Internal versus External DSLs for Trace Analysis[⋆] Extended Abstract

Howard Barringer[1] and Klaus Havelund[2]

[1] School of Computer Science, University of Manchester, UK
Howard.Barringer@manchester.ac.uk

[2] Jet Propulsion Laboratory, California Institute of Technology, USA
Klaus.Havelund@jpl.nasa.gov

Abstract. This tutorial explores the design and implementation issues arising in the development of domain-specific languages for trace analysis. It introduces the audience to the general concepts underlying such special-purpose languages building upon the authors' own experiences in developing both external domain-specific languages and systems, such as EAGLE, HAWK, RULER and LOGSCOPE, and the more recent internal domain-specific language and system TRACECONTRACT within the SCALA language.

Keywords: run-time verification, trace analysis, domain-specific language (DSL), external DSL, internal DSL, TRACECONTRACT, SCALA

Domain-specific languages (DSLs) are simply special-purpose programming languages and, as such, are far from being a new concept; for example in the field of text processing one can find COMIT [16] in the 1950s, which led to SNOBOL [8] in the 1960s, then on to the likes of AWK [1], Perl [15], etc. The naming of such special-purpose programming languages as DSLs is a more recent development that has come about through the field of domain-specific modelling. Fowler [9] presents a rather comprehensive volume on DSLs and their application.

Within the field of run-time verification, as in formal methods in general, specification languages and logics have usually been created as separate, standalone, languages, with their own parsers; these are usually referred to as *external DSLs*. We have ourselves developed several external DSLs for trace analysis, e.g. EAGLE [2], HAWK [7], RULER [6], LOGSCOPE [3], and observe two key points: (i) once a DSL is defined, it is labourious to change or extend it later; and (ii) users often ask for additional features, some of which are best handled by a general purpose programming language. An alternative approach is to try to use a high level programming language that can be augmented with support for temporal specification. These are usually referred to as *internal DSLs*. An internal DSL is really just an API in the host language, formulated using the language's own primitives. Recently, we chose to develop an internal DSL, TRACECONTRACT [4], for trace analysis in SCALA [12]. Indeed, SCALA is particularly well

[⋆] Part of the work to be covered in this tutorial was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

suited for this because of (i) the language's in-built support for defining internal DSLs, and (ii) the fact that it supports functional as well as object oriented programming. A functional programming language seems well suited for defining an internal DSL for monitoring, as also advocated in [13] in the case of HASKELL [14]. An embedding of an internal DSL may be termed as *shallow*, meaning that one makes the host language's constructs part of the DSL, or it may be termed as *deep*, meaning that a separate internal representation is made of the DSL (an abstract syntax), which is then interpreted or compiled as in the case of an external DSL. A shallow embedding has disadvantages, for example not being easily analyzable. In [10] it is argued that the advantage of a deep embedding is that *"We 'know' the code of the term, for instance we can print it, compute its length, etc"*, whereas the advantage of a shallow embedding is that *"we do not know the code, but we can run it"*. Generally, the arguments *for* an internal DSL are: limited implementation effort due to direct executability of DSL constructs, feature richness through inheriting the host language's constructs, and tool inheritance, i.e. it becomes possible to directly use all the tool support available for the host language, such as IDEs, editors, debuggers, static analyzers, and testing tools. In summary, the arguments *against* an internal DSL are: (i) lack of analyzability, i.e. one cannot analyze internal DSLs without working with the usually complex host language compiler, which can then have consequences for performance and reporting to users, and (ii) high complexity of language, i.e. one now has to learn and use the bigger host programming language, which may exclude non-programmers from using the language, and which may lead to more errors. Our main observation is, however, that feature richness and adaptability are both very attractive attributes. To some extent, adaptability "solves" the problem of what is the right logic for runtime monitoring. An additional argument is that often one wants to write advanced properties for which a simple logic does not suffice, including counting and collecting statistics. In a programming language this all becomes straightforward. The use of SCALA, whose functional features can be considered as a specification language in its own right, provides further advantage.

In this tutorial, we will introduce the audience to the above issues in the design of DSLs, both external and internal, in the context of run-time verification. In particular, we will use our own experience with the development of RULER, as an external DSL, and TRACECONTRACT, an internal DSL, to show advantages and disadvantages of these approaches. The tutorial will be presented through a series of examples, it will show how an internal DSL can be quickly implemented in SCALA (within the tutorial session), and it will demonstrate why TRACECONTRACT is being used for undertaking flight rule checking in NASA's LADEE mission [11, 5].

References

1. A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK programming language*. Addison-Wesley, 1988.
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, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.
7. M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.
8. D. Farber, R. Griswold, and I. Polonsky. SNOBOL, A string manipulation language. *Jounral of the ACM*, 11(1):21–30, 1964.
9. M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley, 2010.
10. F. Garillot and B. Werner. Simple types in type theory: Deep and shallow encodings. In *20th Int. Conference on Theorem Proving in Higher Order Logics (TPHOLs'07), Kaiserslautern, Germany.*, volume 4732 of *LNCS*, pages 368–382. Springer, 2007.
11. Lunar Atmosphere Dust Environment Explorer. http://www.nasa.gov/mission_pages/LADEE/main.
12. Scala. http://www.scala-lang.org.
13. 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.
14. The Haskell Programming Language. http://www.haskell.org/haskellwiki/Haskell.
15. The Perl Programming Language. http://www.perl.org.
16. V. H. Yngve. A programming language for mechanical translation. *Mechanical Translation*, 5(1):25–41, 1958.