Specification Facets for More Precise, Focused Documentation

Gary T. Leavens and Clyde Ruby

TR #97-04
January 1997

Keywords: reuse, formal specification languages, metaspecification, facets, expressiveness.

1993 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — Languages; F.3.1 [*Logics and Meaning of Programs*] Specifying and verifying and reasoning about programs — Assertions, invariants, pre- and post-conditions, specification techniques.

This is a short position paper that will appear in the proceedings of the *Eighth Annual Workshop on Software Reuse*, March 23-26, 1997, Columbus, Ohio.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Specification Facets for More Precise, Focused Documentation

Gary T. Leavens

Iowa State University
Department of Computer Science
226 Atanasoff Hall
Ames, Iowa 50011-1040 USA
Tel: (515) 294-1580
Fax: (515) 294-0258
Email: leavens@cs.iastate.edu

Clyde Ruby

Iowa State University
Department of Computer Science
226 Atanasoff Hall
Ames, Iowa 50011-1040 USA
Tel: (515) 294-4377
Fax: (515) 294-0258
Email: ruby@cs.iastate.edu

January 22, 1997

Abstract

Specification languages could aid reuse to a larger extent if they could document all important facets of software, not just functional behavior. Since a specification language designer cannot know exactly what aspects of a piece of software will be important, users should be able to do "metaspecification"; that is, users should be able to declare new facets, and then use these facets to specify their software. Examples of facets that users might want to specify include time and space usage, safety considerations, aliasing, error checking, the user interface, etc.

Keywords: reuse, formal specification languages, metaspecification, facets, expressiveness

Workshop Goals: learning what can be done to aid reuse; networking; understanding and advancing the state-of-the-art in formal methods; finding new problems

Working Groups: Rigorous Behavioral Specification as an Aid to Reuse, Design Guidelines for Reuse, Reuse and OO Methods, Reuse and Formal Methods,

1 Background

Leavens's most prominent involvement in software reuse is as a program committee member for the *ICSR 4* conference in 1996. He has long been active in the theory of object-oriented (OO) methods, studying the concept of behavioral subtyping [Ame91, LW90, Lea91, LW94, LW95, DL96]. Subtyping is a kind of polymorphism [Car91] that allows objects of subtypes to be sent messages as if they were objects of their supertypes. Subtyping by itself, however, only guarantees the absence of type errors. Behavioral subtyping also guarantees the absence of surprising behavior. This allows client code, code that sends messages to objects, to be reused for behavioral subtypes.

In the past six years Leavens has been working in the semantics of specification languages, and has been designing the specification language Larch/C++ [Lea96c, Lea96b], which is a behavioral interface specification language tailored to C++.

Ruby is a graduate student doing research in formal specification of object-oriented programming languages. One of the directions of his research involves specifying enough information about C++ classes so that class libraries and frameworks can be reused and derived classes programmed without the need for the programmer to see the code for the superclasses.

2 Position

A *facet* of a program's behavior is something about the way a program executes that is important to clients of the program. One facet is functional behavior, but there are many others: time, space, security issues, aliasing, etc.

Because various facets are (by definition) important parts of the description of software, and because having an abstract description of software is important for reuse (for example, [LG86, Mey92]), our position is as follows.

> Specification languages should be able to document all facets of program behavior.

The facets that will be important to some users (e.g., communication bandwidth) are not necessarily important to others. Furthermore, since software is so flexible, it seems impossible to predict what facets will be important in advance. This leads us to the following conclusion.

> A specification language should allow users to declare new facets.

The technique of allowing a specification language to declare new facets is called *metaspecification*. Metaspecification could take two forms. One is to allow the user to declare new syntax and semantics for the specification of new facets. This would be similar to an extensible programming language, and would have a similar disadvantage: since each person could potentially use their own syntax, specifications would be difficult for others to read. The other form would be something akin to an abstract data type (ADT) declaration in a programming language: the user would declare a facet and some formal operations on it, but the syntax and framework of the specification language would remain unchanged. Exactly how this would work is a matter for research, but, following

Hehner [Heh93], one approach would be to incorporate such facets as if they were ghost variables[1] (like Hehner's `t` for time). There are several theoretical issues to investigate, such as the interaction of frame axioms among different facets. More importantly, the ease of declaring and using such facets, and their utility in promoting reuse needs to be investigated. Nevertheless, it is our belief that making specification languages better able to document existing software will aid reuse.

Our hypothesis is that the use of ghost variables, like Hehner's `t` for time, would be a sensible way to integrate user-specified facets into a formal specification language. The user would declare the name of the facet, and the vocabulary used to reason about it; the connection to reality would be documented, but would not have any formal bearing on the specification language. For example, to specify the space used by software, one could have a ghost variable `space`, along with an appropriate mathematical vocabulary (including, perhaps, absolute and order-of-magnitude comparisons). What a unit of space means (e.g., a byte), would be documented. How code would be verified to satisfy such a specification would also be something that the user might be able to describe in documentation of a facet. The exact form of this kind of metaspecification is a matter for future work.

3 Comparison with Related Work

Most formal specification languages only allow one to document functional behavior. That is, most formal specification languages do not help one specify how much time a procedure can take, or how much space it can use, or how it affects the firing of torpedos. Hehner's "practical theory of programming" [Heh93] is a notable exception. It uses partial correctness, and procedure specifications can mention a ghost variable, `t`, to describe the time a procedure may take. This allows one to recover total correctness if desired. It also hints at a way of adding facets to a specification language.

Temporal logic [MP92, Lam94] is used in specifying reactive or concurrent systems, where the relative timing of events matters. It allows one to specify liveness and safety properties, such as absence of deadlock and starvation. Ideally, with metaspecification, one would be able to use the ideas of temporal logic for the specification of such properties. However, even if one adopted temporal logic for the time facet, this would not solve the problem of how to specify other facets.

Sitaraman [Sit97] described design considerations for "implementation neutral" and "performance neutral" abstractions. These abstractions have correct implementations that allow many different implementations, including those with widely varying time and space usage. Because of the desire to permit such a wide variation, the abstractions specified are independent of the time and space facets. Our proposal would be more useful for abstractions that are designed for use in systems where time or space are critical resources, or for the documentation of more limited abstractions.

Because the list of facets that might be important to users is daunting (time, space, security, safety, aliasing, etc.), specification language designers should take a lesson from the history of programming language design. In programming, it was clear that programming languages needed to be able to adapt to various application domains. There are two ways in which this was done. The first was extensible languages, such as EL1 [Weg74], which proved difficult to read because

[1] A ghost variable is something that is thought of as a variable for the purposes of a specification, but which is not actually a variable in the code being specified.

everyone's programs had a different syntax. After that it was recognized that by allowing the datatypes of a programming language to be extended, one could allow users to raise the language level by implementing abstract data types that matched the problem domain more closely (as in CLU, Ada, and Smalltalk).

References

[Ame91] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, N.Y., 1991.

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, N.Y., 1991.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996.

[Heh93] Eric C.R. Hehner. *A Practical Theory of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lea91] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[Lea96a] Gary T. Leavens. Larch frequently asked questions. Version 1.62. Available in http://www.cs.iastate.edu/~leavens/larch-faq.html, December 1996.

[Lea96b] Gary T. Leavens. Larch/C++ Reference Manual. Version 4.20. Available in ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz or on the world wide web at the URL http://www.cs.iastate.edu/~leavens/larchc++.html, December 1996.

[Lea96c] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Hiam Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass., 1986.

[LW90] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, October 1990.

[LW94] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LW95] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, N.Y., 1992.

[Sit97] Murali Sitaraman. Impact of performance considerations on formal specification design. To appear in Formal Aspects of Computing, 1997.

[Weg74] Ben Wegbreit. The treatment of data types in EL1. *Communications of the ACM*, 17(5):251–264, May 1974.

4 Biography

Gary T. Leavens is an Associate Professor of Computer Science at Iowa State University, Ames Iowa. He has served on the program committees for OOPSLA (twice) and ICSR 4. His research focuses on formal methods on OO programming, and includes the theory of abstract data types, specification, verification, as well as topics in programming languages such as type theory and semantics. He has been involved in the design of Larch/Smalltalk and is the principal designer of Larch/C++. He is also the author of the Larch FAQ [Lea96a]. He received a Ph.D. in Computer Science from MIT in 1989.

Clyde Ruby is a graduate student in Computer Science at Iowa State University, Ames Iowa. He has more than 15 years experience as an analyst, designer, and implementer of software systems. His current research focuses on formal methods in object-oriented programming, specification, and verification. He is part of the Larch/C++ research group at Iowa State University.