Where Specification and Programming Meet

Extended Abstract

Klaus Havelund*

Jet Propulsion Laboratory, California Institute of Technology, USA

Abstract. We argue that a modern programming language such as SCALA has achieved a level of succinctness, which makes it suitable for program/systems specification, hence able to take the role that early very elegant specification languages, way ahead of their time, served. We illustrate this by comparing the VDM++ specification language with SCALA. We furthermore illustrate SCALA's potential as a specification language by augmenting it with a combination of parameterized state machines and temporal logic, defined as a library, thereby forming a very expressive and convenient runtime verification framework.

1 VDM and its Derivatives

Formal methods generally refer to "mathematically-based techniques for the specification, development and verification of software and hardware systems" [18]. The field covers such topics as specification logics, syntax and semantics, proof systems, industrial strength specification languages, theorem proving, and model checking. Amongst one of the earlier contributions was VDM (Vienna Development Method) [9, 10, 22, 23, 30] and its associated specification language META-IV [9]. META-IV is a so-called *wide spectrum* specification language, including as a subset an executable language comparable to the combination of an imperative programming language and a functional programming language, with data types and pattern matching, and with built-in collections such as finite sets, lists and maps. In addition META-IV early on offered design-by-contract concepts, such as pre/post conditions and invariants, later found in the EIFFEL programming language [15]; predicate subtypes (for example natural numbers as a subset of integers), and general first order predicate logic in the form of universal and existential quantification over infinite as well as finite sets - permitted as Boolean expressions. Indeed a very impressive and forward looking language.

A VDM language standard was subsequently produced in the form of VDM-SL (VDM Specification Language), which combined the so-called "British style"

* Part of the research described in this publication was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Government sponsorship acknowledged.

(which focused on using only pre/post conditions to specify functions and operations) and the "Danish style" (which embraced the executable programming language like subset). Two more derivations were later created in RAISE [19] and VDM++ [17]. RAISE took its starting point in VDM, but followed an algebraic view, where a module consists of a signature and a set of axioms over the names introduced in the signature, including names denoting constants, functions, operations with side-effects, and processes. Derived forms were introduced reflecting the classical VDM definitional style. VDM++ took a less drastic approach, "just" adding object oriented constructs (classes and objects) as well as concurrency to the more classical VDM notation. In this sense VDM++ is interesting since of these systems, it gets the closest to a modern programming language due to its integration of object oriented features. Our comparison is therefore between VDM++ and SCALA.

2 Scala as a Modeling Language

The general experience gained by working with VDM is that of abstraction, elegance and convenient notation. So why are we not all using VDM? A characteristic of VDM, and most formal methods, is that specification fundamentally is considered different from programming, in spite (in this case) of the great overlap with respect to language constructs. For example, META-IV contains as a subset an executable kernel, which is isomorphic to a full blown programming language. In spite of this overlap, the typical use of VDM is to write a specification, and then subsequently either (re)program the system manually (observing the specification), or translate the specification into a program using a special compiler. Many users are uncomfortable with such a translation.

However, a modern programming language such as SCALA [27] offers language constructs, which makes it a healthy alternative for writing abstract high-level models. SCALA offers a uniform combination of object-oriented and functional programming. Furthermore, it supports definition of internal DSLs (Domain Specific Languages) through a set of innovative language constructs. SCALA programs are usually very succinct compared for example to JAVA programs, and have the script-like flavor that PYTHON [26] programs have, but with the static typing that JAVA offers, and compiling to the JVM. Sets, lists and maps are part of the SCALA library, as is the case in JAVA. However, with better notation for manipulating these data structures. We discuss the relationship between VDM++ and SCALA, illustrating differences and similarities on a number of examples as well as on a construct-by-construct basis. We summarize some of those library additions that would be needed in order to write VDM++ like models. Some of these have been suggested elsewhere, for example design by contract for SCALA [25]. We also discuss possible modifications to SCALA, inspired by VDM.

3 Runtime Verification

The field of Runtime Verification (RV) has seen an impressive growth over the last decade [24, 13, 29, 14, 28, 11, 1, 16], including our own work [21, 20, 2, 12, 7, 6, 8, 3]. RV generally is concerned with processing program/system executions with the purpose of testing, understanding, and/or influencing their behavior. An important RV task consists of checking such executions against formal specifications. Various behavior oriented specification notations have been studied for this purpose. These include state machines, temporal logics, regular expressions, grammar systems and rule-based systems. These logics have the advantage of making certain properties easy to express in a succinct manner. More recently focus has been on augmenting such logics with data parameterization, in order to handle for example first order temporal logic. However, properties to be checked are occasionally more complicated than can be handled with any one of these systems in a convenient manner. In the extreme case, one may need to "dive" and program an observer using traditional programming techniques. In some cases the specification needed is a *reference implementation*: a simple abstract, functionally correct, but perhaps inefficient, program. Scala's quality as a specification language makes it ideal for this purpose, especially when augmented with forms of temporal logic suitable for the simple cases. We illustrate this idea with the Scala DSL named TraceContract [4, 5] for writing parameterized state machines, which allows anonymous states, thereby allowing a combination of state machines, temporal logic and code.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*. ACM Press, 2005.
2. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
3. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.
4. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *17th International Symposium on Formal Methods (FM'11), Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.
5. H. Barringer, K. Havelund, E. Kurklu, and R. Morris. Checking flight rules with TraceContract: Application of a Scala DSL for trace analysis. In *Scala Days 2011, Stanford University, California*, 2011.
6. H. Barringer, K. Havelund, D. Rydeheard, and A. Groce. Rule systems for runtime verification: A short tutorial. In *Proc. of the 9th Int. Workshop on Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 1–24. Springer, 2009.
7. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125. Springer, 2007.

8. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.
9. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
10. D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982. ISBN 0-13-880733-7.
11. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.
12. M. D'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *Workshop on Dynamic Program Analysis (WODA'05)*, volume 30(4) of *ACM Sigsoft Software Engineering Notes*, pages 1–7, 2005.
13. D. Drusinsky. The temporal rover and the ATG rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
14. D. Drusinsky. *Modeling and Verification using UML Statecharts*. Elsevier, 2006. ISBN-13: 978-0-7506-7949-7, 400 pages.
15. Eiffel. http://www.eiffel.com.
16. Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime verification of safety-progress properties. In *Proc. of the 9th Int. Workshop on Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 40–59. Springer, 2009.
17. J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.
18. Formal Methods Wikipedia. http://en.wikipedia.org/wiki/Formal_methods.
19. C. George, P. Haff, K. Havelund, A. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England, 1992.
20. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.
21. K. Havelund and G. Rosu. Monitoring programs using rewriting. In *16th ASE conference, San Diego, CA, USA*, pages 135–143, 2001.
22. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990. ISBN 0-13-880733-7.
23. C. B. Jones and R. C. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990. ISBN 0-13-880733-7.
24. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *PDPTA*, pages 279–287. CSREA Press, 1999.
25. M. Odersky. Contracts for Scala. In *Runtime Verification - First Int. Conference, RV'10, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *LNCS*, pages 51–57. Springer, 2010.
26. Python. http://www.python.org.
27. Scala. http://www.scala-lang.org.
28. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 109–124. Elsevier, 2006.
29. V. Stolz and F. Huch. Runtime verification of concurrent Haskell programs. In *Proc. of the 4th Int. Workshop on Runtime Verification (RV'04)*, volume 113 of *ENTCS*, pages 201–216. Elsevier, 2005.
30. VDM. http://en.wikipedia.org/wiki/Vienna_Development_Method.