *guards on duty*. The basic principle is as follows: there are a number of *guards* each of which follows a certain *round*, for example to supervise a factory by night. Each round consists of a *list of stations* which are supposed to be visited (and "hit" by the guard) one-by-one.

The information that is stored by SSD internally, is essentially a *collection of rounds*. In the terms of VDM, the *global state* of SSD is implemented as a *mapping RDS* of type *RMap* from the round's address of type *R-Adr* to elements of type *R*. R contains all relevant information for a round.

```
state SSD of
         RDS : RMap
  end
```

```
types
  RMap = inmap R_Adr to R
  inv r == wf_equally_codable(r);
```

A round can be in state *selected* or *unselected*. The *interesting* rounds are the ones that are *selected*. Only when the round has been selected, it may be supervised by a guard. Each selected round has its proper *selection mode*. Altogether, there are *three* selection modes. In a round, either there is *one guard* or a team of *two guards*. For each of these options, there is one selection mode.

When the operator selects a round, one of the three *selection modes* has to be chosen. The two selection modes aimed at *supervision* are called *executing*. *Recording* is a selection mode where the guard proceeds through the round in order to pre-set the *mean times* it takes him or her from one station to the next.

```
  SelectionMode = <recording> | <exec_one> | <exec_two> | <unselected>;
```

We will now show the *top-level definition* of type R, which is a type for holding all relevant information of a round[2]. A very limited discussion of each of these fields will be given below.

```
types
R ::    r_adr        : R_Adr
        rstate       : RoundState
        selmode      : SelectionMode
        stations     : RdStations
        lasthit      : [LastHitStation]
        nexthit      : [NextHitStation]
        guards       : [RdGuards]
        isrecorded   : bool
        isprogrammed : bool
        timer_ticking: bool
        round_timer  : Time
        interrupted  : [Interrupted]
 inv r==wf_recorded_imp_sectime(r)    and
        wf_programmed_imp_timedev(r)  and
        wf_timer_ticking (r)          and
        wf_assigned_guards_selmode(r) and
        wf_assigned_guards_hit(r)     and
        wf_selected_with_mode (r);
```

R-Adr is the *address* of the round which is simply an identifier of the round. *RoundState* holds a list of stations followed by the *SelectionMode* given above. Then, there is information for the system to be able to keep track of which ones are the *LastHitStation* and the *NextHitStation*, respectively. Once the guard has *hit the first station* of the round (*"starting" it*), the round is considered *running*. A round may only be selected for executing if it is both *programmed* and *recorded* meaning that a list of stations has been assigned to the round and that the *mean time* it may take a guard from *one station to the next* has been determined. Finally, there is a *round-timer* and a field holding information if the round has been *interrupted*.

This block is followed by a so-called *invariant* definition constraining the data type to those rounds that are considered *well-formed*.

2.1.1 The Role of the Operator

In SSD, there is a human operator who supervises the system and the guards. The operator may *select*, *interrupt*, and *terminate rounds*. He or she may also take stations *out of order* temporarily (*disable them*) or enable the guard to *continue* the round at some station if the round has been *interrupted*.

[2]Compared to the original specification, the fields and invariants for a mechanism called *switches* are not shown.

At this level of abstraction, the interface of SSD is modeled by a number of so-called *interface functions*. The first example for such an interface operation is the one for the operator to *select a round for executing with one guard*. Again, we do not expect the reader to understand every detail.

```
operations
  OPselX1 (r0:R) new_r:R ==
  (
   dcl new_r:R := r0;
   dcl old_rounds:RMap := RDS;
   new_r.selmode:=<exec_one>;
   new_r.rstate.waiting_begin:=true;
   RDS:=old_rounds ++ { new_r.r_adr |-> new_r };
   return new_r
  )
  ext wr RDS
  pre  R_exists(r0,RDS)         and
       R_is_recorded(r0)        and
       R_is_programmed(r0)      and
       not R_selected(r0)

  post R_selected(new_r)        and
       R_waiting_begin(new_r)   and
       R_Mode_ex1(new_r)        and
       same_R(new_r,r0)         and
       R_exists(new_r,RDS);
```

This is an example for an *extended implicit operation*, meaning that there are both implicit and explicit representations of the operation's functionality. *OPselX1* takes a round *r0* as its argument and returns the updated round *new-r*. It operates on the global state RDS as a *side-effect* where the updated round is also written back to.

In the explicit part, first two *temporary local states* are defined. Then, the (new) round is put in selection mode *executing with one guard* and a *flag is set*, indicating that the round is *waiting for the guard to hit the first station*. Finally, the new round is *written back* to the global state.

The explicit part is followed by the implicit part. It consists of a pre- and a post condition. Essentially, this operation may only be invoked, provided *recording* and *programming* have already taken place and that once the operation has been carried out. According to the post condition, after invocation, the round is *selected for executing with one guard*.

2.1.2 The Role of the Guard

The main *task* of the guard is to visit (and "*hit*") stations. In selection mode *recording* the guard is asked to walk through the round such that the mean time values to get from one station to the next can be determined.

Hitting a Station. Once the operator has selected a round, he or she has enabled the guard to start the round by hitting the first station of that round[3]:

```
operations
  HX1first (r0 : R, hr: S_hit_return)
                   new_r: R ==
```

[3]From now on, the explicit part of operations will not be shown.

```
(   ...
)
ext wr RDS
pre   R_waiting_begin(r0)              and
      R_Mode_ex1(r0)                   and
      not another_recording_R(r0,RDS)  and
      not time_intersection (r0,RDS)   and
      R_exists(r0,RDS)

post  R_running(new_r)                  and
  G_on_duty_in_R(r0,G_id(hr))           and
  S_last_hit(new_r)=1                   and
  S_time_last_hit(new_r)=when_hit_S(hr) and
  same_R(new_r,r0)                      and
  R_exists(new_r,RDS);
```

Essentially, after operation *HX1first* has been invoked, the round is no longer *waiting to begin*, the *identity of a particular guard* has been assigned to the round, and a reference to the station *last hit* has been stored.

An *intermediate station* is any station but the first and the last one. Similar to the operation dealing with the situation that the guard has hit the first station, there are operations for intermediate and last stations.

2.1.3 Advanced Features

There are a number of advanced features of SSD including *automatic round supervision* and automatic unlocking/locking of doors (*switches*). *Automatic supervision* by *checking error conditions* frees the operator from supervising every round all the time. Whenever a station has been hit, a number of checks are performed: *Is the guard's identity o.k. ? Did the guard hit the expected station ? Is the time it took the guard from one station to the next sufficiently close to the "mean" time ?* In case one of these checks fails, the round is *interrupted* and the operator is asked to interact. Another interesting feature is the (automatic) treatment of *switches*. Switches make sure that devices like doors open or close automatically when a certain station of the round has been hit. Among the devices that may be controlled this way are *intrusion circuits*, *doors* and *light switches*.

2.2 Events and Scenarios

One of the key concepts of SSD is that of an *event*. Essentially, one of about 30 SSD events is *triggered* when the *guard* or the *operator* interacts with the system. For *one single* round, Fig. 1 summarizes SSD's events and the set of possible transitions between events. The naming principle of events is as follows: basically there are three *groups* of events, designated by the first letter/first two letters in the event name ("H", "OP" and "E", respectively).

The *first* type of events may occur when a guard has visited (and *"H"it*) a station. For this group, there are separate events for each of the three *selection-modes* introduced in Sec. 2.1, *executing with one guard* ("X1"), *executing with two guards* ("X2") and *recording* ("RE"). Apart from a "normal" hit (an intermediate station, that is), there are events for the "first" and the "last" station, as well as events for hits which "continue" the round after it has been interrupted. In selection mode *executing with two guards* the first guard ("G1") and the second one ("G2") are distinguished. If, at any time, the guard is threatened by an intruder, a mechanism called *silent alarm* may be activated. The corresponding event is called *Halarm*.

The second type of event corresponds to *operator commands* ("OP"). The operator may select ("sel") a round for any of the selection-modes, interrupt a round ("int"), enable the guard to continue after the round has been interrupted ("cont") or terminate the round ("term"). A pending alarm needs to be cleared

("clralarm") before the guard may be enabled to continue. Finally, there is a mechanism of taking single stations temporarily out of order: the operator may *disable* ("dis") and *enable* ("en") stations, provided the round is *unselected*.

The third type of event deals with error handling features ("E"). Again, most of the events are separated by selection-mode. The basic three types of error are *wrong sequence of stations followed* ("seq"), *time limits exceeded* ("time") and *identification error* ("id").

Fig. 1 presents the set of events and possible sequences of events. To be of practical use, all that remains to be defined is the set of potential *first events*, $FirstEvents = \{OPenS, OPdisS, OPselRE, OPselX1, OPselX2\}$. In other words, it is the operator who always causes the initial event of a round. He or she may either *disable/enable* one of its stations or *select* the round.

2.3 System Size

Both the original implementation of SSD, and the one under development using formal methods, are based on a list of 60 requirements given on 10 A4 pages, expressed in English language. The program developed using "traditional" methods, which of course, has not been made available, consists of about 12.000 lines of PASCAL code.

In the second phase of this case study, about 35 A4 pages of VDM specification have been developed. This has taken 6 months. The entire documentation including the formal specification, informal text and some diagrams, consists of 70 A4 pages.

Subsequently, PVS has been used to perform a rigorous analysis of a *critical part* of SSD's functionality [11]. This specification consists of about a thousand lines.

3 Linking VDM and in PVS

As pointed out previously, this paper aims at answering the following two questions: first, why is it necessary to use two different tools ? This will be subject of sections 3.1, 3.2 and 3.3. Second, how has the transition been made, from one formalism to the other ? This will be discussed in the remainder of Sec. 3. Our approach consists of dividing the specification into two parts, which will be given in Sec. 3.4. In Fig. 1, one aspect of SSD's behavior has been given: its *scenario*. In the present section, it will serve as an example for linking the tools PVS and VDM. It will be shown how to create a two-part specification for both formalisms. The section concludes with the VDM and PVS specifications for scenarios, constructed following the two-part approach.

3.1 Why VDM ?

The scope of this project is characterized by the fact that a large part of the *development life cycle* has to be covered: the development is based on a document containing a set of informal requirements. Ultimately, the module has to be implemented in a programming language like C++ such that the *formally developed version* can be compared to the one developed using "traditional" methods.

In *formal methods* there is a variety of environments (and tools) including a wide range of features such as *theorem proving*, *interpretation* and *automatic code generation* from specifications [25, 15]. At the end of the domain/problem understanding phase, VDM-SL has been selected as a specification language and IFAD's VDMTools for tool support: In [7] a taxonomy of applications is given. According to the item *relative difficulty of data, control, and algorithmic aspects of problem*, SSD may essentially be classified as a *mixed data-control* problem. VDM-SL [21] is a specification language that is well-suited for modeling data. VDM-SL has been chosen as a specification language, even though it might not be the first candidate for modeling *event-based* systems. VDM-SL has been standardized [21]. There is a comprehensive list of publications related to VDM and VDM-SL called the *VDM bibliography* [33].

For VDM-SL there is good tool support: the IFAD Toolbox for VDM-SL [13] has achieved a certain level of maturity. The IFAD Toolbox has been used in a number of substantial projects [18]. For details of the

	Event	Possible next event
01	HX1first	HX1norm HX1last OPint OPterm Halarm
02	HX1norm	HX1norm HX1last OPint OPterm Halarm
03	HX1last	OPselRE OPselX1 OPselX2 OPdisS OPenS
04	HX1cont	HX1first HX1norm HX1last Halarm OPint OPterm
05	HREfirst	HREnorm HRElast OPterm
06	HREnorm	HREnorm HRElast OPterm
07	HRElast	OPselRE OPselX1 OPselX2 OPdisS OPenS
08	HX2firstG1	HX2firstG2 Halarm OPint OPterm
09	HX2firstG2	HX2normG1 Halarm OPint OPterm
10	HX2normG1	HX2normG2 HX2lastG2 Halarm OPint OPterm
11	HX2normG2	HX2normG1 Halarm OPint OPterm
12	HX2lastG2	OPselRE OPselX1 OPselX2 OPdisS OPenS
13	HX2contG1	HX2firstG2 HX2normG2 Halarm OPint OPterm
14	Halarm	OPclralarm OPterm
15	OPselRE	HREfirst OPterm Halarm
16	OPselX1	HX1first OPterm Halarm
17	OPselX2	HX2firstG1 OPterm Halarm
18	OPint	OPcont OPterm
19	OPcont	OPterm HX1cont HX2contG1 Halarm
20	OPterm	OPselRE OPselX1 OPselX2 OPdisS OPenS
21	OPclralarm	OPcont OPterm
22	OPdisS	OPenS OPselRE OPselX1 OPselX2
23	OPenS	OPdisS OPselRE OPselX1 OPselX2
24	Eintersect	HX1first HREfirst HX2firstG1 OPterm Halarm
25	EX1seq	OPcont OPterm
26	EX1time	OPcont OPterm
27	EX1id	OPcont OPterm
28	EX2seq	OPcont OPterm
29	EX2timeG1	OPcont OPterm
30	EX2id	OPcont OPterm
31	EX2twiceG1	OPcont OPterm
32	EX2twiceG2	OPcont OPterm
33	EX2anotherG2	OPcont OPterm
34	EX2timeG2	OPcont OPterm

Figure 1: SSD's scenario: table of events and possible transitions. There are three *groups* of events, denoted by the first (two) letter(s) in the event name in the middle column ("H", "OP" and "E"). See the text for a more detailed explanation.

use of the IFAD Toolbox in this project see [9]. We would like to point out that the IFAD Toolbox supports automatic C++ code generation for an (executable) subset of VDM-SL.

3.2 Why PVS ?

In the second phase of this project, a formal VDM-SL specification has been developed. It has been analyzed using the *animation* and *testing* facilities of the IFAD Toolbox. One of the results of this first analysis has been that the requirements, out of which the specification has been developed, were contradictory [11]. In fact, the requirements include a proposal for an algorithm which has been shown to fail for a certain scenario. This observation has given rise for further analysis. Even though the testing facilities of the Toolbox enabled us to gain evidence for the contradictory nature of the requirements, we felt that testing was not a sufficient approach for further analysis.

[5, 25] provide a discussion of different approaches to formal specification based analysis, including *theorem proving* and *model checking*. Model checkers essentially perform an enumeration of all possible states. Thus, the complexity of the state space of the application to be analyzed may be a limiting factor. Since SSD is essentially a data-problem, model checkers were considered *unsuitable* for this task, even though advanced approaches to *state compression* have been proposed [6, 17].

There is a great number of theorem provers [22]. The *Prototype Verification System* (PVS) [26, 28, 30] is one of these tools. It has been applied successfully to many substantial examples [27]. Even though it is known that PVS has potential bugs, PVS is one of the most popular tools and is thus very well supported.

Another problem is that it may be difficult to introduce a new formalism in a project where a major specification exists. However, VDM-SL specifications have been related to PVS specifications [1].

For the type of PVS analysis performed in this project, please refer to Sec. 3.5.6 where an example proof is shown.

3.3 Why VDM and PVS ?

In *formal methods* there is a variety of features [25] and tools [15]. From what has been said above, it should have become clear, that it is unreasonable to expect *one single tool* to be used for the *whole* of a formal development project [4, 20, 23, 24, 34, 35]. It has been shown that not all formalisms are equally well suited for the various issues arising in projects applying formal methods. The question that arises is, whether two or more tools may be integrated to form a new tool and how to do that efficiently.

3.4 Transition Principle LOG-ENV

As it has been said before, some challenges of the case study have been tackled using IFAD VDM and some using PVS. In order to exploit the features of PVS and those of the IFAD Toolbox, their formalisms need to be linked. In other words, we have to deal with two *system artifacts*: a VDM-SL specification and a PVS specification. Our approach for this problem consists of dividing the specification into *two parts*, one called *LOG* (for logic) and another one called (*ENV* for environment). This principle is illustrated in Fig. 2. For brevity, we call it *LOG-ENV*.

3.4.1 How should the specification be divided ?

The key idea here is to put the part of the specification that it likely to change into *LOG*.

The goal of *LOG-ENV* is to ease the transition between the VDM specification and the PVS specification, each of which consists of a *LOG* and an *ENV* part. Thus we have two specifications $SPEC_{VDM}$ and $SPEC_{PVS}$. In each of these specification, there are two parts, LOG_{VDM}, ENV_{VDM} and LOG_{PVS}, ENV_{PVS}, respectively. Thus, we may want to write $SPEC_{VDM} = LOG_{VDM} + ENV_{VDM}$ and $SPEC_{PVS} = LOG_{PVS} + ENV_{PVS}$.

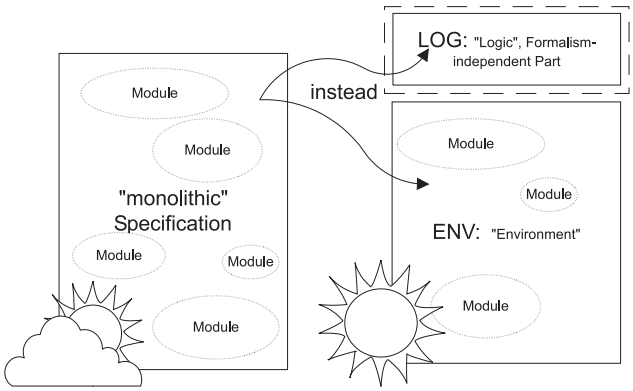

Figure 2: THE LOG-ENV principle: use of a two-part specification instead of having a monolithic specification. The *LOG* part (for *logic*) contains the part of the specification that it likely to change. The rest of the specification is assigned to ENV.

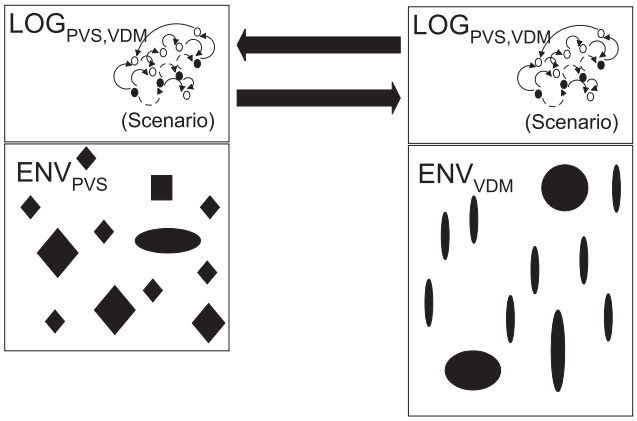

Figure 3: LOG-ENV applied to PVS and VDM: the LOG part can be exchanged easily, because $LOG_{PVS,VDM}$ is part of both the VDM and the PVS specifications. $LOG - ENV$ differs from earlier work [1] in a sense that there exists a *mapping* only between the LOG parts of the specifications rather than between the *entire* specifications.

Another question that arises is whether the assumption $LOG_{VDM} \cong LOG_{PVS}$ is reasonable, for example because of differences in the specification languages. However, in the Sec. 3.5 a pattern will be presented for which the assumption is considered valid.

Even though the transition between the two formalisms is not meant to be totally automatic, by making the two *LOG* parts as identical as possible, we aim at *facilitating* the transition. By writing $LOG_{VDM} \cong LOG_{PVS}$ we mean that the differences between LOG_{VDM} and LOG_{PVS} are "minor" (in the examples given below, the only differences between LOG_{VDM} and LOG_{PVS} are due to the differences of the specification languages). In Fig. 3 the LOG-ENV principle is applied to integrating PVS and VDM: the LOG part can be exchanged easily, because $LOG_{PVS,VDM}$ is part of both the VDM and the PVS specification.

3.5 Example

In the following, $LOG-ENV$ will be applied to the scenario example. We will begin by identifying adequate representations in PVS and VDM, based on the task to be performed by PVS on the example.

3.5.1 Dividing the example specification.

The process of drawing a line between *LOG* and *ENV* can be guided by the following questions: Given the scenario example...

1. ... how can the border between *LOG* and *ENV* be drawn ?
2. ... which roles do PVS and VDM have to play ?
3. ... how would the (monolithic) PVS and VDM specifications look like ?
4. ... which part of the specification is the one that is likely to change ?
5. ... which part of the specifications is common and, according to our terminology, corresponds to $LOG_{PVS,VDM}$?

What is the objective of PVS in this example ? The set of possible sequences of events given in Fig. 1 (the scenario) has been defined based on the insights gained in the *analysis* of the specification rather than *directly* on the *requirements*. However, the following rule is part of the requirements:

Rule: In selection mode executing, it shall be possible to interrupt a round at any time.

The question is whether the given scenario satisfies this rule. PVS can be used for the proof of such a property. The result of such a *proof attempt* is either a *change request* for the scenario, or an *approval*. In consequence, it is the *scenario itself* that will be subject to change. It is assigned to LOG_{PVS} (and LOG_{VDM}).

3.5.2 LOG_{VDM} for the Scenario

In the case study as well as in the present example, the VDM specification covers the *entire* functionality of SSD. This specification is structured as follows: there are a *state*, a number of *interface operations*, *data type definitions*, and *auxiliary functions*. Parts of the VDM specification have been shown in Sec. 2. There is a strong link between *interface functions* and the events given in Fig. 1, simply because for each of the given events, there is one interface function. As it can be seen in Sec. 2, they even share the same names.

What is the objective of VDM in the given example ? It is important for the VDM specification to have at its disposal a means for determining whether a sequence of events that has occurred in reality is in fact covered by the system. In consequence, the system may be guaranteed never to leave the well-defined behavior *without noticing it*. This mechanism may make use of function `next_event`.

```
functions  ------------- LOG VDM part -----------------------------------
 next_event : event * event -> bool
 next_event (e,f) ==
  cases e :
   <HX1first> -> f=<HX1norm> or f=<HX1last> or
                 f=<OPint>   or f=<OPterm>  or f=<Halarm>,
   <HX1norm> ->  f=<HX1norm> or f=<HX1last> or
                 f=<OPint>   or f=<OPterm>  or f=<Halarm>,
   <HX1last> ->  f=<OPselRE> or f=<OPselX1> or
                 f=<OPselX2> or f=<OPdisS>  or f=<OPenS>,
 ...
   <EX2twiceG2>     ->  f=<OPcont> or f=<OPterm>,
   <EX2anotherG2>   ->  f=<OPcont> or f=<OPterm>,
   <EX2timeG2>      ->  f=<OPcont> or f=<OPterm>
   end ------------  end LOG VDM part -----------------------------------
```

Figure 4: The Scenario Example: LOG_{VDM}

```
types ------------------ ENV VDM part -----------------------------------
  event =
   <HX1first>   | <HX1norm>    | <HX1last>      | <HX1cont>    |
   <HREfirst>   | <HREnorm>    | <HRElast>      |
   <HX2firstG1> | <HX2firstG2> | <HX2normG1>    | <HX2normG2>  |
   <HX2lastG2>  | <HX2contG1>  | <Halarm>       |
   <OPselRE>    | <OPselX1>    | <OPselX2>      |
   <OPint>      | <OPcont>     | <OPterm>       | <OPclralarm> |
   <OPdisS>     | <OPenS>      |
   <Eintersect> | <EX1seq>     | <EX1time>      | <EX1id>      |
   <EX2seq>     | <EX2timeG1>  | <EX2id>        |
   <EX2twiceG1> | <EX2twiceG2> | <EX2anotherG2> | <EX2timeG2>;

  ... insert rest of VDM specification here ...
 -----------------  end ENV VDM part -----------------------------------
```

Figure 5: The Scenario Example: ENV_{VDM}

Among other things, interface functions are part of ENV_{VDM} whereas the scenario is again the only thing that is part of LOG_{VDM}, shown in Fig. 4.

Please note that there are some syntactic differences between LOG_{PVS} and LOG_{VDM}. This is why we write $LOG_{VDM} \cong LOG_{PVS}$ instead of $LOG_{VDM} = LOG_{PVS}$.

3.5.3 ENV_{VDM} for the Scenario Example

Major parts of the VDM specification have been shown in Sec. 2. According to this example, these pieces of specification are all part of ENV_{VDM}. The only definition that has to be added is the data type `event`. ENV_{VDM} is shown in Fig. 5.

```
scenario : THEORY % --- LOG PVS part -----------------------------------
  BEGIN
next_event (e,f:event) : bool =
 cond
   e=HX1first  -> f=HX1norm OR f=HX1last OR f=OPint OR f=OPterm OR f=Halarm,
   e=HX1norm   -> f=HX1norm OR f=HX1last OR f=OPint OR f=OPterm OR f=Halarm,
   e=HX1last -> f=OPselRE OR f=OPselX1 OR f=OPselX2 OR f= OPdisS OR f=OPenS,
 ...
   e=EX2twiceG2     ->  f=OPcont  OR f=OPterm,
   e=EX2anotherG2   ->  f=OPcont  OR f=OPterm,
   e=EX2timeG2      ->  f=OPcont  OR f=OPterm
 endcond % ------- end of LOG PVS part ----------------------------------
...
```

Figure 6: The Scenario Example: LOG_{PVS}

3.5.4 LOG_{PVS} for the Scenario

The LOG_{PVS} part of the specification is given in Fig. 6[4]. This is the first half of a file called `scenario.pvs`. It defines a function `next_event` which returns the boolean value `true` for all *valid* sequences of events. The function takes a tuple of two events as arguments, where `f` follows `e` *in time*[5].

3.5.5 ENV_{PVS} for the Scenario Example

The second part of the file `scenario.pvs`, corresponding to ENV_{PVS} in shown in Fig. 7. Data type `event` defines all possible events, followed by a number of predicates each of which divide the predicates into two groups. These predicates will be needed by the theorem called `interrupt_always_possible`, shown at the end of the file: `is_error_event` holds for all events signaling that an error has occurred. This information is relevant because in the case of error the round will be interrupted automatically, and can thus not be interrupted once more.

`is_recording_event` denotes all the events of selection-mode recording, where interruption is not possible. `is_opinterrupt_event` holds for the *operator command* of *round interruption*. Clearly, the operator may not interrupt the same round twice in a row.

`event_leads_to_active_round` holds only for those events, which do not "un-select" the round.

3.5.6 A PVS Example Proof

Theorem provers like PVS allow to formally prove that specifications satisfy mathematical properties. Finally, it will briefly be shown on an example how SSD's is being analyzed by means of PVS. Please note that the PVS analysis of SSD, based on events and scenarios is ongoing. However, the final results of PVS analysis of another aspect of SSD are reported in [11].

In Fig. 8 an example proof in PVS is shown. The need for a proof is based on the rule given above. Please recall that it has been taken from SSD's user's requirements. Fig. 8 includes a theorem which formally expresses what is to be shown.

[4]The order of the blocks within the specification has been changed for presentation purposes. In PVS, data type `event` has to be declared before it may be used.

[5]The entire function can easily be reconstructed from the table given in Fig. 1.

```
...
event : TYPE = % ------- ENV PVS part ------------------------------------
  {
   HX1first, HX1norm, HX1last, HX1cont,
   HREfirst, HREnorm, HRElast, Halarm,
   HX2firstG1, HX2firstG2, HX2normG1, HX2normG2, HX2lastG2, HX2contG1,
   OPselRE, OPselX1, OPselX2,   OPint, OPcont, OPterm, OPclralarm,
   OPdisS, OPenS,
   EX2seq, EX2timeG1, EX2id,    Eintersect, EX1seq, EX1time, EX1id,
   EX2twiceG1, EX2twiceG2, EX2anotherG2, EX2timeG2 }

  % ---- predicates dividing events into groups -----
is_error_event (e:event ) : bool =
  e=Eintersect OR e=EX1seq     OR e=EX1time      OR e=EX1id     OR
  e=EX2seq     OR e=EX2timeG1  OR e=EX2id        OR
  e=EX2twiceG1 OR e=EX2twiceG2 OR e=EX2anotherG2 OR e=EX2timeG2

END scenario %---- end of ENV PVS part ----------------------------------
```

Figure 7: The Scenario Example: ENV_{PVS}

```
  % -------------------------  aux functions -----------------------------
is_recording_event (e:event) : bool =
 e= HREfirst OR e=HREnorm OR e=HRElast

is_opinterrupt_event (e:event) : bool =
 e= OPint

event_leads_to_active_round (e:event) : bool =
NOT( e=HX1last AND e=HRElast AND e=HX2lastG2 AND
     e=OPterm  AND e=OPdisS  AND e=OPenS )

  % -------------------------- theorem to be proved ----------------------
interrupt_always_possible: THEOREM
 FORALL (e,f: event) : (NOT is_error_event(e)               AND
                         NOT is_recording_event(e)          AND
                         NOT is_opinterrupt_event (e)       AND
                             event_leads_to_active_round (e) AND
                             is_opinterrupt_event (f)     ) IMPLIES
         next_event(e,f)
```

Figure 8: An Example Proof in PVS

4 Conclusion

In this paper, we have presented an approach for integrating two tools: PVS and the IFAD Toolbox. Based on an example, it has been shown why it is *necessary* to use two different tools, and *how* the transition from one formalism to the other has been made. What we called the $LOG - ENV$ principle encourages to divide a specification into *two parts*. It has been shown, how the part of the specification that is *likely to change*, can be (1) *separated* from the rest of the specification, (2) expressed in a *compact* form, (3) kept almost *independent* of the *specification language*, and (4) *modified* easily in either specification.

Compensate for Differences in the Environments. *LOG-ENV* provides a means for compensating for differences between the features among two environments. In the example given above, the PVS proof makes use of a number of of auxiliary predicates. On the VDM side, the specification contains "implementations" of events by means of interface functions. Either of these constructs are irrelevant in the opposite formalism.

A Means for Abstraction. Given two specifications $SPEC_{VDM}$ and $SPEC_{PVS}$, we assumed that each of them consists of two parts LOG_{VDM}, ENV_{VDM} and LOG_{PVS}, ENV_{PVS}, respectively. In order to gain an understanding of the high-level concepts of the system, the ENV parts of the specification are of secondary interest. We would like to attract the attention of the reader to the fact, that *LOG-ENV* represents a means for *abstraction without restricting the features that may be performed* by the two tools.

4.1 Future Work

The selection of the boundary between LOG and ENV very much depends on the concrete task. So far, in [11] and the present paper, two applications have been presented. However, it remains to be investigated, how specifications - in general- can be split.

The central topic of our work is tool support for formal methods. An aspect that has been found important, is tool support in the context of formalizing requirements which are expressed in natural language. First results have been reported in [12] which will lead to a more comprehensive view of the issue of tool support than it has been provided in the present paper.

5 Acknowledgments

The author of this paper would like to thank Erwin Schoitsch, Wolfgang Herzner, Hossein Selami and Walter Kuhn of the Austrian Research Center for their interest in applying formal methods on an industrial application. Peter Lucas has provided invaluable guidance throughout this project. Thanks go to the members of the local formal methods group for comments on various aspects of this work. In particular, we would like to mention Bernhard Aichernig, Brigitte Fröhlich, Andreas Gerstinger, Johann Hörl, Heinz Kammerlander, Andreas Kerschbaumer and Rudi Schlatte. We are most grateful to Rich Paige and two anonymous referees for providing feedback on this paper. Andreas Gerstinger, Peter Lucas and Rudi Schlatte have read earlier versions of this text.

References

[1] Agerholm S. Translating Specifications in VDM-SL to PVS. In: Grundy J., Von Wright J., Harisson J. (eds) Proceeding of the 9th International Conference On Theorem Proving in Higher Order Logics, Turku, Finland. Springer Verlag, 1996, pp 1–16 (Lecture Notes in Computer Science No. 1125)

[2] Agerholm S. Experiments with ZF Set Theory in HOL and Isabelle. In: Schubert E.T., Windley P.J., lves-Foss J. (eds) Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications, Aspen Grove, Utah, USA. Springer-Verlag, 1995, pp 32–45 (Lecture Notes in Computer Science No. 971)

[3] Agerholm S. An Isabelle-Based Theorem Prover for VDM-SL. In: Gunter E.L., Felty A. (eds) Proceedings of Theorem proving in higher order logics: 10th International Conference, TPHOLs '97, Murray Hill, NJ, USA. Springer-Verlag, 1997, pp 1–? (Lecture Notes in Computer Science No. 1275)

[4] Ambriola V., Cignoni G.A., Fernstroem C. Current Issues on Integration. In: Schafer W. (ed) Proceeding of Software process technology: 4th European workshop EWSPT '95, Noordwijkerhout, The Netherlands. Springer-Verlag, 1997, pp 197–? (Lecture Notes in Computer Science No. 913)

[5] Clarke E.M., Wing J.M. Formal Methods: State of the Art and Future Directions. ACM Computing Surveys 1996; 28:626-643

[6] Clarke E.M., McMillan K.L., Campos S., Hartonas-Garmhausen V. Symbolic model checking. In: Alur R., Henzinger T.A. (eds) Proceedings of the Eighth International Conference on Computer Aided Verification CAV, New Brunswick, NJ, USA. Springer-Verlag, 1996, pp 419–422 (Lecture Notes in Computer Science No. 1102)

[7] Davis A. M. Software Requirements: Objects, Functions and States. Second Edition. Prentice Hall, Englewood Cliffs, 1993.

[8] Desharnais J., Khedri R., Frappier M. Mili A. Integration of Sequential Scenarios. In: Jazayeri M. and Schauer H. (eds) Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97). Springer-Verlag, 1997, pp 310–326 (Lecture Notes in Computer Science No. 1013)

[9] Droschl G. Formal Specification and Analysis of an Access Control using IFAD's VDMTools. Institute for Software Technology, Technical University of Graz, Austria, 1999. Technical Report IST-TEC-99-06. Submitted for Publication.

[10] Droschl G. Design and Application of a Test Case Generator for VDM-SL.
To appear in the proceedings of FM'99: VDM in Practice! Toulouse, France, September 20–21.

[11] Droschl G. Using PVS for Requirements Analysis of an Access Control. To be presented at FM'99: World Congress on Formal Methods (Industrial Experience Discussion Session), Toulouse, France, September 20–24.

[12] Droschl G. Towards a Framework for Formalizing Requirements. Technical Report IST-TEC-99-10, Institute for Software Technology, Technical University of Graz, Austria, June 1999.

[13] Elmstrøm R., Larsen P.G., Lassen, P. B. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. ACM SIGPLAN Notices 1994; 29:77-80

[14] Fitzgerald J., Larsen P.G. Modeling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, Cambridge, UK, 1998.

[15] Mac an Airchinnigh M. Formal Methods Europe (FME) Hub.
http://www.csr.ncl.ac.uk/projects/FME/InfRes/

[16] Schneider F., Easterbrook S.M., Callahan J.R., Holzmann G.J. Validating Requirements for Fault Tolerant Systems using Model Checking. In: Proc. International Conference on Requirements Engineering, ICRE, Colorado Springs, Co., USA. IEEE Computer Society Press 1998.

[17] Holzmann G.J. State Compression in Spin. In: Proc. Third Spin Workshop, Twente, The Netherlands. Twente University, 1997.

[18] IFAD. VDM Examples Repository. Maintained by Larsen P.G.
http://www.ifad.dk/examples/examples.htm

[19] Kellomäki P. Verification of Reactive Systems Using DisCo and PVS. In: Fitzgerald J., Jones C.B., Lucas P. (eds) FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria). Springer-Verlag, 1997, pp 589–604 (Lecture Notes in Computer Science No. 1313)

[20] Kramer J., Finkelstein A., Nuseibeh B. Method Integration and Support for Distributed Software Development: An Overview. In: Lamb D.A. (ed) Studies of software design: ICSE '93 workshop Baltimore, Maryland, USA. Springer-Verlag, 1996, pp 115–? (Lecture Notes in Computer Science No. 1078)

[21] Larsen P.G., Plat N. An Overview of the ISO VDM-SL Standard. ACM SIGPLAN Notices 1992; 27:76-82.

[22] Paulson L. List of Theorem Provers. http://www.cl.cam.ac.uk/users/lcp/

[23] Paige R.F. A Meta-Method for Formal Method Integration. In: Fitzgerald J., Jones C.B., Lucas P. (eds) FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria). Springer-Verlag, 1997, pp 473–? (Lecture Notes in Computer Science No. 1313)

[24] Paige R.F. Case Studies in using a Meta-Method for Formal Method Integration. In: Johnson M. (ed) Algebraic methodology and software technology: 6th International Conference, AMAST 97, Sydney Australia. Springer-Verlag, 1997, pp 395–? (Lecture Notes in Computer Science No. 1349)

[25] NASA. Formal Methods, Specification and Verification Guidebook for Verification of Software and Computer Systems. Vol 2: A Practitioner's Companion. Washington, DC, USA. 1997.

[26] Owre S., Rushby J., Shankar, N. PVS: A Prototype Verification System. In: Kapur D. (ed) Automated deduction, CADE-11: 11th International Conference on Automated Deduction, Saratoga Springs NY, USA. Springer-Verlag, 1992, pp 748–? (Lecture Notes in Computer Science No. 607)

[27] Rushby J. List of PVS applications. SRI Computer Science Laboratory, Menlo Park, CA, USA. http://pvs.csl.sri.com/applications.html

[28] Rushby J., Owre S., Shankar N. Subtypes for Specifications: Predicate Subtyping in PVS. IEEE Transactions on Software Engineering 1998; 24:709-720.

[29] Rushby J., Ubiquitous Abstraction: A New Approach for Mechanized Formal Verification (Extended Abstract). In: Proceedings of Second International Conference on Formal Engineering Methods (ICFEM '98), Brisbane, Australia. pp 176–178. IEEE Computer Society, 1998.

[30] Rushby J., Stringer-Calvert D. A Less Elementary Tutorial for the PVS Specification and Verification System. Computer Science Laboratory, SRI International, Menlo Park, CA, 1995. Technical Report SRI-CSL-95-10.

[31] Rusu V., Singerman E. On Proving Safety Properties by Integrating Static Analysis, Theorem Proving and Abstraction. To appear in the proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), 1999.

[32] Austrian Research Center Seibersdorf, ARCS. http://www.arcs.ac.at

[33] Larsen P.G. The VDM Bibliography. http://liinwww.ira.uka.de/bibliography/SE/vdm.html

[34] Zave P., Jackson M. Conjunction as Composition ACM Transactions of Software Engineering and Methodology 1993; 2:379-411.

[35] Zave P., Jackson M. Where Do Operations Come From: A Multiparadigm Specification Technique. IEEE Transactions on Software Engineering 1996; 22:508-528.