Tools and Behavioral Abstraction: A Direction for Software Engineering

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Capsule Summary. As in other engineering professions, software engineers rely on tools. Such tools can analyze program texts and design specifications more automatically and in more detail than ever before. While many tools today are applied to find new defects in old code, I predict that more software-engineering tools of the future will be available to software authors at the time of authoring. If such analysis tools can be made to be fast enough and easy enough to use, they can help software engineers better produce and evolve programs.
A programming language shapes how software engineers approach problems. Yet the abstraction level of many popular languages today is not much higher than that of C programs several decades ago. Moreover, the abstraction level is the same throughout the program text, leaving no room for *behavioral abstraction* where the design of a program is divided up into stages that gradually introduce more details. A stronger arsenal of analysis tools can enable languages and development environments to give good support for behavioral abstraction.

0 Introduction

The science and practice of software engineering have made great strides in the few decades since its inception. For example, we have seen the rise of structured programming, we have come to appreciate types (enforced either statically or dynamically), we have developed models for program semantics that underlie reasoning about programs, we have recognized the role of (unit, system, regression, white-box, . . .) testing, we are starting to understand how to collect metrics that help in setting software-development schedules, and, having available far more CPU cycles than would have been easily imaginable in 1968, we have produced tools that assist in all these tasks.

Nevertheless, software engineering remains difficult and expensive.

What is it all about? Software engineering produces software—if we had no interest in software, the activities that make up software engineering would have no purpose. We have a number of desiderata for this engineering task. We want to develop software. . .

– **with the right features.** A software project is successful only if the final software does what its users need it to do. The requirements for a piece of software can be difficult to determine, the assumptions made by users and the software team may be in conflict with each other, and that which seems important when prototyping may end up being different from what is important during deployment of the final software.

- **that is easy to use.** At times, we have all been frustrated at desktop software where we found menus to be unnecessarily cumbersome or non-standard, and at web software where the number of clicks required for a purchase seems ridiculously high. Software in a car must allow operation while the driver stays on task.
- **that is hard to misuse,** both accidentally and maliciously. Letting an operator open up the landing gear of an airplane too far off the ground may not be what we want of our embedded software. And we expect banking software to apply judicious security measures to prevent various forms of privacy breaches.
- **can be developed effectively,** on schedule and free of defects. Ideally, we want all software to have zero defects (like crashes, deadlocks, or incorrect functional behavior). But reducing defects comes at a price, so market forces and engineering concerns may bring about compromises. Whatever defect rates can be tolerated, we would like the software engineering process to get there as easily and confidently as possible.
- **can be evolved,** to add or remove features, to adapt to new environments, to fix defects, and to preserve and transfer knowledge between developers. Successful software lives far beyond its first version, an evolution in which feature sets and development teams change. As the software is changed, one needs to understand what modifications are necessary and also which modifications are possible.

The future of software engineering holds improvements in these and related areas. In this essay, I will focus on issues concerning the software artifacts themselves, paying much less attention to the important issues of determining requirements and of usability of the software itself.

1 Composing Programs

The phrase "composing programs" has two senses. One sense is that of authoring programs, like a musician composes music. Another sense is that of combining program elements to form a whole program. When we create a piece of software, we take part in activities that comprise both senses of "composing programs": we both write new code, designing new data structures and algorithms, and combine existing or planned program elements, calling libraries and reusing types of data structures.[0] Both activities wrestle with complexity. Complexity arises in the algorithms used, in the data structures used, and in the interaction between features. The major tool for keeping complexity at bay is *abstraction*.

Common Forms of Abstraction. When authoring new program elements, abstraction is used to group related functionality to make larger building blocks. Standard programming languages offer several facilities for abstraction. *Procedural abstraction* provides the ability to extend the primitive operations of the language with user-defined, compound operations. These can be named and parameterized, and they separate uses of the

[0] Encompassing both senses, the IFIP Working Group 2.3 on *Programming Methodology* states as its aim, "To increase programmers' ability to compose programs".

operation from its implementation. *Data abstraction* provides the ability to extend the primitive data types of the language with user-defined, compound data types. These can be named and sometimes parameterized, and they can separate uses of the data from the actual representation.

For example, one part of a program can create and apply operations to a file stream object without having to understand how many underlying buffers, file handles, and position markers are part of the actual data representation or to understand the operations of the protocol used by the disk to fetch the contents of a particular sector.

When combining program elements, abstraction is needed to let a program element declare indifference about certain aspects of its uses. The main facility for this is *parameterization*. This is as important for the provider of a program element as it is for clients of the program element. Sometimes, there are restrictions on how a program element can be parameterized. Such restriction can be declared, for example, as type constraints or precondition contracts.

For example, a sorting procedure can be parameterized to work for any data type, as long as the objects of that data type have some means of being compared. This can be declared as a type constraint, requiring the sorting procedure's type parameter to be a type that implements a *Compare* method. Alternatively, the sorting procedure can be parameterized with a comparison procedure, and a precondition of the sorting procedure can require the comparison to be transitive.

Abstraction by Occlusion. These and similar abstraction facilities go a long way to organizing a program into understandable and manageable pieces. But not far enough. There is one major aspect of abstraction that is conspicuously underdeveloped in popular languages. The facilities for abstraction I mentioned above provide a way to occlude parts of a program, revealing the insides of those parts only in certain scopes. For example, the callers of a procedure see only the procedure signature, not its implementation, and the fields of a class may be visible only to the implementation of the class itself, not to clients of the class. We may think of this as *abstraction by occlusion*. What is lacking in this kind of abstraction is that it only gives us two options, either we have to try to understand a particular part of the program by digging into its details or we don't get any information to help us understand it. Stated differently, a view of a program is a subset of the code that is ultimately executed.

For example, consider a type that represents a graph of vertices and edges, and suppose you want to evolve the functionality of this type, perhaps to fix a defect or add a feature. When you look at the implementation, you immediately get immersed with the details of how edges are represented (perhaps there is a linked list of vertex pairs, perhaps each vertex stores an array of its edge-ordering successors, or perhaps both of these representations are used, redundantly) and how results of recent queries on the graph may be cached. This may be more details than you need to see. Furthermore, suppose you change the explicit stack used by an iterative depth-first graph traversal from using a vertex sequence to reusing reversed pointers in the graph à la Schorr-Waite. You will then remove the previous vertex sequence, which means that the next person to look at this code will have to understand the complicated pointer swiveling rather than the easier-to-grasp stack of vertices. Finally, the fact that you would remove the previous vertex-sequence implementation is unfortunate, because it would have been nice to have kept it around as a reference implementation.

Behavioral Abstraction. Abstraction by occlusion does not provide a good story for understanding more abstractly what the code is supposed to do, or even giving a more abstract view of how the code does what it does. If you were to explain a piece of software to a colleague, you are not likely to explain it line by line. Rather, you would first explain roughly what it does and then go into more and more details. Let me call this *behavioral abstraction*.

While behavioral abstraction is mostly missing from popular languages, it does exist in one important form: *contracts*, and especially procedure postcondition contracts, which are found in some languages and as mark-ups of other languages [17, 16, 6, 7, 5]. A postcondition contract abstracts over behavior by giving an expression that relates the possible pre- and post-states of the procedure. To give a standard example, the postcondition of a sorting procedure may say that its output is a permutation of the input that arranges the elements in ascending order. Such a postcondition is easier for a human to understand than trying to peer into the implementation to figure out what its loops or recursive calls do. It also opens the possibility for verification tools to compare the implementation with the intended behavior of the procedure as specified by the postcondition.

A weaker form of behavioral abstraction is found in software engineering outside programming languages, namely in test suites. To the extent that a test suite can be seen as a set of use cases, it does provide a form of behavioral abstraction that, like contracts, shows some of the intent of the program design.

The behavioral abstraction provided by contracts and use-case test suites provide one layer of description above the actual code. But one can imagine applying several layers of behavioral abstraction.

Stepwise Refinement. The development of many programs today goes in one fell swoop from sketchy ideas of what the program is supposed to do to low-level code that implements the ideas. This is not always wrong. Some programmers have tremendous insights into how, for example, low-level network protocols for distributed applications ought to behave. In such cases, one may need tools for verification or property discovery to ensure that the program has the desired properties. But, upon reflection, this process of writing code and *then* coming up with properties that the code should have and trying to ascertain that the code does indeed have those properties seems terribly twisted. Why wouldn't we want to *start* by writing down the salient properties of the software to be constructed, and then in stages add more detail, each maintaining the properties described in previous stages, until the program is complete? This process is called *stepwise refinement*, an old idea pioneered by Dijkstra [8] and Wirth [26].

Well, really, why not? There are several barriers to applying this technique in practice. Traditionally, stepwise refinement starts with a high-level specification. Writing high-level specifications is hard, so how do we know they are correct? While programming is also hard, a program has the advantage that it can be executed. This allows us to subject it to testing, simulation, or other use cases, which can be more immediately satisfying and can, in some development organizations, provide management with a facade of progress milestones.

If we are to use layers of behavioral abstraction, then we must ensure that any stage of the program, not just the final program, can be the target of modeling, simulation, verification, and application of use cases.

Another problem with stepwise refinement—or, a misconception, really—is that the high-level specification encompasses all desired properties of the final program. In reality, the specification is introduced gradually, just like any other part of the design. For example, the introduction of an efficient data structure or a cache may take place at a stage in the middle of the refinement process. By splitting the specification in this way, each piece of it may be easier to get right.

Finally, a problem to overcome with stepwise refinement is that so much of the programming culture today is centered around final code. It will take some training to start writing higher-level properties instead, and engineers will need to learn when it is worthwhile to introduce another stage versus when it is better to combine some design decisions into one stage. There is good reason to believe that software engineers will develop that learning, because engineers have learned similar trade-offs in today's practices; for example, the trade-off of when it is a good idea to introduce a new procedure to perform some series of computational steps versus when it is better to just do the steps wherever they are needed.

Let's consider a vision for what a development system that embraces behavioral abstraction may look like.

2 A Development Environment for Behavioral Abstraction

First and foremost, a development environment is a tool set that allows engineers to *express and understand designs*. The tools and languages we use guide our thinking when developing a program and are key to human comprehension when reading code. By supporting behavioral abstraction, the environment should allow engineers to describe the system at different levels of abstraction. Here are some of the characteristics of such a development environment.

Descriptions at Multiple Stages. The environment permits and keeps track of increasingly detailed descriptions of the system, each a different *stage*. A stage serves as a behavioral abstraction of subsequent stages.

Ceaseless Analysis. At each stage, the environment analyzes the given program. For example, it may generate tests, run a test suite, simulate the description, verify the description to have certain well-formedness properties (*e.g.*, that it parses correctly, type checks, and adheres to the usage rules (preconditions) of the operations it uses), infer properties about the description, and verify that the stage maintains the properties described in previous stages.

To the extent possible, trouble spots found by such analyses only give rise to warnings, so as not to prevent further analyses from running.

There is no reason for the analyses to be idle, ever. Instead, the environment ceaselessly runs the analyses in the background, noting results as they become available.

Multiple Forms of Descriptions. The descriptions at one stage can take many different forms. For example, they may spell out a use case, they may prescribe pre- and postcondition contracts, they may give an abstract program segment, they may introduce variables and associated invariants, or they may supply what we today think of as an ordinary piece of program text written in a programming language. Many of these descriptions (like test cases and weak postconditions) are incomplete, homing in on what is important in the stage and leaving properties irrelevant to the stage unspecified.

In some cases, it may be natural to switch languages when going into a new stage. For example, a stage using JavaScript code may facilitate the orchestration of actions described in more detailed stages as C# code for Silverlight [18]. Other examples of orchestration languages include Orc [20] and Axum [19].

Note, since one form of description is code in a programming language, one could develop software using this system in the same way that software is developed today. This is a feature, because it would allow engineers to try out the capabilities of the new system gradually. One gets a similar effect with dynamically checked contracts: if a notation for them is available, programmers can start taking advantage of them gradually and will get instant benefits (namely, run-time checks) for any contract added. In contrast, if a machine readable format for contracts is not available, contracts that appear in the head of an engineer while writing or studying the code turn into lost opportunities for tools to help along.

Change of Representation. It is important that a stage can refine the behavior of a previous stage not just algorithmically—to make an incomplete description more precise—but also by changing the data representation. For example, the description at one stage may conveniently use mathematical types like sets and sequences, whereas a subsequent stage may decide to represent these by a list or a tree, and an even later stage may want to add some redundant representations, like caches, to make the program more efficient.

By describing an optimization as a separate stage, the overall design can still be understood at previous stages without the code clutter that many good optimizations often bring about.

This kind of transformation is known as *data refinement* and has been studied extensively (*e.g.*, [12, 3, 21, 22, 9]).

Executable Code. The development environment allows designs to be described down to the level of the executable code of the final product. That is, the development environment is not just for modeling. However, the code need not all be produced by traditional compilers; parts of it may be synthesized from higher-level descriptions. Similar ideas have been explored in the language SETL [23] and in correct-by-construction synthesis frameworks [24].

Automation. For the most part, we think of program analysis as an automatic process. Sometimes, however, an analysis may need manually supplied hints to improve the analysis or to keep it from getting stuck. For example, a static type checker may need a dynamic type cast, a type inference engine may need type annotations for recursive procedures, and a proof assistant may need a lemma or a proof-tactic command. To provide a seamless integration of such analysis tools, the hints are best supplied using concepts at the level of the description at hand. For example, an assert statement in a program text can serve as a lemma and may, in that context, be more readily understood by an engineer than an indication that a certain proof-tactic command needs to be invoked at some point in the proof.

Room for Informality. The development environment allows formal verification of properties in and between stages. Indeed, such verification may be easier than in a monolithic program with specifications, because behavioral abstraction naturally breaks up the verification tasks in smaller pieces. However, it is important not to prevent programs from being written, simulated, or executed just because they cannot be formally verified at the time. Instead, an engineer has the option to accept responsibility for the correctness of certain conditions.

3 Challenges

A number of challenges lie ahead before a development environment built around behavioral abstraction can become a reality. Here are some of them.

User Interface. A behavioral-abstraction environment could benefit from a different user interface than the file-based integrated development environments of today. Conceptually, a stage contains only the differences in the program since the previous stage. Sometimes, one may want to see only these differences; at other times, one may want to see the combined effect of a number of stages.

Early Simulation. A stage is more useful if something can be done with it. The closer something is to executable code, the easier it is to imagine how it may be simulated. How should one simulate the program in its most abstract and most incomplete stages?

Tools in the spirit of Alloy [13] or Formula [14] would be useful here.

Prioritizing Analyses. To run analyses ceaselessly is harder than one might first imagine. Many analyses are designed with various compromises in mind, terminate quickly, and then have nothing more to do. For example, a type checker may be designed to report an error when types don't match up; but instead of being idle after warning about such a condition, it could continue being useful by searching for program snippets that supply possible type conversions. If an analysis could use boundless amounts of CPU time, the development environment must use some measures of priority. For example, if the engineer just modified the implementation of a procedure, it would seem a good idea to give priority to analyses that can tell the engineer something useful about the new implementation. This may require recognition of differences in the abstract syntax tree (*e.g.*, to re-analyze only those code paths that were changed), keeping track of dependencies (*e.g.*, the call graph), and maybe even using a program verifier's proof to determine detailed dependencies (*e.g.*, to figure out that one procedure relies only on some specific portions of other specifications).

Allowing Informality. Ideally, we want a program to be all correct in the end. But before then, engineers may be able to be more productive if they can continue simulating or testing their program even in the presence of some omissions or errors. How to go about doing this is a challenge. Perhaps we can draw inspiration from dynamically typed languages that try hard to give some interpretation to code at run time, even if a static type checker would have had a hard time with the program. Also, what run-time checks could be used to make up for failed or incomplete proofs?

Refinements into Dynamically Allocated State. Many modern applications require, or at least benefit from, use of dynamically allocated state. Yet, this issue has not received nearly the same amount of attention in data refinement as it has in the area of program verification [11].

Supporting Program Evolution. Getting a new program together is a big feat. But even more time will be spent on the program as is evolves into subsequent versions. To determine if a development system really helps in program evolution, one needs to undertake some long-running deployments.

4 Related Work and Acknowledgments

Many people have had ideas in this space and much work has been done on various components. I cannot do them all justice here, but in addition to what I have already mentioned above, I will mention a number of efforts and people who have been most influential in my own thinking.

Built on the Event-B formalism [1], the Rodin system [2] is perhaps the closest of existing systems to what I have sketched. With its pluses and minuses, it sports an integrated development environment, uses a sequence of files to record stages of the design, mostly unifies descriptions into one form ("events"), lacks direct support for dynamic memory allocation, does not routinely output executable code, and requires a noticeable amount of prover interaction.

Other important systems that let designs be described at various levels of abstraction are B [0] and VDM [15]. The Play-in, Play-out technique [10] simulates systems from given use cases.

Bertrand Meyer pioneered the inclusion of behavioral descriptions (contracts) of program elements as compiled constructs in a modern programming language [17].

When using verification tools today, one of the focus activities is finding specifications that describe what the code does. It would seem a better use of our time to shift that focus to finding code that satisfy specifications. In some cases, programs can even be automatically synthesized from specifications, as has been investigated, for example, by Doug Smith and colleagues [25, 24].

This essay has also benefited from ideas by and discussions with Jean-Raymond Abrial, Mike Barnett, Michael Butler, Mike Ernst, Leslie Lamport, Michał Moskal, and Wolfram Schulte, as well as Mark Utting, who formulated some ideas around a collection of descriptions on which tools operate to produce efficient code and report potential problems. Kuat Yessenov worked as my research intern on language and verification support for refinement.

I have been personally influenced by Ralph Back, whose ground-breaking and inspiring work gave stepwise refinement mathematical rigor [3, 4]. Work by Shaz Qadeer and Serdar Tasiran hammered home to me how much simpler invariants can be if they are written for an abstract program instead of for the final code. A number of people have reminded me that some descriptions are most easily formulated as code; among them, Wolfram Schulte, Ras Bodik, and Gerwin Klein. Cliff Jones has repeatedly pointed out the importance of requirements and the high cost incurred by getting them wrong.

Bart Jacobs and Rosemary Monahan provided comments on a draft of this essay.

5 Conclusion

Software engineering uses lots of tools. To take a bigger step in improving software engineering, we need not just to improve each tool, but to combine tools into usable development environments. The time is ripe for that now. As a vision for how tools and languages can be combined in the future of software engineering, I discussed in this essay a development environment built around behavioral abstraction, where programs are divided up not just into modules, types, and procedures, but also according to the level of abstraction at which they describe the program under development.

References

0. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, April 2010.
3. R.-J. R. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978. Report A-1978-4.
4. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
5. Mike Barnett, Manuel Fähndrich, and Francesco Logozzo. Embedded contract languages. In *ACM SAC - OOPS*. ACM, March 2010.

6. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
7. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. http://frama-c.com/.
8. E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968.
9. David Gries and Dennis Volpano. The transform — a new language construct. *Structured Programming*, 11(1):1–10, 1990.
10. David Harel, Hillel Kugler, Rami Marelly, and Amir Pnueli. Smart Play-out of behavioral requirements. In Mark Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002*, volume 2517 of *Lecture Notes in Computer Science*, pages 378–398. Springer, November 2002.
11. John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-011, University of Central Florida, School of EECS, 2009.
12. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
13. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
14. Ethan K. Jackson, Dirk Seifert, Markus Dahlweid, Thomas Santen, Nikolaj Bjørner, and Wolfram Schulte. Specifying and composing non-functional requirements in model-based development. In Alexandre Bergel and Johan Fabry, editors, *Proceedings of the 8th International Conference on Software Composition*, volume 5634 of *Lecture Notes in Computer Science*, pages 72–89. Springer, July 2009.
15. Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.
16. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, March 2006.
17. Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.
18. Microsoft. Silverlight. http://www.microsoft.com/silverlight/.
19. Microsoft. Axum. http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx, 2010.
20. Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling*, 6(1):83–110, March 2007.
21. Carroll Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, 1990.
22. Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
23. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Texts and Monographs in Computer Science. Springer, 1986.
24. Douglas R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.
25. Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Transactions on Software Engineering*, 11(11):1278–1295, November 1985.
26. N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14:221–227, 1971.