Design and Application of a Test Case Generator for VDM-SL

Georg Droschl

Austrian Research Center Seibersdorf *and*
IST – Technical University of Graz
Münzgrabenstr. 11, 8010 Graz, Austria, Europe.
`droschl@ist.tu-graz.ac.at`, `http://www.ist.tu-graz.ac.at`

Abstract. This paper describes a pragmatic approach to the development of test cases which proved to be of value in validating a specification of an access control system. VDMTools[1] is a commercial product for the development of formal VDM-SL specifications [7]. Using VDMTools, formal specifications may be analyzed by running test cases. Test data generation is one of the most technically challenging steps of testing software. However, test case generation is not supported by VDMTools. Our interest in test case generation is purely practical: a test case generator has been developed and used in the formal development of an access control system. The test case generator draws test cases from a knowledge base containing the collections of valid sequences of events. By valid we mean those traces that are to be supported by the application. This paper briefly discusses the access control system, the generator as well as the experiences gained in the testing process.

1 An Access Control as a Case Study

CSS is a comprehensive security system which has been developed by ARCS[2]. In a case study investigating the benefits of formal methods, part of CSS (the SSD-0e, or simply SSD, module) is re-developed based on the original requirements.

System Size. Both the existing 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, consists of about 12.000 lines of PASCAL code. The size of the VDM-SL specification currently consists of about 70 A4 pages, half of which are informal text and diagrams.

1.1 Basic Functionality of SSD

In the following, SSD will briefly be introduced. See [5] for details.

[1] VDMTools is (c) 1999 by IFAD (www.ifad.dk).

[2] This work is supported by a PhD scholarship provided by ARCS - Austrian Research Center Seibersdorf, 2444 Seibersdorf, Austria, Europe (www.arcs.ac.at).

CSS includes features ranging from digital video recording to automatic door control. SSD is the module which essentially deals with access control issues. Simply put, it provides an interface between the guards on duty, the operator and the rest of the system.

The basic principle of SSD is as follows: there are a number of guards, each of which follow a certain round, for example to supervise an industrial site by night. The task of the guard is to monitor the area of the site which is covered by a particular round. On the way through the round, the guard visits (and "hits") stations one-by-one. An example round is shown in Fig. 1.

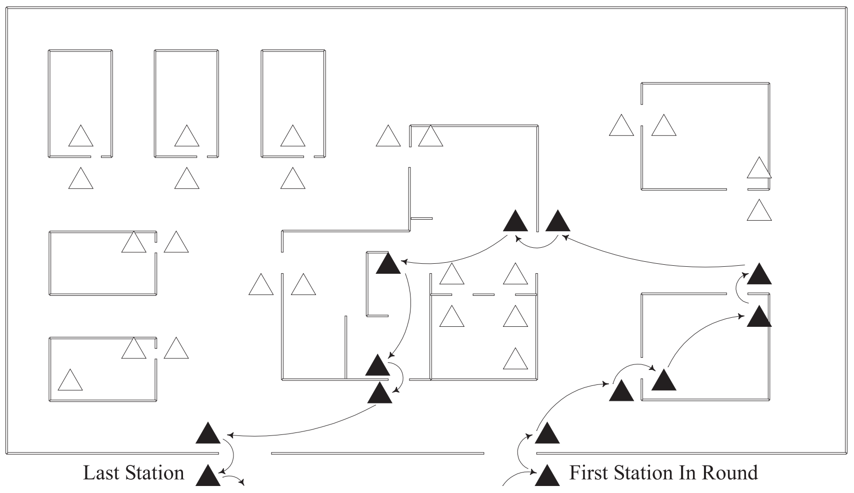

Fig. 1. An Example Round. Stations are represented by triangles. The stations to be visited by the guard on a specific round are marked in black.

Basically, for each round there is one guard. Then, there is a human operator whose task is to supervise the guards on their way through the rounds. SSD has a number of features which support the operator and the guards. For example, doors may unlock automatically after the guard has hit some station. Also, lights may be turned on automatically and intrusion circuits deactivated. Once the guard has passed that area (and hit the following station), the respective devices are put back in their prior states.

1.2 Events, Interface Functions and Scenarios

In the current formal specification of SSD, there are 34 events. In most cases, an event is triggered when the guard or the operator interact with the system. A *trace* is a particular sequence of events. A *scenario* is defined as the set of all possible traces for which the system is required to behave correctly. For one

	Event/Interf. Func.	→ Potential next event(s)
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

Fig. 2. (Most of) SSD's scenario: table of events and potential transitions (the actual scenario is made a little more restrictive by extraneous constraints). There are three groups of events, *Hit* ("H"), *Operator Interference* ("OP") and *Error* ("E"). For hit, there are three possible *selection modes*: *executing with one guard* ("X1"), *executing with two guards* ("X2") and *recording* ("RE").

single round, Fig. 2 summarizes SSD's events and the set of potential transitions between events.

In the formal specification, for each event, there is an *interface function*. Interface functions model changes in system state. Most important, each round has its own state. An interface function is shown in Fig. 3. The formal specification consists of type declarations, a state, a scenario, 34 interface functions and a number of auxiliary functions.

Each interface function has a pre condition, restricting the input parameters to those, that can be handled by that function. Using the interface functions' pre conditions, a *selector* function may choose an appropriate interface function, whenever an event has occurred. Even though, there may be more than one interface function that is considered appropriate, it has to be made sure that there is at least one.

A scenario has been incorporated into the VDM specification, to reject illegal sequences of events. The goal of this work has been to create test cases in the validation process of the specification: Make sure that all possible sequences of events are covered by the interface functions. In particular, is there an interface function covering all events, that may occur, according to the scenario ?

```
operations

1.0  $HX1norm : R\text{-}Adr \times ROUNDS \times S\text{-}hit\text{-}return \overset{o}{\rightarrow} R$
 .1  $HX1norm\,(r\text{-}adr, rounds, hr) \;\underline{\triangle}$
 .2     (dcl $r0 : R := f\,(r\text{-}adr, rounds)$,
 .3          $new\text{-}r : R := r0$;
 .4     $new\text{-}r.lasthit :=$ mk-$LastHit\,(which\text{-}hit\text{-}S\,(hr), when\text{-}hit\text{-}S\,(hr))$;
 .5     $new\text{-}r.nexthit :=$ mk-$NHS\,(next\text{-}enabled\text{-}S\,(r0, which\text{-}hit\text{-}S\,(hr))$, nil );
 .6     $new\text{-}r.stations(which\text{-}hit\text{-}S\,(hr)).ststate :=$ HIT;
 .7     return $new\text{-}r$ )
 .8  pre  $R\text{-}adr\text{-}exists\,(r\text{-}adr, rounds) \wedge$
 .9     let $r0 = f\,(r\text{-}adr, rounds)$ in
 .10    $hit\text{-}S\text{-}known\,(hr) \wedge$
 .11    $which\text{-}hit\text{-}S\,(hr) > 1 \wedge$
 .12    $which\text{-}hit\text{-}S\,(hr) < max\text{-}S\,(r0) \wedge$
 .13    $which\text{-}hit\text{-}S\,(hr) = next\text{-}hit\text{-}S\,(r0) \wedge$
 .14    $R\text{-}Mode\text{-}X1\,(r0) \wedge$
 .15    $R\text{-}running\,(r0) \wedge$
 .16    $(S\text{-}returned\text{-}id\,(hr) \;\Rightarrow\; G\text{-}on\text{-}duty\text{-}in\text{-}R\,(r0, G\text{-}id\,(hr)))$
 .17 post let $new\text{-}r = RESULT$ in
 .18      $same\text{-}R\,(new\text{-}r, f\,(r\text{-}adr, rounds)) \wedge$
 .19      $S\text{-}last\text{-}hit\,(new\text{-}r) = which\text{-}hit\text{-}S\,(hr) \wedge$
 .20      $S\text{-}time\text{-}last\text{-}hit\,(new\text{-}r) = when\text{-}hit\text{-}S\,(hr)$ ;
```

Fig. 3. An interface function in VDM-SL (one out of 34). There is both an explicit/executable part (in lines 1.2-1.7) and in implicit part (pre condition in lines 1.8-1.16 and post condition in lines 1.17-1.20).

1.3 Motivation

VDMTools [7] supports analysis of specifications by animation and test. However, in order to guarantee a systematic approach to testing the specification, the test suite has to be selected in a thorough manner [11]. Even though test case generation has been applied to VDM-SL [2, 10], there is no ready-to-use tool available. Fig. 4 shows the testing process [13] of VDMTools.

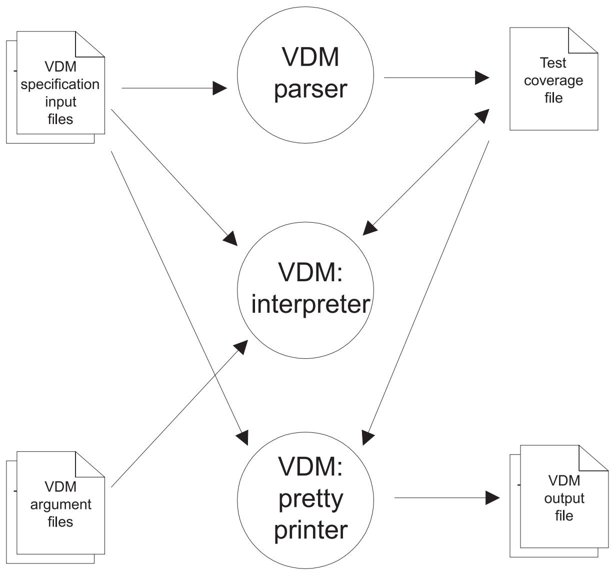

Fig. 4. The Testing Process of VDMTools

2 Test Case Generation for the Access Control

In this section, the test case generator will be briefly introduced. The generator creates a test suite which is submitted to VDMTools. The approach taken is closely related to test sequencing as discussed in [2].

The testing process aims at answering two questions:

- Does the specification support all sequences of events that are drawn from the scenario? Since a SSD interface function may only be invoked if its pre condition holds, and only valid events pass the pre condition, illegal traces may be detected by the pre condition check facility of VDMTools.
- Is the refinement is correct ? In other words, do the explicit, executable parts of the functions satisfy their implicit counterparts ? For all function invocations a post condition check is performed.

Algorithm. The algorithm of the test case generator consists of a four step process, in which a hypothetical (or "virtual") scene is created. A key issue in test case creation is that they are essentially randomly drawn. However, the user needs to offer some input, e.g. on the maximum size of the round. Figure 5 shows an example of a testfile.

```
---------- Startup ----------------------------
clear_RDB(),
exinsR( mk_simple_R(13,[
    mk_simple_S(10,true,<normal>),
    mk_simple_S(6,true,<normal>),
    mk_simple_S(2,false,<disabled>),
    mk_simple_S(3,true,<normal>)],true) ),
---------- Trace ----------------------------
OPselX1_t(13),
HX1first_t(13,mk_S_hit_return(11,nil,true,false,mk_Time(0,1,0),1,true)),
HX1norm_t(13,mk_S_hit_return(13,nil,true,false,mk_Time(0,2,0),2,true)),
HX1last_t(13,mk_S_hit_return(13,nil,true,false,mk_Time(0,3,0),4,true)),
OPselX1_t(13),
HX1first_t(13,mk_S_hit_return(11,nil,true,false,mk_Time(0,4,0),1,true)),
HX1norm_t(13,mk_S_hit_return(13,nil,true,false,mk_Time(0,5,0),2,true)),
HX1last_t(13,mk_S_hit_return(13,nil,true,false,mk_Time(0,6,0),4,true)),
OPselX1_t(13),
HX1first_t(13,mk_S_hit_return(11,nil,true,false,mk_Time(0,7,0),1,true))
---------- End of Testfile ----------------------------
```

Fig. 5. An simple example test script that has been created automatically. There is just one trace of 10 events, and a single guard. First, the created round is submitted to VDM by invoking some function that has been created for the test generator. The actual round (ID 13) consists of four stations, with number 3 disabled, which will thus be skipped by the guard. The are two cycles of the following: The operator selects the round for executing by one guard. Then, the guard hits stations 1, 2 and 4.

1. *station setup* - a collection of stations is "invented". For each station, the generator requires[3]: (1) a station identifier, (2) a flag called *codable*, which is true if and only if the station will return the identity of the guard after a hit, and (3) a flag which is set if the station has been disabled.
2. *round setup* - a collection of virtual rounds is build upon the stations created in step one. A round consists of (1) a unique identifier, (2) a list of stations, and (3) a flag telling whether the round has gone through a certain start up procedure called *recording.*
3. *guard setup* - a number of virtual night guards are invented. The guard information consists of (1) (2) two personal codes, and (3) references to

[3] Please note that in the VDM-SL specification covering the full functionality of SSD, there is much more detail.

the round the guard is assigned to, as well as (4) to the station that has previously been hit. In line 2 of Fig. 6 the respective data type definition is shown.

4. *traces setup* - traces are drawn from the scenario according to Fig. 2. For each event, a guard has to be selected as illustrated by Fig. 6.

```
1 type
2  GUARD= CODE * ALARMCODE * INROUND * LASTHIT;
3
4 fun
5  draw_nonbusy_guard() =                  (* draw one arbitrary guard *)
6  let                                  (* from created ones previously *)
7    val num=rand(!num_guards-1)+1;
8    val (cod,alcod,x,y)=nth(!guards,num-1);
9  in
10   (cod,alcod,x,y) : GUARD
11 end;
```

Fig. 6. An SML function for the choice of a "non-busy" guard. When the first station is hit, a non-busy guard has to be selected. That is, a guard that is not assigned to any of the rounds. Every guard has two codes (line 2). The "dynamic" information consists of the round the guard is assigned to, and the station that has previously been hit. Initially these fields are empty.

The programming language *Standard ML* has been selected for implementation of the test case generator [9, 12]. The test case generator consists of about 900 lines of ML code. The design of the test case algorithm and its development in Standard ML have taken about 150 hours. Some of SSD's VDM-SL data structures had to be re-build in SML.

SML has an environment that is quite mature. Thus, the development of the code has progressed quickly. However, errors on the VDM side were sometimes a little more difficult to track down, partly because its interface does not always behave as flexible as desirable. For example, a feature like SML's facility of evaluating/declaring single expressions would greatly be appreciated.

3 Conclusion

Test case generation has been considered for tests executed as part of the specification validation process of the access control system SSD. Testing SSD's specification has been carried out in two phases: First, remove the most basic errors on a function-by-function basis. Then, automatically created test cases have been applied on the specification.

In SSD there are three round selection modes: There are either (1) one or (2) two guards on a round. (3) There is a feature which allows to determine the mean time it takes a guard from one station to the next. In the following, we will discuss a testing session where most of the essential events of the main mode called "executing with one guard" have been taken into account: this corresponds to 10 of the 34 interface functions given in Fig. 2.

In the first phase, the focus was on invoking all of the interface functions to be considered in the test. Non-interface functions have only been invoked if a problem had previously occurred in the dynamic type check, the pre- and post condition checks or the check of invariants. Some of the errors found in the first phase were due to SML programming errors. On the specification side, some non-executable expressions and type problems have shown up. In 3 out of 10 interface functions mistakes in the functionality have been spotted. On average, it took about 5 minutes to track down the source of a problem.

In the second phase, a set of traces were automatically created and applied on the specification. Since the sequences of events have been selected following a random order, and the test case generator supports multiple guards and multiple rounds, even a small test suite was sufficient to reveal shortcomings of the specification. In the present project, the next step will be to re-work certain parts of the specification, and to re-run the test suite.

In principle, the taken approach has proved to be very effective in the detection of errors. On the VDMTools side, pre- and post condition checks have been used intensively.

The model of SSD has a strong state-transition character. In [1] 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 [8] is a specification language that is well-suited for modeling data. For event-based problems, usually other formalisms are employed. Still, VDM has been selected, mainly because the data aspect of SSD is an important one, and because there is good tool support.

This approach can, in principle, be used to test the implementation of the specification. The IFAD Toolbox supports C++ code generation. In order to test the C++ code, part of the test case generator would have to modified, to invoke C++ functions.

What are the shortcomings of this approach ? Currently, interface functions consist of both pre- and post conditions and of an explicit counterpart. The tests described in this paper only cover specifications which are on an operational level. It does not cover implicit functions, without an explicit part, because such functions cannot, in general, be executed.

Second, the test generation algorithm does not take into account auxiliary functions. For example, there may be sequences of operation applications in the auxiliary functions, that remain uncovered. Even though the test coverage facility provided by the Toolbox may give feedback, it is not straightforward to take this into account in test case generation program.

Third, the random event selection process requires further analysis. Currently, there is no way to measure at which degree the scenario has actually been covered by test cases.

In parallel to this work, theorem proving has been investigated for analyzing the specification (see [6] and [3]). [4] investigates the situation of using two tools (namely VDM and PVS) in one project.

4 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, Bernhard Aichernig, Brigitte Fröhlich, Andreas Gerstinger, Johann Hörl, Heinz Kammerlander, Andreas Kerschbaumer and Rudi Schlatte. John Fitzgerald has provided useful feedback on this paper.

References

1. A. Davis. *Software Requirements: Objects, Functions and States.* Prentice Hall, Englewood Cliffs, 1993.
2. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 268–284. Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.
3. G. Droschl. Analyzing the requirements of an access control using VDMTools and PVS. To appear in the proceedings of FM'99: World Congress on Formal Methods (Works in Progress: Industrial Experiences), Toulouse, France, September 20–24, Springer, 1999.
4. G. Droschl. On the integration of formal methods: Events and scenarios in PVS and VDM. To appear in the proceedings of 3rd Irish Workshop in Formal Methods, Juli 1–2, Galway, Ireland, 1999.
5. G. Droschl. Formal specification and analysis of an access control using IFAD's VDMTools. Technical Report IST-TEC-99-06, IST, TU-Graz, Austria, January 1999.
6. G. Droschl. Using PVS for requirements analysis of an access control. Technical Report IST-TEC-99-05, IST, TU-Graz, Austria, February 1999.
7. R. Elmstrøm, P.G. Larsen, and P. B. Lassen. The IFAD VDM-SL toolbox: A practical approach to formal specifications. *ACM SIGPLAN Notices*, 29(9), September 1994.
8. P.G. Larsen and N. Plat. An overview of the ISO VDM-SL standard. *ACM SIGPLAN Notices*, 27(8), August 1992.
9. L. Paulson. *ML for the Working Programmer.* Cambridge University Press, second, paperback edition, 1992.
10. J.K. Pedersen. Automatic test case generation and instatiation for VDM-SL specifications. Master's thesis, Odense University, 1998.
11. R.M. Poston. *Automating Specification-Based Software Testing.* IEEE Computer Society, Los Alamitos, first edition, 1996.
12. J.D. Ullman. *Elements of ML Programming.* Prentice-Hall, 1994.
13. The VDM Tool Group. *User Manual for the IFAD VDM-SL Toolbox.* IFAD, Odense, Danmark., 1999.