Automated Black-Box Testing with Abstract VDM Oracles

Bernhard K. Aichernig

Technical University of Graz, Institute for Software Technology (IST),
Münzgrabenstr. 11/II, A-8010 Graz, Austria.
email: `aichernig@ist.tu-graz.ac.at`,
fax: ++43 316 873 5706

Abstract. In this paper the possibilities to automate black-box testing through formal requirement specifications are explored. More precisely, the formal method VDM (Vienna Development Method) serves to demonstrate that abstract requirement models can be used as test oracles for concrete software. The automation of the resulting testing frame-work is based on modern CASE-tools that support a light-weight approach to formal methods. The specification language used is VDM-SL, but the results are easily transferred into similar model oriented methods such as B, Z or RAISE.

1 Introduction

During the last few years the interest in formal software development has been growing rapidly. One of the main reasons for this is the availability of tools to assist the developer in using these formal methods. The author, too, has contributed to the growing repertoire of automated tools by an extension of a commercial tool [5]. However, formal methods are not often applied in industrial projects, despite the growing maturity of the theories and tools, and the need of a mathematical basis [6]. Several reasons can be identified for the absence of formality in the software development process: Too many different notations have been invented, the lack of integration into informal approaches, and the strong emphasis on formal proofs.

The last point needs clarification. Of course, the formal theory and hence the possibility of conducting formal proofs is the important attribute of a formal method. However, the advantage of a formal notation to specify requirements precisely and unambigously should not be missed, even if formal proofs are not the project's focus.

Consequently, instead of promoting formal correctness proofs, we regard the automation of functional testing as the next step in a smooth integration of formal methods, to raise the level of reliability, after formal specification techniques have been introduced. To sum up, this work is a further contribution to the lately propagated light-weight approach to formal methods [19].

The ISO-standardized Vienna Development Method (VDM) [18, 11, 20] serves to demonstrate how a well-established formal method supports the automation of testing. VDM is one of the most widely used formal methods, and it can be applied to the construction of a large variety of systems. It is a model oriented method, i.e. its formal descriptions (specifications) consist of an explicit mathematical model of the system being constructed. Furthermore, VDM is supported by CASE-tools and even allows an integration into informal methods like UML or Structured Analysis, and so does our testing approach.

Previous work has shown how test-cases may be derived from formal specifications [9]. However, little attention had been given to the fact that formal models of software requirements are inherently abstract in the sense that detailed design decisions are not included. Consequently, the test-cases, derived (or even generated) from such an abstract model, are abstract too, and thus inappropriate for a direct automatic test of a target system. For that reason, a mapping between abstract and concrete test data is required.

The presented framework focuses on the usage of formal requirements specifications as test oracles for concrete implementations. The approach is based on the formal definition of abstraction as a homomorphism, called the retrieve function, which maps the concrete level into the abstract. If the retrieve function is implemented and the post-condition is executable, then the model may serve as a test oracle and specification as well. The approach is not limited to VDM, but can also be applied in other model oriented methods like B [1], Z [25] or RAISE [13].

After this introduction, the following Section 2 describes the general approach of using specifications as test oracles. Then, Section 3 explains three possibilities to automate the approach by using a commercial tool, the IFAD VDM-SL Toolbox. Finally, Section 4 contains some concluding remarks and the identification of possible directions to future work.

2 From Formal Specifications to Oracles

The Vienna Development Method provides the two needed concepts in order to support the presented testing process. First, the formal specification itself, which precisely defines what a function or operation should compute. Second, the concept of data-reification, that provides a formal definition of abstraction. In the following, both concepts are explained and their role in our testing approach will be clarified.

2.1 VDM-SL Specifications as Oracles

As already mentioned, in VDM a system is specified in terms of abstract mathematical models. VDM-SL, the general purpose specification language of VDM, provides mathematical objects, like sets, sequences, maps etc., to model a system's state. The functionality is formulated by imperative operations, which manipulate these abstract state representations or applicative functions. Two styles can be distinguished to define functionality: implicit and explicit definitions. An implicit operation defines *what* should happen by pre- and post-conditions. The pre-condition is a logical constraint on the input stating what must hold that the functionality is defined. The essential part is the post-condition, a relation between the input, the old state, the output and the new state.

The following example is inspired by an industrial project in which a voice communication system for air traffic control has been formally specified [16]. Here, an abstract view of the system's radio switch is modeled. The switch assigns frequencies to operator positions on which communication may be established. The VDM-SL model refers to frequencies and operator positions by the identifiers *FrqId* and *OpId*. The relation is modeled by a finite mapping from operator positions to a set of frequencies. A map can be interpreted as a two-column tabular whose left- and right-side entries can be accessed by the domain (*dom*) and range (*rng*) operators.

The set of frequencies represents frequency coupling, which means that communication is forwarded to all frequencies in the set. A requirement of the switch is that a frequency must not belong to two different coupling sets. This is expressed by means of a data-invariant, stating that for all two frequency sets, with more than one element, they may not have frequencies in common.

$$Switch\text{-}a = OpId \xrightarrow{m} FrqId\text{-}\mathsf{set}$$

$$\mathsf{inv}\ s \triangleq \forall fs1, fs2 \in \mathsf{rng}\ s \cdot \mathsf{card}\ fs1 > 1 \wedge \mathsf{card}\ fs2 > 1 \ \Rightarrow\ fs1 \cap fs2 = \{\}$$

A function *couple-frequency*, defined by pre- and post-conditions, adds frequencies to an operator position. The pre-condition says that if the operator position has already a frequency associated, then no frequency set with a cardinality greater than two must exist that already contains the frequency to be added. This would lead to violation of the system's invariant.

The post-condition establishs the following relation between the input (*f*, *op* and *s*) and the resulting output *r*: If the operator position has already a frequency associated, then the resulting map equals the old one with the input frequency added to the existing set of frequencies of the operator position. This is expressed by overriding (†) a map entry. In case of a new operator position, the result equals the old switch with the new map entry.

```
$couple\text{-}frequency\ (f : FrqId, op : OpId, s : Switch\text{-}a)\ r : Switch\text{-}a$
$\mathsf{pre}\ \ op \in \mathsf{dom}\ s \ \Rightarrow\ \neg\,\exists fs \in \mathsf{rng}\ s \cdot f \in fs \wedge \mathsf{card}\ fs > 1$
$\mathsf{post\ if}\ op \in \mathsf{dom}\ s$
    $\mathsf{then}\ r = s \dagger \{op \mapsto s\,(op) \cup \{f\}\}$
    $\mathsf{else}\ r = s \dagger \{op \mapsto f\}$
```

From a testers perspective, the post-condition serves as a test oracle. In general, an oracle is any program, process or data that specifies the expected outcome of a set of tests as applied to a tested object [7]. Here, the oracle is a predicate, a Boolean function, which evaluates to true if the correct output is returned, with the premise that the pre-condition holds. The signature of the post-condition oracle above is:

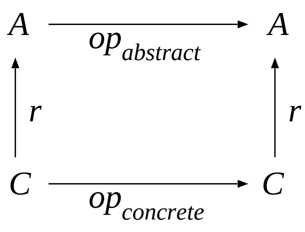
$$post\text{-}couple\text{-}frequency : OpId * FrqId * Switch\text{-}a * Switch\text{-}a \to \mathbb{B}$$

The two arguments of type *Switch-a* define the relation between the old and new value of the switch. Note that pre-conditions define the input values for 'good' tests, where valid outputs can be expected.

However, the system's model and consequently the oracle is abstract and cannot be directly used as a test-oracle for a test on implementation level. What is needed, is a link between the abstract level and the concrete level. In VDM, and other formal methods, this link is provided by a precise definition of abstraction.

2.2 The Definition of Abstraction

The M in VDM is based on a refinement of an abstract specification to a more concrete one, which usually is carried out in several steps. Refining the data model is called data-reification, which is correct if it establishs a homomorphism between the abstract and refined, more concrete, level. In Figure 1 our notion



$$\begin{tikzcd}
A \arrow[r, "op_{abstract}"'] & A \\
C \arrow[u, "r"'] \arrow[r, "op_{concrete}"'] & C \arrow[u, "r"']
\end{tikzcd}$$

Fig. 1. The morphism of abstraction.

of abstraction is represented by a commuting diagram, where operations (*op*) are viewed as functions from input to output states. The abstract operation $op_{abstract}$ manipulates an abstract state representation of type A. The concrete operation $op_{concrete}$ incorporates detailed design decisions and maps a refined input state to an output state. Examples for such data-refinement would be to implement a set through a balanced binary-tree, or the other way round, to view a data-base system as a set of data. The relationship between the abstract and the concrete is defined by a a function r mapping a concrete state to its abstract counterpart. The diagram shows that the retrieve function r is a homomorphism for which the following formula holds:

$$\forall\ in : C \cdot op_{abstract}(r(in)) = r(op_{concrete}(in))$$

Using an implicit function definition on the abstract level, the formula for a correct implementation (refinement) is:

$$\forall\ in : C \cdot pre\text{-}op_{abstract}(r(in)) \ \Rightarrow\ post\text{-}op_{abstract}(r(in), r(op_{concrete}(in)))$$

This simply means that the abstract post-condition must hold for the concrete input and output, mapped to the abstract level by r, if the pre-condition is satisfied.

Modern tools allow the interpretation of explicit VDM-SL definitions. If the retrieve function and the post-condition predicate are defined explicitly, the formula above can be evaluated and thus provides a testing frame-work inside VDM-SL. However, to test functionality implemented in a programming language like C++, the language gap between programming languages and the specification language has to be overcome. In the following it is shown how modern tools provide these bridges and help to automate the approach.

3 Automated Test Evaluation

The combination of the notion of abstraction and the possibility to use specifications as test oracles leads to a testing framework with abstract oracles. Modern tools like the IFAD VDM-SL Toolbox [17] allow the interpretation or even code-generation of such pre- and post-conditions which leads to an automated test evaluation through post-condition oracles. In Figure 2 the data-flow of the new testing scenario is presented. An implementation is executed with a concrete

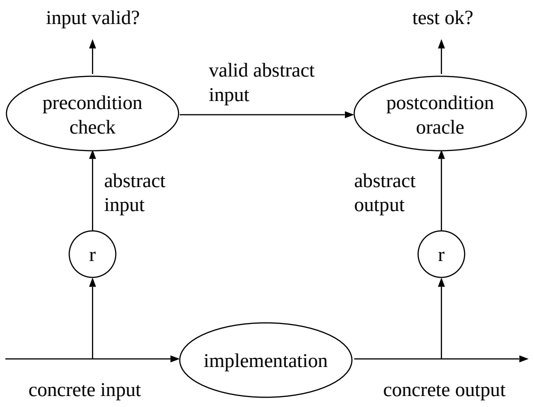


Fig. 2. DFD of the testing scenario.

test input (state). An implemented retrieve function maps the concrete input and output to its abstract representations. A pre-condition check validates the input and feeds it into the oracle which checks the relation to the produced output. If the post-condition evaluates to true, the test passed. The advantage of the approach is the automation of black-box testing by the usage of structural test-data. Another possibility would be to code the reverse of r in order to test with abstract test-data derived from the specification.

In the following, the possibilities for automation by using the IFAD tools are explained in more detail.

3.1 Code-Generation of Oracles

The IFAD Toolbox provides a C++ code-generator, which translates explicit VDM-SL specifications to C++ source code, using a VDM library. With this tool the post-condition functions, like in our Switch example, can be automatically translated to C++ . Even the pre-condition with its quantors over finite sets can be code-generated. Below the generated C++ code for *post-couple-frequency* is shown:

```
Bool post_couple_frequency(int f, int op, Map s, Map r) {
  Bool varRes;
  Bool cond;
  cond = (Bool) s.Dom().InSet(op);
  if (cond.GetValue()) {
    Map var2;
    Map modmap;
    Set tmpVar2;
    Set var2;
    var2 = Set().Insert(f);
    tmpVar2 = (Set) s[op];
    tmpVar2.ImpUnion(var2);
    modmap = Map().Insert(op, tmpVar2);
    var2 = s;    var2.ImpOverride(modmap);
    varRes = (Bool) (r == var2);
  }
  else {
    Map var2;
    Map modmap;
    modmap = Map().Insert(op, f);
    var2 = s;    var2.ImpOverride(modmap);
    varRes = (Bool) (r == var2);
  }
  return varRes;
}
```

What remains to implement manually is the retrieve function. Special C++ generator functions in the VDM library facilitate the conversions of C++ types into VDM objects.

3.2 Interpretation of Oracles with the Dynamik Link Facility

Another approach is to use the Toolbox interpreter in order to evaluate the oracle inside VDM-SL. The dynamik link facility may be used to call the C++ functions to be tested from inside the Toolbox [12]. The dynamik link facility enables a VDM-SL specification to execute parts written in C++, during interpretation.

It is called dynamik link facility because the C++ code is dynamically linked together with the Toolbox.

In order to access the implementation, a *dlmodule* has to be defined which contains the function's declaration. For our example that would be

$$
\begin{array}{l}
\textsf{dlmodule}\ \mathit{CPP} \\
\quad \textsf{exports} \\
\qquad \textsf{functions}\ \mathit{couple\text{-}frequency\text{-}ext} : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \xrightarrow{m} \mathbb{Z}\textsf{-set} \to \mathbb{Z} \xrightarrow{m} \mathbb{Z}\textsf{-set} \\
\textsf{end}\ \mathit{CPP}
\end{array}
$$

, if we model frequencies and operator positions as integers ($\mathbb{Z}$). In order to test *couple-frequency* in this case two functions are to be implemented. One mapping the input to the implementation representation and calling *couple-frequency*, and the retrieve function which converts the output back to the abstract VDM-SL data-type.

With this approach abstract VDM-SL test-cases are fit into the implementation. Hence, here the following must hold for a correct implementation $op_{concrete}$:

$$\forall\ in : A \cdot pre\text{-}op_{abstract}(in) \;\Rightarrow\; post\text{-}op_{abstract}(in, r(op_{concrete}(rep(in))))$$

, where the function $rep : A \to C$ maps the abstract input to its concrete representation.

3.3 CORBA Oracles

A third elegant way to bridge the language gap is CORBA, the Common Object Request Broker Architecture [22]. The VDM-SL Toolbox provides a CORBA based API through which a program can access the Toolbox [27]. The CORBA interface of the VDM-SL Toolbox is defined in IDL, the interface definition language of CORBA. The language mapping is done by the IDL interface through which the full functionality of a running toolbox may be invoked.

This includes the loading of a specification and the starting of the interpreter. With this technique a post-condition oracle can be accessed and evaluated through the API. Since CORBA is a distributed object architecture the Toolbox may even run on a different (test-)server providing the oracles. Since CORBA IDL converts different programming languages, the test approach can be applied to as many programming languages as IDL mappings have been implemented, e.g. C++ and Java.

4 Concluding Remarks

In this paper we have presented an approach for automated test evaluation through VDM-SL oracles. We have shown the general strategy and presented more details how the resulting test approach can be automated. The three presented solutions for automation are based on the commercial tool IFAD VDM-SL Toolbox.

To our present knowledge, only one automated black-box testing approach has been based on an explicit mapping from an implementation level to a formally specified and abstract post-condition oracle [8]. In this early work, the post-condition oracle had to be translated manually into Prolog. Of course, other formal model oriented approaches to testing have been published, but they differ in two aspects: First, the focus rather lies on test-case generation, than on oracle generation and automated test evaluation [21, 9, 26, 15, 24, 10, 2, 23]. Hence, the abstraction problem is not considered at all. Second, explicit specifications serve as active oracles which calculate and compare the specified output to the program under test [14, 3, 28]. In contrast to our passive oracles, these solutions cannot handle non-deterministic results.

Our idea has been presented in [4] the first time. This work could also be extended with test-case generation techniques in order to automate the whole test process. However, the automated test evaluation especially supports random testing.

In future we envisage an instrumentation of objects with post-condition oracles. Objects can be instrumented through inheritance without changing the actual code. Then, testable objects inherit the functionality from its superclass and provide additional retrieve and oracle functions as testing methods.

We feel that the application of both, formal specifications and formal development tools to testing, as presented here, will be a powerful combination. However, more case studies are needed in order to evaluate the approach formally.

Acknowledgments

The author would like to thank IFAD's team for their support concerning the VDM-tools. Many thanks belong to Peter Lucas and the members of our formal methods group for the stimulating discussions.

References

1. J.-R. Abrial. *The B-Book, Assigning programs to meanings.* Cambridge University Press, 1996. ISBN 0521 49619 5(hardback).
2. Lionel Van Aertryck. *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels.* PhD thesis, Université de Rennes, January 1998.
3. Sten Agerholm, Pierre-Jean Lecoeur, and Etienne Reichert. Formal specification and validation at work: A case study using VDM-SL. In *Proceedings of Second Workshop on Formal Methods in Software Practice, Florida, Marts.* ACM, 1998.
4. Bernhard K. Aichernig. Automated requirements testing with abstract oracles. In *ISSRE'98: The Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany,* pages 21–22, IBM Thomas J.Watson Research Center, P.O.Box 218, Route 134, Yorktown Heights, NY, USA, November 1998. Ram Chillarege. ISBN 3-00-003410-2.

5. Bernhard K. Aichernig and Peter Gorm Larsen. A proof obligation generator for VDM-SL. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, 1997.
6. Bernhard K. Aichernig and Peter Lucas. Softwareentwicklung — eine Ingenieursdisziplin!(?). *Telematik, Zeitschrift des Telematik-Ingenieur-Verbandes (TIV)*, 4(2):2–8, 1998. ISSN 1028-5068.
7. Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 2nd edition, 1990.
8. R.E. Bloomfield and P.K.D. Froome. The application of formal methods to the assessment of high integrity software. *IEEE Transactions on Software Engineering*, SE-12(9):988–993, September 1986.
9. Jeremy Dick and Alain 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.* Springer-Verlag, April 1993.
10. Michael R. Donat. Automating formal specification-based testing. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97:Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 833–847. Springer-Verlag, April 1997.
11. John Fitzgerald and Peter Gorm Larsen. *Modelling Sytems, Practical Tools and Techniques.* Cambridge University Press, 1998.
12. Brigitte Fröhlich and Peter Gorm Larsen. Combining VDM-SL specifications with C++ code. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME96, Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science, pages 179–194. Springer, March 1996.
13. Chris George et al. *The Raise Development Method.* The BCS Practitioner Series. Prentice Hall, 1995.
14. Andrew Harry. The value of reference implementations and prototyping in a formal design and testing methodology. Report 208/92, National Physical Laboratory, Queen's Road, Teddington, Middelsex TW11 0LW, UK, October 1992.
15. Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating test case generation from Z specifications with Isabelle. In *ZUM'97*, 1997.
16. Johann Hörl. Formal specification of a voice communication system used in air traffic control. Master's thesis, Institute for Software Technology (IST), Technical University Graz, Austria, December 1998.
17. IFAD. IFAD's homepage. http://www.ifad.dk/.
18. Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.
19. Cliff B. Jones. Formal methods light: A rigorous approach to formal methods. *IEEE Computer*, 29(4):20–21, April 1996.
20. P. G. Larsen, B. S. Hansen, H. Bruun, N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996. International Standard ISO/IEC 13817-1.
21. Janusz Laski. Data flow testing in STAD. *The Journal of Systems and Software*, 12(1):3–14, 1990.
22. OMG. The common object request broker architecture and specification, revision 2.0. Technical report, OMG, 1996.

23. Jesper Pedersen. Automatic test case generation and instantiation for VDM-SL specifications. Master's thesis, Department of Mathematics and Computer Science, Odense University, September 1998.
24. J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Jounal*, 19:53–77, 1997.
25. J. M. Spivey. *The Z Notation.* Series in Computer Science. Prentice-Hall, 1989.
26. Philip Alan Stocks. *Applying formal methods to software testing.* PhD thesis, The Department of computer science, The University of Queensland, 1993.
27. Ole Storm. The VDM Toolbox API users guide. Technical report, IFAD, 1998.
28. H. Treharne, J. Draper, and S. Schneider. Test case preparation using a prototype. In *B'98 — Second B-Conference*, 1998.