A CASE STUDY ON THE APPLICATION OF PVS TO REQUIREMENTS ANALYSIS

Georg Droschl
droschl@ist.tu-graz.ac.at
Austrian Research Center Seibersdorf
and
Institute for Software Technology, Technical University of Graz
Münzgrabenstr. 11 / 2, 8010 Graz, Austria, Europe

ABSTRACT

This paper presents the results of a formal methods case study in which the Prototype Verification System (PVS) has been used for requirements analysis of one module of a physical access control system. PVS is a tool for writing formal specifications and constructing proofs. Previously, the same requirements have been analyzed by means of testing supported by the IFAD Toolbox for VDM-SL. The capabilities of the two formal methods are compared.

INTRODUCTION

Formal Methods

Informal requirements are inappropriate to serve as a basis for software development. Still, most requirements documents are expressed in natural language. "Formal methods are one of a number of techniques that when applied correctly have been demonstrated to result in systems of the highest integrity" (Bowen and Hinchey, 1995). Formal methods exploit the power discrete mathematics. Their mathematical basis includes the disciplines of set theory, abstract algebras, the LambdaCalculus, Petri Nets, etc.
The term 'formal methods' embraces formal specification and verified design. The two main approaches to formal specification are called model-oriented, and property oriented. Model-oriented specification models the behavior of the system by choosing specific data domains, and by defining functions and operations on the data domains. Model-oriented specification languages include CCS, CSP, B, VDM-SL, and Z.
The basic idea of property oriented specification is to describe data structures by only giving names of data structures and basic functions, and by defining characteristic properties among the functions. The main asset of the property oriented specification style is to encourage underspecification - specifying less rather than more, and doing so as abstractly as possible - thereby avoiding the tendency to focus on how a concept is realized rather than what is required of it. Property oriented approaches include CLEAR, Larch, LOTOS, and OBJ.
A good introduction to formal methods is given by (Clarke and Wing, 1996). This paper includes an overview of notable methods and industrial applications.

Requirements Analysis

CSS is a comprehensive security system which has been developed at the Austrian Research Center in Seibersdorf (ARCS). CSS includes features like digital video recording and automatic door control. In a series of case studies, the SSD module is used as a practical example for the investigation of the benefits of formal methods (Droschl, 1999a; Droschl, 1999c). Prior to the present case study, SSD's requirements have been analyzed using IFAD's VDMTools (Elmstrom et al., 1994). By animating the specification using the IFAD Toolbox, a simple test case has been found, revealing the presence of inconsistencies. However, we failed to establish a better understanding of the nature of the contractions. Second, several minor ambiguities had to be resolved.
The results of the analysis using the IFAD Toolbox are summarized in Fig. 1. The Toolbox includes a wide range of useful features: It can animate a subset of VDM-SL constructs by applying a test suite. In an interactive mode, functions and operations may be invoked. Test coverage is a powerful feature which enables the user to determine to which extent the specification is covered by the test suite. Due to an automatic C++ code generator, the Toolbox can also be used in projects where code has to be delivered.

Assets	Liabilities
Supports animation / testing	No test case generator readily available
Statement / expression based test coverage	Exhaustive test difficult
Automatic C++ code generation	Test „low level“

Figure 1: Experiences with IFAD's Toolbox for VDM-SL for Requirements Analysis.

In order to guarantee a systematic approach to testing the specification, a test suite has to be selected in a thorough manner (Poston, 1996). However, the Toolbox does not have a test case generator. This and the huge size of the state space of SSD makes an exhaustive test difficult. According to our experience, testing supported by the IFAD Toolbox is a "low level" activity. It would be desirable to generate test cases based on general properties. In order to overcome this problem, parts of the logic of SSD were used for the generation of test cases (Droschl, 1999b).
Tool support is known to be one of the success factors in formal specification based analysis. Essentially, there are rewriting based methods (including theorem proving) and state enumeration based methods (including the application of test cases like in the IFAD Toolbox, and model checking (Holzmann, 1997)). For an overview to formal specification based analysis see for example (Clarke and Wing, 1996; NASA, 1997). In enumeration based techniques, the size of the state space is known to be the major limiting factor.
In the present case study, for the following reasons a theorem prover has been selected: first, SSD has a huge state space. As mentioned above, this makes achieving complete test coverage a difficult task. Second, we were interested in evaluating an approach which is opposed to the one employed in the first case study.
The list of theorem provers provided by Paulson[17] includes Isabelle, PVS, HOL, Otter, COQ and Z/Eves. Despite its potential bugs[18], the Prototype Verification System (PVS) (Owre et al., 1992; Rushby and Stringer-Calvert, 1995) is one of the most popular tools. Thus, it is very well supported.
PVS is a tool for writing specifications and constructing proofs. It comprises a specification language, a number of predefined theories, a theorem prover, and various utilities. The specification language of PVS supports property oriented specifications.
The Prototype Verification System combines an expressive logic with an interactive proof checker that supports top-down proof exploration and construction. In addition to its proof checker, the PVS system includes a parser, prettyprinter, and typechecker.
PVS has been applied to many applications including real-time (Kellomäki, 1997; van de Pol et al., 1998), protocols (Rajan and Fujita, 1997), avionics (Dutertre and Stavridou, 1997), space applications (Lutz and Ampo, 1994), and digital circuits (Miner and Johnson, 1996; Shostak, 1983). PVS has been used for requirements analysis before (Butler, 1996; Di Vito and Roberts, 1996; Dutertre and Stavridou, 1997; Easterbrook et al., 1998; Heimdahl and Czerny, 1996). The issue of integrating VDM and PVS for SSD is discussed by (Agerholm, 1996) and (Droschl, 1999c).

Structure of this Paper

This paper is structured as follows: section 2 gives a general presentation of SSD, followed by some advanced features. That section is concluded by a discussion of „switches“, which are the subject of this analysis. In Sec.3 a formal PVS specification for switches will be developed. In the second half of that section, the specification will be used for proving requirements properties. Section 4 discusses the insights gained in the previous section, as well as a comparison to previous work.

[17]Paulson's List of Theorem Provers `http://www.cl.cam.ac.uk/users/lcp/`
[18]PVS frequently asked questions and bug tracking database. SRI Computer Science Laboratory, Menlo Park, CA, USA. `http://pvs.csl.sri.com/bugs.html`

AN ACCESS CONTROL SYSTEM

In this section, some of SSD's features will be presented, including the notions of guard, round, station, target, and in particular switches.

Basic Features

SSD is one module of an access control system called CSS. SSD provides an interface between the guards on duty, devices like intrusion-circuits, and the operator. The basic principle is as follows: a number of guards supervise a physical plant by night. By means of rounds, the plant is divided in areas. Each round consists of a list of stations. The guards are required to visit (and "hit") stations one-by-one. This principle is illustrated by Fig. 2.

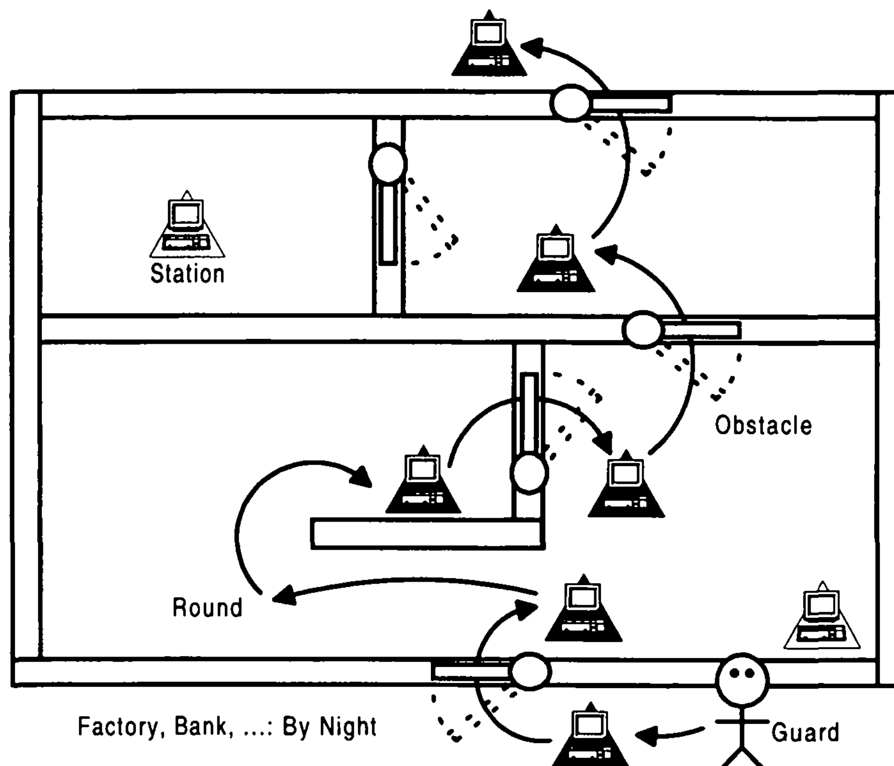

Figure 2: A site to be supervised by a guard. Stations are physical devices (terminals) which are spread over the site. Stations are shown as terminals over triangles. The area to be supervised is defined by a series of stations, called a round. Stations which shall be visited by the guard are marked in black.

The most obvious task of a guard is to visit stations. Typically, in one round, there is one guard. The system and the entirety of guards are supervised by an operator. The operator interface includes features to select, interrupt, and terminate rounds. He or she may also take stations out of order temporarily (deactivate them), or enable the guard to continue the round at any station if the round has been interrupted.

Advanced Features

SSD supports more advanced features, including the following: a guard may be on his own, and two guards may form a team. Rounds may run in parallel, and share stations. Under certain circumstances, a guard may hit a station of another guard's round. The guard's identity has to be checked whenever a station has been hit. However, there are stations which do not return the identity of the guard. If the guard is threatened by an intruder, the guard can make use of a feature called "silent operator notification". There is a range of causes which cause a round to be interrupted. Then, as part of a recovery procedure, the operator must clear the interrupt. To catch up after a (possible) delay that has occurred after an interrupt, the operator may enable the guard to skip any of the following stations. One of SSD's advanced features is the switching mechanism.

Switches

CSS is a system which may be used for night-time supervision of physical plants. At night, doors are usually locked, lights are off and intrusion circuits are activated. All of these items represent obstacles for the guards who need to circulate. A switching mechanism is concerned with unblocking and blocking of these obstacles whenever it is necessary.
Essentially, the guard's interface to the system is restricted to a location mechanism: when a guard has hit a station, the system is aware of the position of the guard in the site. Unblocking (and blocking) must take place automatically. Since such a switch triggers the submission of data to one (or many) obstacles, they are also called switch targets or simply targets. Each target may be in one of two states: blocked or unblocked. When a station gets hit by a guard, switches may cause the state of the target to change. The action of unblocking a target is called an ENTRY-switch, and blocking the target is referred to as an EXIT-switch. This principle is shown in Fig. 3.

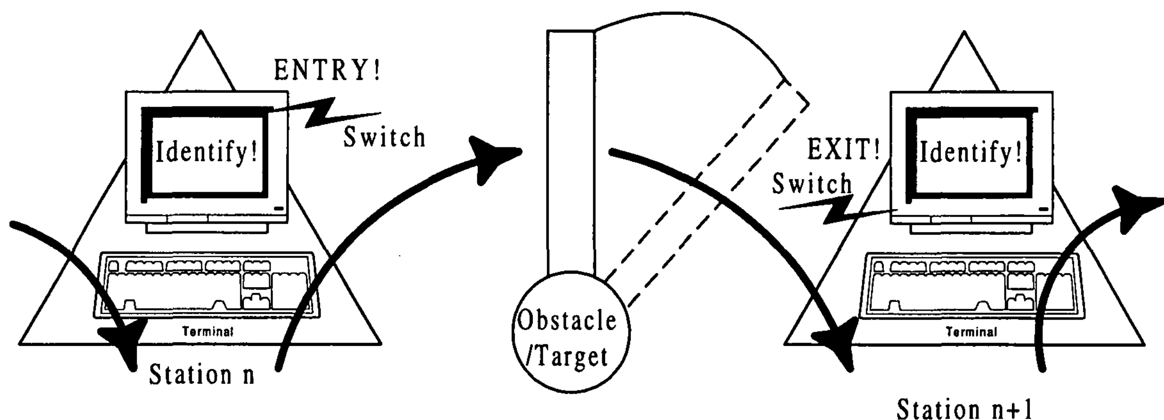

Figure 3: Part of an example round: the guard first visits some station n which causes an ENTRY switch to the given target. The guard then proceeds to station n + 1, which will send an EXIT switch.

Unfortunately, the switches principle becomes (unnecessarily) more complicated. First, parts of the requirements are given on an operational level: rather than stating the properties a solution satisfying the requirements shall have, an explicit algorithm is proposed. Second, there are an important number of special cases, which all need to be treated separately. Also, some of these cases may occur simultaneously.

Motivation for this Analysis

We will now show the test case that has indicated that there are contradictions in the requirements.
On the way through the round, the guard has to visit stations one after another. However, the operator may temporarily deactivate single stations while leaving the round functioning. As pointed out above, for a certain scenario the proposed requirements are known to fail: in a round with 5 stations, number 2 and 4 inactive, the guard may be unable to finish the round, because a target is impassable. A detailed explanation of why this is so goes beyond the scope of this paper. Clearly this given scenario should not require the operator to interfere. Thus, it is unacceptable.

System Size

The present paper reports on one of a series of case studies. These case studies and the original development are carried out in parallel. They are all based on a collection of 60 informal requirements rules, given on 10 A4 pages. The VDM specification consists of 60 A4 pages, and the PVS specification covering only the switches aspect is about 1000 lines long.

FORMAL SPECIFICATION AND ANALYSIS

The previous section has briefly introduced some of SSD's principles. In the present section, the discussion will get a little more technical. First, a formal specification will be developed. It will then be used in the proof of safety and liveness properties.

Formal Specification

We will begin by formalizing some of the concepts which are related to switches. The full PVS specification consists of about 1000 lines, including empty lines and comments. For brevity, only a few examples can be given.

Terminology

PVS supports identifiers composed of uppercase and lowercase letters as well as a wide range of special characters. In order to make the specification more easily readable, we have introduced a few restrictions. Names of types will be written in UPPERCASE_LETTERS, functions and constants in lowercase_letters. Boolean_valued_functions? have a question mark at the end of their name.
PVS functions are total. Thus, undefined values can be handled by means of supertypes and subtypes. In the specification, there are plenty of type declaration pairs where one is a subtype of the other. In these cases, the supertype has an additional question mark at the end, as, for example, in T_ST and T_ST?. Some more information on super- and subtypes will be given below.

A Glimpse on the Specification

A target is a physical device which can be in one of the two states blocked and nonblocked. In state blocked, the target is impassable for the guard. Examples for targets are light switches, doors, and intrusion circuits.

Target Identifier. T_ID? and T_ID are datatypes. They both contain all physically existing („valid“) target id's. In T_ID? there is an extra value called null_tid. Predicate t_id_notnull? holds for all valid targets. As pointed out above, T_ID is a subtype of T_ID?. Note the "+" in the declaration of T_ID? which makes sure that the type is non-empty.

```
T_ID? : TYPE+
null_tid : T_ID?
t_id_notnull? ( tid : T_ID? ) : bool = tid /= null.tid
T_ID : TYPE = (t_id_notnull?)
```

Target State. T_ST indicates the present state of the target: blocked or nonblocked. T_ST contains all values of T_ST? but T_ST_NULL. Again, T_ST is a subtype of T_ST?.

```
T_ST? : TYPE = { NONBLOCKED, BLOCKED, T_ST_NULL }
t_st_notnull? ( tst : T_ST? ) : bool = tst /= T.ST.NULL
T_ST : TYPE = (t_st_notnull?)
```

Round Identifier. Essentially, a round consists of a list of stations. Stations are devices which are visited and hit by the guard on his or her way through the site. Examples for stations are card readers, terminals, or ordinary push buttons. Similar to the target id, each round has a unique identifier of type R_ID.

```
R_ID? : TYPE+
null_rid : R_ID?
r_id_notnull? (rid : R_ID? ) : bool = rid /= null_rid
R_ID : TYPE = (r_id_notnull?)
```

Target. In the PVS specification, a target is modeled by its state and a reference to the round that has last modified that state. Each of these fields may hold null values: first, if t_lastmod? is undefined, then the model is unaware of a previous state change. However, in such a case the target may be either blocked or nonblocked. Second, a null value of the state is used to indicate that it is a non-existing target null_t.

```
T? : TYPE=      [#
                t_state? : T_ST?,
        t_lastmod? : R_ID?
#]

t_notnull? ( t: T?) : bool = t_st_notnull?(t_state?(t))
T : TYPE = (t_notnull?)

null_t : T? =       (#
        t_state?:=T_ST_NULL,
        t_lastmod?:=null_rid
        #)

t_st_mod_yet? ( t: T ) : bool = r_id_notnull?(t_lastmod?(t))
```

Target Database. The collection of known targets is modeled as a mapping from target id's to targets. T? is used as the domain of this mapping. Since in PVS mappings (they are functions) are total, some id's may map to a null target. It should now become clear why there has to be a null value for the target type T?.

```
T_DB : TYPE = [ T_ID -> T? ]
```

System State. According to the PVS model, the system state consists of the target database defined above, and a round database which is specified elsewhere using the same principles.

```
SYS_ST : TYPE =[#
        rdb:R_DB,
        tdb:T_DB
        #]
```

Event. An event of type EV is a subtype of uu_EV. Essentially, it models the transition from one system state to another, triggered by some switch.

```
uu_EV : TYPE = [#
        sys_before : SYS_ST,
trig : TR,
sys_after : SYS_ST
#]
```

Scenario. A scenario of type SCENARIO is a subtype of u_SCENARIO. A scenario is the set of all possible sequences of events. According to the PVS model, these are defined by means of axioms. This is in contrast to (Droschl, 1999b) and (Droschl, 1999c), where SSD's scenario is essentially defined as a relation between all pairs of events.

```
U_SCENARIO : TYPE = list[EV]
```

Formal Proof of Requirements

In the first part of this section, parts of SSD's formal PVS specification have been shown. Now, this specification will be used in a discussion of formal proof of requirements.
Which kind of properties can be proved in PVS ? In general, axioms and theorems. We defined a scenario as the set of all possible sequences of events. Let P(scenario) be a certain predicate over some given scenario. The following axiom formally states that property P should hold for all possible scenarios.

```
example_property:
  AXIOM FORALL (scenario:SCENARIO) : P(scenario)
```

Safety and Liveness

An example rule taken from the requirements is given in Fig. 4. The main problem of requirements validation is to find something to validate the system against (Sommerville and Sawyer, 1997). This is particularly true for

this present problem, simply because the switches' requirements lack a clear definition of which high-level properties shall hold. Instead, most of the requirements are given on a low operational level.

Rule 17:

An EXIT switch may only be sent, if the target is in state nonblocked, set by the same round.

Figure 4: One rule from SSD's requirements. It is meant to serve as an example showing that many of the requirements focus on a low operational level rather than stating which high-level properties should hold.

However, from re-reading the requirements and interviewing domain experts, the issues shown in Fig. 5 have emerged. These will serve us as a starting point for further analysis.

A missing link in the requirements

In an attempt to formalize the safety requirement given in Fig. 5, it has turned out that there is a missing link between a switch and the station that needs to be hit in order to trigger that switch. This is particularly surprising, because the operator needs to be aware of such a concept in round setup.

Liveness:
Is it possible that a guard may not finish a round,
for example, because a door remains locked ?

Safety:
Is the state of the target after the guard hit equal to its initial state ?

Figure 5: Two liveness and safety properties.

A more restricted Property

There are two main consequences: first, an assumption[19] has to be added to the requirements in order to avoid possible misunderstandings between developers and users of the system. Second, even though such an assumption could be made prior to further formal analysis, we need to find a more relaxed liveness property for SSD. Otherwise, a conclusion meaningless for the existing implementation of SSD could possibly be drawn from a too restrictive assumption.

The following axiom tst_do_not_change states that for all scenarios sc the initial target state is equal to the final one.

```
tst_do_not_change:
 AXIOM FORALL (sc:SCENARIO) :
  (
  length(sc)?=1 AND FORALL (ev:EV), (rid:R.ID) :
    member(ev,sc) AND rid=tr_rid(trig(ev))
  ) IMPLIES
  tdb(sys_before(nth(sc,0))) =
```

[19]Please recall that there are ENTRY and EXIT switches. An ENTRY switch unblocks some target/obstacle, whereas and EXIT switch blocks it. An assumption could be of the following form: if some target T_1 is assigned to some station si in terms of an ENTRY switch, the guard passes that target on the way from station s_i to station s_{i+1}. If a target T_2 is assigned to some station s_j in terms of an EXIT switch, the guard passes it between station s_{j-1} and station s_j (It is assumed that stations s_i, s_{i+1} and s_{j-1}, s_j are not deactivated).

```
  tdb(sys_after(nth(sc,length(sc)-1)))
```

length(sc)?=1 restricts the consideration to those scenarios consisting of one event or more. member(ev,sc) makes sure that event ev is indeed part of scenario sc. Round id rid is bound to the round related to the trigger by rid=tr_rid(trig(ev)). tdb(sys_before(nth(sc,i))) extracts the states of the targets before the ith event.
According to our understanding, this property should be provable. However, we did not succeed. Is it impossible to show, based on the given specification ? For several reasons, this is a difficult question to answer: first, PVS has a great variety of prover commands with plenty of options. Which ones are suitable ? Second, the very structure of the specification is known to have a major impact on the success of a proof. Third, it is not straightforward to track down the relationship between the description of PVS logic in the manuals and its actual behavior. Finally, PVS is known to have possible sources of errors.
The following two axioms can be shown to hold: tst_may_change states that there exists a scenario where the equality does not hold. In fact, it is the inverse of the previous axiom. tst_do_not_change_restrict can also be proved. It states that for all scenarios, where there is no interrupt, the initial target state is equal to the final one.

```
tst_may_change:
 AXIOM EXISTS (sc:SCENARIO) :
 (
  length(sc)?=1 AND FORALL (ev:EV), (rid:R.ID) :
   member(ev,sc) AND rid=tr_rid(trig(ev))
  ) IMPLIES
 tdb(sys_before(nth(sc,0))) /=
  tdb(sys_after(nth(sc,length(sc)-1)))

tst_do_not_change_restricted:
 AXIOM FORALL (sc:SCENARIO) :
 ( FORALL (ev:EV), (rid:R.ID) :
  length(sc)?=1 AND r_state(rdb(sys_before(ev))(rid)) /= INTERRUPTED AND
  member(ev,sc) AND rid=tr_rid(trig(ev))
  ) IMPLIES
 tdb(sys_before(nth(sc,0))) =
  tdb(sys_after(nth(sc,length(sc)-1)))
```

DISCUSSION

This case study had two main objectives: first, to analyze SSD's requirements. The results will be given in Sec. 4.1. Second, to investigate the benefits of formal proof on an industrial application. Prior to this case study, the requirements have been analyzed by means of testing supported by the IFAD Toolbox. A comparison between these alternative approaches will be given in Sec. 4.2.

Application Specific Result

Formal Properties. In this case study, a more thorough understanding of the requirements has been achieved, including a few properties which have been expressed formally by means of axioms. In a sense that their level of generality can be chosen by the developer, axioms have shown to be more flexible than single test cases.

Complementary Models. In the first case study, all of SSD's requirements have been formally specified. The IFAD Toolbox for VDM-SL has been selected, because it supports both analysis and code development. SSD's VDM specification may be used with little modifications to automatically generate C++ code. The PVS specification developed in the present case study is targeted at analysis only. Thus, the two models are complementary. They not only cover different parts of the requirements (the PVS specification is only concerned with the switches part), they are also given in a different manner: in the VDM specification, events are modeled by means of executable functions. The PVS specification makes use of axioms. Those interested in the details of the VDM specification are asked to refer to (Droschl, 1999d; Droschl, 1999b). The (non-)duality of the VDM and PVS specifications is investigated more closely in (Droschl, 1999c).

Missing Link in Requirements. This work has revealed that the requirements lack a „link“ between switches and stations. There is no statement like the following one: when the guard has previously visited station i, and an ENTRY switch is required such that the guard may reach station i + 1, then that switch shall be assigned to station i. Vice versa for EXIT switches.

Clearly, using such an assumption in this analysis could have lead to a conclusion meaningless in practice. In consequence, a more relaxed property has been used.

Proof vs. Testing

The IFAD Toolbox can help to analyze specifications by means of animation and testing. One possible scenario of using testing for requirements analysis is the following: first, identify the properties to be checked. Then, select a suitable collection of test cases for which that property should be checked. Run the test cases and evaluate the properties.

In these two case studies, testing has turned out to be much easier to handle than formal proof. However, this has been the first project where the author of this paper was involved in testing and formal proof of an application of this size. What seems to make testing a challenge is the need to cover all relevant test cases. In theory, even if just a single test case is left unconsidered, the entire result may turn out to be false (Droschl, 1999b).

As already pointed out in the first part of this section, the VDM and the PVS models are complementary: for example, in the VDM specification events are modeled by means of executable functions. In PVS, the emphasis has been on analysis aspects. Thus, non-executable concepts could be used. Initially, even the VDM specification has made use of non-executable concepts: some of the functions have been defined by means of pre- and post conditions. However, these implicit functions have subsequently been complemented by their explicit counterparts. In general, it is easier to develop an executable model from a collection of implicit function definitions, than from a specification with lots of axioms.

In a situation, where a major body of knowledge exists in VDM (the pure VDM specification without comments is about 30 A4 pages long), there are two main arguments for performing an analysis in the IFAD framework: first, there is no need for developing a new specification in an alternative environment. This could result in a major effort, partly due to differences in the logics. Even though there are efforts to extending the IFAD Toolbox with a prover (see for example (Agerholm and Frost, 1997)), at present, there is no such feature. Second, a developer is believed to be able to get started testing a specification in a reasonable time. Even though the application of single test cases may be straightforward and insightful, their choice remains the critical step.

In the first case study, a test case has been identified indicating the presence of contradictions in the requirements. In order to achieve a more thorough understanding of these contradictions, an alternative analysis approach had to be found. Parts of the requirements have been re-formalized in the PVS environment. Even though it has been shown how a VDM-SL specification can be turned into its PVS counterpart (Agerholm, 1996), the original requirements have been used as the basis for formalization: first, only parts of the requirements had to be formalized in PVS. Second, the structure of the VDM specification is targeted at making the specification executable. This was in contrast to the aims of the PVS specification. Third, the construction of a "fresh" specification out of the original requirements was believed to lead to good structure.

There are a variety of formal methods tools. Most of them have their own specification languages, incompatible with other languages. At best, there are syntactic differences only. However, this is rarely the case. Current attempts of integrating formalisms (Ambriola et al., 1997; Kramer et al., 1996) indicate that the transition from one formalism to another is a key element to the efficient application of formal methods. This has also been observed as part of our work (Droschl, 1999c).

In principle, using PVS for requirements analysis seems to be a promising approach. Theorems can be used to express arbitrarily general/abstract properties. In contrast, single test cases cover some very restricted aspect only. However, the (potential) user of PVS may face substantial obstacles. For example, PVS has shown to require a significantly longer start up time than testing. Also, proof attempts can get quite challenging from a technical perspective. On the one hand, PVS supports a variety of different types of proofs. On the other, this results in a many prover commands and a confusing user interface. The need for a practical user interface in formal methods has been recognized before (Heitmeyer, 1998).

Figure 1 summarized the main findings of the VDM analysis. In Fig. 6 the results of this analysis using PVS are given.

Assets	Liabilities
Formal Proof has shown to lead to a better understanding of the requirements than testing.	Proof is more difficult than testing.
Proof is flexible – level of generality of axioms can be chosen by the user.	Concrete form of specification may have a major impact on the success of proof.
The act of thinking about and of writing down properties may yield important insights.	The PVS user interface suffers from a large variety of commands.
	Re-formalization may result in a significant effort.

Figure 6: Experiences with PVS for Requirements Analysis.

Further Work

Model checking can become feasible once a suitable abstraction is found. This approach has not been pursued yet, because it was unclear which aspects to abstract away from. However, it would be interesting to see if our understanding of SSD's requirements could further benefit from model checking, and how these techniques relate to previous research.
An important issue is the construction and maintenance of formal specifications. Due the results of requirements analysis, the informal requirements may now be changed. Obviously, this will result in modifications of the VDM and the PVS specifications. Even though the informal requirements were thoroughly formalized, the modifications may result in a major effort. One approach to overcome this problem is given in (Droschl, 1999e). However, only model oriented specifications have been considered.

CONCLUSION

In this paper, we have presented the experiences gained in a formal methods case study in which the Prototype Verification System has been used for requirements analysis. The experiences with this formal method have been compared with previous work where the IFAD Toolbox has been used for a similar purpose.
The conclusion is that both approaches have their weeknesses and strengths - the IFAD Toolbox can be used as part of larger projects where C++ code has to be developed. However, it lacks a test case generator which can make testing a painstaking task, depending on the application.
The kind of analysis that is supported by PVS frees the user mostly from having to bother with low level details. However, a certain amount of technical knowledge is required for using this tool it effectively. Also, the variety of commands at the interface may be considered difficult to overlook. Finally, the concrete form of specification may have a major impact on the success of proof.

ACKNOWLEDGMENTS

The author of this paper would like to thank Wolfgang Herzner, Walter Kuhn, Michael Thuswald, and Erwin Schoitsch (of ARCS) for their interest in applying formal methods to an industrial application. ARCS grants the author a PhD scholarship. Peter Lucas (Software Tech., TU Graz) has provided invaluable guidance throughout this project. Pascal Fenkam, Ralf Huuck, Christian Luidolt, Walther Neuper, Christian Schinagl, and Rudi Schlatte have read earlier versions of this paper.

REFERENCES

Agerholm, S. (1996). Translating specifications in VDM-SL to PVS. In J. Von Wright, J. and J. Harisson, editors, **Proceeding of the 9th International Conference On Theorem Proving in Higher Order Logics**, volume 1125 of Lecture Notes in Computer Science, pages 1-16, Turku, Finland. Springer Verlag.

Agerholm, S. and Frost, J. (1997). An Isabelle-based theorem prover for VDM-SL. In Gunter, E. and Felty, A., editors, **Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97**, volume 1275 of Lecture Notes in Computer Science, pages 1-?? Springer-Verlag.

Ambriola, V., Cignoni, G., and Fernstroem, C. (1997). Current issues on integration. In Schafer, W., editor, Proceeding of Software process technology: **4th European workshop EWSPT '95**, Noordwijkerhout, The Netherlands, pages 197-?? Springer-Verlag. Lecture Notes in Computer Science No. 913.

Bowen, J. P. and Hinchey, M. G. (1995). Ten commandments of formal methods. **Computer**, 28(4):56-63.

Butler, R. W. (1996). An introduction to requirements capture using PVS: Specification of a simple autopilot. NASA **Technical Memorandum 110255, NASA** Langley Research Center, Hampton, VA.

Clarke, E. M. and Wing, J. M. (1996). Formal methods: State of the art and future directions. **Technical Report CMU-CS-96-178**, Carnegie Mellon University.

Davis, A. M. (1993). **Software Requirements: Objects, Functions and States**. Prentice Hall, Englewood Cliffs.

Davis, A. M. et al. (1993). Identifying and measuring quality on software requirments specification. In **Proc. Software Metrics Symp.**, pages 141-152. IEEE CS Press.

Di Vito, B. L. and Roberts, L. W. (1996). Using formal methods to assist in the requirements analysis of the Space Shuttle GPS Change Request. **NASA Contractor Report 4752**, NASA Langley Research Center.

Droschl, G. (1999a). Analyzing the requirements of an access control using VDMTools and PVS (abstract). In Wing, J. M., Woodcock, J., and Davies, J., editors, **Proceedings of FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems**, Toulouse, France, number 1709 in Lecture Notes in Computer Science. Springer Verlag.

Droschl, G. (1999b). Design and application of a test case generator for VDM-SL (extended abstract). In Fitzgerald, J. and Larson, P. G., editors, **Workshop Materials: VDM in Practice!, Part of the FM'99 World Congress on Formal Methods**, Toulouse. Springer-Verlag.

Droschl, G. (1999c). Events and scenarios in VDM and PVS. In **3rd Irish Workshop in Formal Methods**, Galway, Electronic Workshops in Computing. Springer-Verlag.

Droschl, G. (1999d). Formal specification and analysis of an access control using IFAD's VDMtools. **Technical Report IST-TEC-99-06**, Institute for Software Technology, TU-Graz, Austria.

Droschl, G. (1999e). Interactive development and maintenance of model-oriented formal specifications. **Technical Report IST-TEC-99-19**, Institute for Software Technology, Technical University of Graz, Austria. Submitted for Publication.

Dutertre, B. and Stavridou, V. (1997). Formal requirements analysis of an avionics control system. volume 23 of **IEEE Transactions on Software Engineering**, pages 267-278.

Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., and Hamilton, D. (1998). Experiences using lightweight formal methods for requirements modeling. volume 24 of **IEEE Transactions on Software Engineering**, pages 4-14.

Elmstrom, R., Larsen, P. G., and Lassen, P. B. (1994). The IFAD VDM-SL toolbox: A practical approach to formal specifications. volume 29 of **ACM SIGPLAN** Notices, pages 77-80.

Fitzgerald, J. and Larsen, P. G. (1998). **Modelling Systems - Practical Tools and Techniques in Software Development**. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK.

Heimdahl, M. P. E. and Czerny, B. J. (1996). Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In **IEEE High-Assurance Systems Engineering Workshop (HASE '96)**, Niagara on the Lake, Canada, pages 252-262.

Heitmeyer, C. (1998). On the need for "practical" formal methods. In Ravn, A. P. and Rischel, H., editors, **Proceedings of the Fifth International Symposium on Formal Techniques in RealTime and Fault-Tolerant Systems (FTRTFT'98)**, Lyngby, Denmark, volume 1486 of Lecture Notes in Computer Science, pages 18-?? Springer-Verlag.

Hoare, C. A. R. (1996). How did software get so reliable without proof ? In Gaudel, M.-C. and Woodcock, J., editors, **FME '96: Industrial Benefit and Advances in Formal Methods**, volume 1051 of Lecture Notes in Computer Science, pages 1-17. Springer-Verlag.

Holzmann, G. (1997). The model checker spin. **IEEE Trans. on Software Engineering**, 23(5):279-295. Special issue on Formal Methods in Software Practice.

Kellomäki, P. (1997). Verification of reactive systems using Disco and PVS. In Fitzgerald, J., Jones, C. B., and Lucas, P., editors, **FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods** Europe, Graz, Austria, September 1997), volume 1313 of Lecture Notes in Computer Science, pages 589-604. SpringerVerlag. ISBN 3-540-63533-5.

Kramer, J., Finkelstein, A., and Nuseibeh, B. (1996). Method integration and support for distributed software development: An overview. volume **1078 of Lecture Notes in Computer Science**, pages 115-??, Baltimore, Md. Springer-Verlag.

Lutz, R. and Ampo, Y. (1994). Experience report: Using formal methods for requirements analysis of critical spacecraft software. In **Proceedings of the 19th Annual Software Engineering Workshop**, pages 231-248, Greenbelt, MD. NASA Goddard Space Flight Center.

Miner, P. S. and Johnson, S. D. (1996). Verification of an optimized fault-tolerant clock synchronization circuit: A case study exploring the boundary between formal reasoning systems. In Sheeran, M. and Singh, S., editors, **Designing Correct Circuits**, Bastad, Sweden. SpringerVerlag Electronic Workshops in Computing.

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

Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A Prototype Verification System. In Kapur, D., editor, **11th International Conference on Automated Deduction (CADE), volume 607 of Lecture Notes in Artificial Intelligence**, pages 748-752, Saratoga, NY. Springer-Verlag.

Plat, N. and Larsen, P. G. (1992). An overview of the ISO/VDM-SL standard. volume 27 of **Sigplan Notices,** pages 76-82.

Poston, R. M. (1996). Automating Specification-Based Software Testing. **IEEE Computer Society**, Los Alamitos.

Rajan, S. P. and Fujita, M. (1997). ATM switch design: Parametric high-level modeling and formal verification. In Johnson, M., editor, **Algebraic Methodology and Software Technology, AMAST'97**, volume 1349 of Lecture Notes in Computer Science, pages 437-450, Sydney, Australia. Springer-Verlag.

Rushby, J. and Stringer-Calvert, D. W. J. (1995). A less elementary tutorial for the PVS specification and verification system. **Technical Report SRI-CSL-95-10,** Computer Science Laboratory, SRI International, Menlo Park, CA. Revised, July 1996. Available, with specification files, at http://www.csl.sri.com/csl-95-10.html.

Shostak, R. E. (1983). Formal verification of circuit designs. In Uehara, T. and Barbacci, M., editors, **Computer Hardware Description Languages**, pages 13-29. North-Holland.

Sommerville, I. and Sawyer, P. (1997). Requirements Engineering. Wiley, Chichester. van de Pol, J., Hooman, J., and de Jong, E. (1998). **Formal requirements specification for command and control systems. In Engineering of Computer Based Systems (ECBS)**, pages 37-44, Jerusalem, Israel. IEEE Computer Society.