Software Evolution and ‘Light’ Semantics

Extended Abstract

Dewayne E. Perry
Bell Laboratories
600 Mountain Ave
Murray Hill, NJ 07974 USA
+1.908.582.2529
dep@research.bell-labs.com

Keywords
Inscape Environment, Software Complexity, Software Evolution, Implications of Changes, Light Semantics, Interface Specifications, Software Composition

1 INTRODUCTION
The motivation for Inscape [4] came from my experience as a programmer, designer and architect. There were two major (and inter-related) problems that I encountered while building software systems where I had to use components built by other people: the pieces often did not fit when I put them together and changing code often produced surprising and unexpected results.

The first problem was due primarily to the informality and often incompleteness of component interfaces. The second problem was due ultimately to the complexity of the software and an inability to foresee or determine the consequences of changes. These problems result from three essential [1] and intertwined properties of building software systems: composition, evolution [2] [7] and complexity [5].

In coming to grips with the problem of composition, using formal interface specifications is the obvious choice. Enhancing the syntactic interfaces with semantic information is one useful way of expressing the intent of the interface provider and enabling the user to have all the information necessary to its correct and effective use.

How to attack the problem of evolution is not as obvious. The approach I took in the Inscape experiment was to use the specifications constructively in order to determine and maintain semantic dependencies [3]. The metaphor is that of a hardware chip: the dependencies are the pins that are used. Keeping track semantically as to how the interfaces are used is the analog of expressing the interface creator’s intent: it is capturing the users intent. Given that both interfaces and implementations evolve, keeping track of the dependencies enables the environment to help in understanding the effects of changes and where those effects take place.

The primary question then is how to make it work. An important clue to that question comes from considering the problem of complexity.

2 COMPLEXITY & ‘LIGHT’ SEMANTICS
In [5] I delineated two kinds of complexity. One kind of complexity is that of the intricacies of some algorithms and data structures. Here the complexity is analogous to a Bach four-voice fugue where there are very intricate and intertwining multi-dimensional constraints (in the case of a fugue, melodic and harmonic, or horizontal and vertical, constraints). One cannot make arbitrary changes without having (usually) disastrous consequences. All of the constraints must be understood, and in this case, that means gaining deep insight and understanding before successful changes can be made. Reasoning about this type of complexity is just plain hard. It is often hard for even the creator of the software to remember all the lines of reasoning that resulted in the final state.

However, there is another and completely different type of complexity that has different characteristics. This is the complexity that arises from the sheer wealth, or mass, of details. Here comprehension is hindered by the problem of scale, not the inherent complexity of the individual details. The analogy here is more to a Strauss tone poem or a Mahler symphony where the complexity is due to the mass of detail rather than a fugue’s intricate complexity (though there may be that kind of complexity underlying or buried in the mass of details).

My claim is that it is this latter kind of complexity that dominates most of the systems we build. There are parts of the systems that have the intricate complexity, but the rest of the system has the wealth of detail complexity. In support of this thesis, I offer the evidence of an error study of a release in the evolution of a large, realtime software system [8]. Overwhelmingly the faults
are the normal development faults with most of them fairly shallow and easy to find and solve. The hard ones were predictable: race conditions and performance.

This wealth of small details as the dominant form of complexity suggests that one needs to consider managing the small details rather than automating deep insight. For these small details, a much lighter form of semantic approach should be possible than full automated theorem proving but which goes beyond the current available forms of type checking.

There are various forms this idea of light semantics might take. One form might be by using only part of the semantic information or use it in rather simple but automated ways. Another form might be approximations to aid the developer (often interactively and iteratively) in developing an understanding of the code involved.

In Inscape, I chose the former. The underlying thesis of the semantic interconnections and the semantic propagation [6] is that it is made up of a large number of very small theorems rather than a very small number of large theorems. Supporting this is the fact that there is no notion of an invariant in the underlying logic of Inscape — just the small details of what is needed before and what is guaranteed after a operation has been executed.

Thus it is this idea of light semantics that was used in Inscape as the basis for the constructive use of interface specifications and semantic dependency based evolution.

3 COMPOSITION & EVOLUTION

One of the primary contributions of Inscape to interface research was the introduction of obligations and multiple results. Obligations are an often unmentioned side effect of an operation. They may arise for a variety of reasons, but typically are not made explicit. Multiple results are needed for any kind of software where exceptions are needed (which is most of it). Adding these two elements to interfaces then provided a solid base for describing interfaces for use in production (rather than academic or toy) software.

There is only one basic rule in constructing programs: all preconditions and obligations must be satisfied within an implementation or propagated to the interface. There are, however, constraints on what may be propagated to the interface. These constraints result in what I called precondition ceilings and obligation floors. Those preconditions and obligations which could not be propagated to the interface represented semantic problems that needed to be resolved.

By keeping track in the construction of an implementation where (and how) the preconditions and obligations were satisfied, changes to either the interfaces used or to the implementations could enable the environment to determine how those changes affected an encompassing interface and thus enable the programmer to understand the effects of those changes.

4 RELATED WORK

There have been a number of projects that I know of that have been based on Inscape. Among these was an internally sponsored research project at Anderson Consulting called Software Interface Specification and Analysis. This project evolved into a Component-Based Software Engineering (CBSE) project, co-sponsored by the U.S. government Department of Commerce and Anderson Consulting. This work in turn was incorporated in their internal component framework and deployed in many of their client companies. This project in turn influenced David Curtis, now one of the authors of the Corba Component Model. See the papers in Appendix B by Jim Ning and/or Wojtek Koczaczynski.

Daniel Jackson's PhD thesis and some of his subsequent work was influenced by my 'light' semantic approach in Inscape. Aspect uses partial specifications provided in an annotation language which can be efficiently checked. The Aspect checker then uses a dependency analysis to check the code against the annotations. His more recent work also fruitfully explores the space of light semantics to provide various kinds of analyses and models.

Don Batory's work takes an approach very similar to my propagation logic in generating systems from components. These components are annotated with various properties which are then used compositionally to stitch the components together in a consistently well-formed composition.

Appendix B has further references for work either inspired by Inscape or related work exploring this exciting space of light semantics.

5 SUMMARY

The space of light semantics is an exceedingly fruitful field for research. Inscape was one of the first projects to explore this space and it has been followed some very interesting work in the subsequent years. The space is by no means exhausted — indeed it has been not been explored much at all. It is a ripe area for research that has too long been overlooked. And for me, the spirit of Inscape continues in my software architecture work.

The viewgraphs for this talk as well as all the relevant Inscape papers (See Appendix A below) can be found at my web site: www.bell-labs.com/user/dep/

ACKNOWLEDGEMENTS

First, I thank Jim Christ, Bill Schell, Steve Popovich, Peggy Quinn and Helen Diamontitus for their various contributions to Inscape. And of course, special thanks go to Prof. Gail Kaiser for our long and fruitful collaboration on Infuse, the change management part of
Next, I thank Don Batory, Prem Devanbu, Daniel Jackson, Wojtek Kozaczynski, Jim Q. Ning, and David Notkin for their various clarifying discussions and contributions to this talk and abstract.

Finally, I thank the ICSE99 Program Committee for selecting the Inscape paper as the most influential paper from ICSE11.

REFERENCES


APPENDIX A: Inscape Bibliography


Proceedings of the 21st International Conference on Software Engineering (ICSE'99)


APPENDIX B: Some Related Work

This should by no means be considered an exhaustive bibliography. Instead it represents some of the work that is either directly or indirectly influenced by Inscape or, like Inscape, explores this space of ‘light’ semantics.


Don Batory and Sean O’Malley. “The Design and Imple-