Roadmap for Enhanced Languages and Methods to Aid Verification

Gary T. Leavens
Iowa State University, Ames, IA USA
leavens@cs.iastate.edu

Jean-Raymond Abrial
ETH Zürich, Switzerland
jabrial@inf.ethz.ch

Don Batory
University of Texas, Austin, TX USA
batory@cs.utexas.edu

Michael Butler
University of Southampton, UK
M.J.Butler@ecs.soton.ac.uk

Alessandro Coglio
Kestrel Institute, CA, USA
coglio@kestrel.edu

Kathi Fisler
Worcester Polytechnic Institute, MA, USA
kfisler@gmail.com

Eric Hehner
University of Toronto, Canada
hehner@cs.utoronto.ca

Cliff Jones
Newcastle, UK
cliff.jones@ncl.ac.uk

Dale Miller
INRIA-Futurs, Polytechnique, France
dale@lix.polytechnique.fr

Simon Peyton-Jones
Microsoft Research, Cambridge, UK
simonpj@microsoft.com

Murali Sitaraman
Clemson University, SC, USA
murali@cs.clemson.edu

Douglas R. Smith
Kestrel Institute, CA, USA
smith@kestrel.edu

Aaron Stump
Washington University of St. Louis, MO, USA
stump@cse.wustl.edu

Abstract

This roadmap describes ways that researchers in four areas — specification languages, program generation, correctness by construction, and programming languages — might help further the goal of verified software. It also describes what advances the "verified software" grand challenge might anticipate or demand from work in these areas. That is, the roadmap is intended to help foster collaboration between the grand challenge and these research areas.

A common goal for research in these areas is to establish language designs and tool architectures that would allow multiple annotations and tools to be used on a single program. In the long term, researchers could try to unify these annotations and integrate such tools.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications — languages, methodologies; D.2.4 [*Software Engineering*]: Software/Program Verification — assertion checkers, class invariants, correctness proofs, formal methods, model checking, programming by contract; D.2.10 [*Software Engineering*]: Design — methodologies, representation; D.2.11 [*Software Engineering*]: Software Architecture — data abstraction, information hiding, languages; D.2.12 [*Software Engineering*]: Reusable Software — reusable libraries; D.3.1 [*Programming Languages*]: Definitions and Theory — semantics; D.3.2 [*Programming Languages*]: Language Classifications — extensible languages, specialized application languages, very high-level languages; D.3.3 [*Programming Languages*]: Language Constructs and Features — abstract data types, classes and objects; D.3.4 [*Programming Languages*]: Processors — translator writing systems and compiler generators; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs — assertions, invariants, logics of programs, mechanical verification, pre- and post-conditions, specification techniques; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages — program analysis; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs— type structure

General Terms Languages, Verification

Keywords Verification, verified software grand challenge, specification languages, program generation, correctness by construction, programming languages, tools, annotations.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.

1. Introduction

Hoare has proposed a grand challenge project, formerly called the "verifying compiler" grand challenge [69], and now called the "verified software" grand challenge by Hoare, Misra, and Shankar [70]. The original idea was to automatically check correctness of programs that are "specified by types, assertions, and other redundant annotations." However, the current version of the grand challenge recognizes the possibility of many tools, some of which may require human intervention or assistance. In any case, verification would be based on the text of the program and the annotations contained within it.

1.1 Audience

This report is addressed to two audiences. The first is researchers interested in program verification, especially related to the "verified software" grand challenge. The second is researchers in the following areas:

specification languages that describe behavior or properties to be verified,

program generation that automatically synthesizes code,

correctness by construction that concerns development and documentation of implementations especially to facilitate verification, and

programming languages that describe algorithms and data.

The report is addressed to researchers in these four areas who are interested in verification, specifically how their work might aid the verifying software grand challenge. This report explains what these four areas might do to help the overall grand challenge project and thus foster the goal of verified software within the scope of the grand challenge project. It is not intended to suggest an overall research agenda for any of these areas.

1.2 Motivation

There are many approaches to verification, all of which are embraced by the grand challenge effort. One can write or find code and verify it using a variety of tools and approaches.

While recognizing the value of many approaches to producing verified software, researchers in the four areas mentioned above are often motivated by the idea of gaining benefits (in ease, productivity, or power of verification) by providing the verifier with more information than just a bare program in some standard programming language. Verifying a bare program after-the-fact has the following fundamental problems.

- Without a specification or some annotations in the code, the properties that one can verify must be implicit and thus very weak, such as that the program will not crash or throw exceptions.
- Even with a specification, a program can be arbitrarily difficult to verify (due to lack of modularity or other artificial complexities).

With regard to the first point, even adding some partial specifications makes the verification problem more interesting and the results more useful. This is a potentially valuable technique for legacy code. For example, one might specify that a function returns a list of length equal to its input, which is only a partial specification of what the function does. Indeed, there is an entire spectrum of properties that one might consider, as shown in Figure 1. So there is not necessarily a unique best specification for a function, since some kinds of properties, such as resource consumption, its behavior in a transactional setting, its real-time behavior, and so on, may best be thought of as outside of the traditional specification of functional behavior.

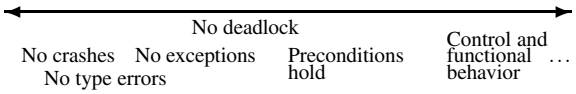

Figure 1. A spectrum of specification properties, from partial specifications on the left to more complete specifications on the right.

With regard to the second point, researchers believe that information about design decisions made in the program's development can be of great use to the verification process. Well-known examples are annotations for loops and object invariants, but information can also be obtained from the process of generating a program (up to and including a complete proof), and the process of constructing a program and its proof hand in hand. Intermediate modeling and refinement steps are also believed to greatly aid verification and may in the limit constitute a proof. Types in programming languages can also be augmented with additional information related to correctness proofs, and other program annotations, such as those describing ownership in the heap, can be of great value. To summarize, the motivation for all these areas is to make such information available to a verifier.

1.3 Limitations

The "research roadmap" that follows is limited in several ways.

First, the roadmap focuses on the four research areas named above and their relation to verification. Other techniques and research areas related to verified software are largely ignored. Furthermore, although there are many ways in which these four research areas might aid the general goal of more reliable software, this roadmap only focuses on the specific ways that these areas might produce verified or more easily verifiable software in the context of the grand challenge project. Much research is already going on in all of these areas to promote more reliable software, and such research would also contribute, indirectly, to the goal of making software easier to verify. However, discussing all such research would lead to a very broad survey which would be of less use to the verified software grand challenge.

The second way in which our roadmap is limited is that it has only (thus far) drawn on the expertise of a very small sample of researchers in each of the research areas.[1] The authors of this report were selected in the following way. A conference on the verified software grand challenge was held in Zürich Switzerland in October 2005 [68]. At that conference, the organizers — Hoare, Shankar, and Misra — picked leaders for three committees to write research roadmaps. Leavens was picked to lead the committee writing this report. Leavens in turn picked the committee members, intentionally aiming for a small committee, using a selection that was biased toward people who had attended the conference in Zürich.

Finally, the preceding limitations result in limitations on the applicability of our roadmap. First it is biased toward research directly related to the verified software grand challenge. Second, since the committee is small compared to the number of researchers in the four research areas, this report does not necessarily represent a consensus of the researchers in any of the four research areas.

[1] However, it also reflects feedback from the members of IFIP working group 2.3, the mini-conference on verified software April 1–2, 2006 held at SRI, and the Dagstuhl workshop on "The Challenge of Software Verification" July 10–13, 2006.

1.4 Outline

The next section gives some background about verification problems and challenge problems. Following that, Section 3 describes the common goal of the four areas with respect to the grand challenge, that is, what they might, overall, provide to it. Sections 4–7 describe the more specific needs and potential research directions in each of the four areas. Section 8 concludes this report.

2. Background

This section gives some background on verification problems and lays out some needs that researchers in the four areas have for challenge problems.

2.1 Verification Problems

An enhanced language or tool is intended to work on some class of verification problems. A precise way to state a such class of *verification problems* is to describe:

- a specification language, in which to state the assumptions and guarantees of a correct implementation, and
- a programming language, in which to implement the specifications, and whose code is to be verified.

A specification in the specification language together with a program in the programming language constitute a problem for a verification system. A pair of a specification and programming language describe the set of possible such problem instances that such a system should be able to handle.

The specification language and programming language might be integrated; there is no need to have two separate languages. Some examples of integrated languages are Gypsy [7], Alphard [64, 93, 124], Euclid [84, 92], Eiffel [98, 99], Resolve [45, 116], and SPARK [18].

For various reasons the grand challenge project has not articulated, and will probably not articulate, constraints on what verification problems are of interest. But verification problems of interest will be described indirectly, through challenge problems.

2.2 Challenge problems

Challenge problems can help stimulate research, especially in the short term. The following are some suggestions for such challenge problems.

To reward research that can handle problems of significant size, the challenge problems should be big enough to require reusable modules and structuring (at multiple levels).

Challenge problems at a minimum need to have explicitly stated (informal) requirements. It will also be helpful to have formal requirement models.

A formal specification of the properties of interest for each challenge problem is also needed by each of the four areas. Those working in specification languages could use the formal specification as a baseline for case studies that compare their work against the notation used to state the properties of the challenge problem. The other areas need a formal specification as a starting point for certain kinds of research.

As a practical matter, and as an aid to those working in all four areas, challenge problems should also come with test cases.

To aid work on programming languages and some researchers in the correctness by construction approach, it would also be helpful to provide well-tested candidate implementations with each challenge problem. Such implementations would be useful to researchers in programming languages, who could try to devise alternative implementations or languages that would allow easier verification of implementations.

3. Common Goal: Verifiable Artifacts

To set out goals for the four areas, we make some assumptions. The main assumption is that the grand challenge is interested in *at least* the following:

- specification of safety properties (e.g., the relation between inputs and outputs, lack of deadlock), and
- imperative programming languages (such as Pascal or C), including object-oriented languages (such as Java).

On the one hand, although it is non-trivial and of some economic importance, this is a rather small class of verification problems. For example, most imperative programming languages have only limited support for concurrency (e.g., threads in Java), but different models of concurrency may become increasingly important in the next several years. On the other hand it is still perhaps too large, because it encompasses the entire spectrum of safety properties, including everything in Figure 1. The reader should keep in mind that the grand challenge project may indeed be interested in other kinds of specifications and programs. In that case this report will most likely be missing some potentially interesting research directions.

Assuming the goal of the project is to build tools that will be able to handle at least verifying safety properties for imperative languages, we see the following short-term and long-term goals that are shared across the four areas.

3.1 Short Term: Extensible Languages and Tools

In the short term (i.e., in the next 5-7 years), a common goal is to allow for extension of tools and languages by other researchers (and ultimately, by users).

For specification and programming languages, this means designing languages so that other researchers (and ultimately users) can add new specification notations and new annotations to aid in verification proofs. These languages should allow specifications to be added (and proved) incrementally.[2]

Such extensions should ideally not just describe syntax, but also have access to information from the language processor (e.g., a compiler). User-extensible annotation mechanisms, such as those found in C# and Java may be a useful technique for achieving parts of this goal.

In all four areas, tool builders should strive to define architectures that will permit other researchers to easily add new specifications and other proof-oriented annotations, that will enable other tools to cooperate on verification of the same program. XML may be an aid for achieving parts of this goal. Overall, the idea is to recognize that no one tool will have all the necessary features for attacking all parts of a difficult verification problem. Tool (framework) builders should make it easier to build new tools or extend existing tools. This in turn will help other researchers gain much needed experience with their approaches, but at a lower cost.

Since efforts in building extensible tools can have a multiplicative effect in enabling research, such efforts should be highly encouraged by the project.

3.2 Long Term: Unification

In the long term (8-15 years), researchers should attempt some consolidation of various languages and tools in their areas. This is desirable because the software industry does not want to deal with many different languages, notations, methods, and tools. Furthermore, it is also theoretically unsatisfying to have to explain a wide

[2] In addition to the utility of such annotations in verification, the more properties one proves, the more confidence one has in a program. This is an additional motivation for the goal of allowing language extension.

diversity of approaches. Thus, while research will continue to make progress by exploring a wide range of approaches to attacking verification problems, in the second half of the project some researchers should also build on and consolidate the ideas of several tools and languages.

4. Research in Specification Languages

This section was mainly written by: Gary T. Leavens, Kathi Fisler, Cliff Jones, Douglas R. Smith, and Murali Sitaraman.

4.1 Need for Specification Languages

Research in (formal) specification languages is central to the grand challenge, because interesting verification problems contain interesting specifications. Thus the grand challenge project needs at least one specification language for stating the properties that are to be verified in the class of verification problems of interest. Even if the class of verification problems only encompasses very weak or partial specifications, such as those on the left side of Figure 1, there will still be the need for a specification language (although in the extreme case, the specification language might be trivial in the sense that it contains just one sentence: "the program should not crash").

4.2 Assumed Scope

Since it is not clear what properties are of interest to the grand challenge, this section assumes that the set of properties of interest includes *at least* safety properties for sequential and concurrent programs. That is, the remainder of this section assumes that the grand challenge is interested in specifying at least:

- assertions about states and data values, which allow one to describe the functionality of procedures in imperative programming languages, and
- properties of the history of events in a program's execution.

4.3 Background: Kinds of Specification Languages

This section defines terms used in the description of short-term and long-term research directions, particularly about different kinds of specification languages.

Specifications can be stated at many different abstraction levels. At the highest level of abstraction are *requirements* [138], which describe the behavior of entire programs from the end-user's perspective, often including non-functional properties, such as cost or time. Requirements are initially informal, but may be (partially) formalized later. What we hereafter refer to as *specifications* are statements that may describe or refer to a program's internal states or events, which may not be directly visible to a program's user. Such statements are usually formal and describe a class of programs or program modules (components) that have a design with features that can be related to the internal states or events mentioned in the specification. Thus what we call specifications are at a level of abstraction that is more relevant to the detailed design of a program. Such detailed-design specifications are capable of documenting interfaces of individual program modules, such as procedures or classes [144].

One technique for writing such specifications is algebraic [57, 56, 46, 54], in which one writes axioms that relate operations to other operations. While the early papers described non-imperative examples, this technique has also been adapted to specification of imperative code [25, 55, 67]. The CLEAR language [28, 29], which provides category-theoretic foundations for the structuring and refinement of algebraic specifications. In CLEAR, specification morphisms are used to structure specifications, and colimits serve to compose specifications. Later examples of this approach include Specware [79] and CASL [24].

Another technique for writing such specifications is the pre- and postcondition style originated by Hoare [65]. In this technique, if a purely mathematical language, such as higher-order logic (as in the PVS theorem prover [117] or Isabelle/HOL [111]) or temporal logic [94] is used for specification of a program, there must be some abstraction function (or relation) that maps the states or events in the program's execution to the abstract states or event models that the specification's formulas mention [66, 82, 147]. Many behavioral specification languages, such as VDM [77], Z [131], Object-Z [120], and OCL [140] have more structuring mechanisms, many of which resemble structures (such as procedures and classes) in programming languages. Besides helping structure larger specifications, such mechanisms constrain what kinds of abstraction functions are considered in proofs.

Carrying these structuring mechanisms farther, by writing specifications as annotations to programs in some particular programming language, yields an *interface specification language* [143]. In such a language, a correct implementation must have both the specified interface and specified behavior (or properties), and thus the relation between a program's state (or events) and the abstract state (or events) described by the specification is much more tightly constrained. Examples of behavioral interface specification languages include the Larch family [58, 143], the Resolve family [45, 116], SPARK [18], Eiffel [98, 99], JML [27, 86], and Spec# [19, 20, 87]. Examples of history-based interface specification languages include Bandera [39] and Java Pathfinder [59]. Interface specification languages, with their close relationship to a programming language, seem likely to be important for the grand challenge, especially in the short term.

4.4 Short-Term Research Goals

The following are some short-term (5-7 years) research goals for specification language research.

4.4.1 Open Languages and Tools

Specification languages should be designed to be extensible and open, so that researchers can more easily experiment with variations and extensions. Tools for specification languages, such as type checkers or verification condition generators, should also be designed with an architecture that makes for easy variation and extension. Tools should also allow different analysis and verification systems easy access to and manipulation of specifications, as these will aid the verification efforts of the grand challenge.

4.4.2 Reasoning about Partial Specifications

Tools for specification languages should make it easy to state and prove logical consequences of specifications. These can be used both for debugging specifications and for proving connections with formalizations of requirements, etc. It should not be necessary to have a complete specification in order to do such reasoning; in other words, it should be possible to reason about partial specifications in which many parts are underspecified, to permit early debugging of the specification.

4.4.3 Refinement

Tools for specification languages should make it easy to state refinements between specifications [14, 61, 100, 101, 42, 79]. There should be automated support for both debugging and proving such refinements, using techniques such as model checking for finding problems with proposed refinements. Section 6 discusses both the posit-and-prove and transformational approaches to proving refinements, and how these techniques can aid verification.

4.4.4 Modularity and Reuse

Specification languages should permit modular descriptions of reusable interfaces. While verified software does not have to be reusable, reusable modules can make it easier to develop larger and more interesting verified software.

4.4.5 Specification of Resources

If non-functional properties, such as time and space consumption, are of interest to the grand challenge, then specification and reasoning techniques for such nonfunctional properties [62, 81, 126] should be further developed and integrated with other kinds of specification.

4.4.6 Interface Specifications

The design of interface specification languages poses some special problems.

Specification and Translation of Assertions Experience with Eiffel [98, 99] and Larch seems to suggest that programmers may find that specification languages like Eiffel, in which assertions are written in the syntax of the programming language, are easier to use than Larch-style languages. (See also Finney's study of mathematical notations [51].) However, other efforts in teaching mathematical specifications to undergraduate students appear to be quite successful, suggesting that the exact notations and language might play a significant role in ease of understandability and use [127]. Thus one research problem is to understand the ease of use of different specification notations (both in practice and for use in verification).

Another research problem is to study how to translate assertions in different languages into logical formula that are useful in reasoning (e.g., in a theorem prover) [33, 6, 86].

Heap Structuring Better techniques for heap structuring, using concepts such as ownership seem to hold promise for aiding verification of pointer-based and object-oriented programs. At the very least, some way to prevent representation exposure [89, 113] seems necessary to do modular reasoning about frame axioms and invariants [78, 104, 105]. Heap structuring also seems helpful for making sense of object invariants in systems built from abstraction layers [19, 87, 106].

It may be that other simplifications in reasoning can be obtained by introducing specifications that further restrict heap structures, for example, to cycle-free-pointers, where such restrictions are appropriate (e.g., in the implementation of lists and trees). What are the right techniques for specifying such restrictions and what kinds of reasoning benefits are obtainable?

Assistance in Writing Specifications To verify large programs that use many modules and libraries, it is often necessary to specify large libraries or code. Many such specification tasks are quite labor-intensive and somewhat unrewarding intellectually. Some automation would help. Tools like Daikon [48, 110] and Houdini [53] have demonstrated that it is possible to recover some formal specifications from code using various heuristics. It might be interesting to infer specifications from examples or directly from test cases. A research goal would be to have such tools work with user-specified abstractions, so that they could be used to more quickly write more abstract specifications. Or perhaps some automatic abstraction heuristics could be used. An environment for writing specifications could allow users to edit out some cases in a specification, to achieve more abstraction by underspecification.

New Language Features If more advanced programming languages are of interest to the grand challenge project, then how to specify properties of programs that use advanced features, like advice in aspect-oriented languages, will be important.

4.5 Long-Term Research Goals

The following are some longer term (8-15 years) goals for specification languages.

4.5.1 Integration of Data and Control

An important challenge for specification language design is to integrate the two disparate worlds of state-based and history-based (or event-based) specification languages. Typically, specification languages either focus on sequential programs and describe properties of data values, or they focus on concurrent programs and described properties of event histories. However, complete verification of concurrent programs demands reasoning about both data and control. Some potential approaches are to use atomicity [91, 119] or to use transitions over relations.

4.5.2 Traceability

Links between requirements and detailed design specifications should be able to be explicitly stated and reasoned about. One approach may be to develop techniques for stating and proving refinement relationships between (particular pairs of) requirement and specification languages. Another approach might be to design languages that are good both for formalizing requirements and for specification of the detailed design.

4.5.3 Tool Frameworks that Support Integration

Frameworks that would make it easy to build tools for specification languages and to integrate different tools for reasoning about specifications should be a long-term goal. Integration among reasoning tools, such as model checkers and theorem provers, would also be helpful.

4.5.4 Interface Specification Language Design

A theory of how to design interface specification languages should be developed that allows a new specification language to be quickly designed for a new programming language, at least within a fixed set of programming paradigms. Ultimately such a theory should extend beyond the imperative and object-oriented paradigms to other paradigms of interest to the grand challenge.

Along the same lines, it may also be useful to understand how to tailor the design of such a language to a specific architectural style. This would potentially help with verification of programs written in such styles.

5. Research in Program Generation

This section was mainly written by: Gary T. Leavens, Don Batory, Alessandro Coglio, and Douglas R. Smith.

5.1 Background on Program Generation

A program *generator* [40] is a tool that produces code from some higher-level description of the code. Conventional compilers for languages such as C and Java fit this characterization, because they generate lower-level assembly or bytecode from higher-level programming languages. However, the term "program generator" is typically used for tools that produce code in relatively high-level languages such as C and Java, and where the higher-level description of the code is a specification. Nonetheless, we do not rule out the view of compilers as generators; in fact, the research directions advocated here apply to compilers as well.

A program generator operates on the syntax of the source (specification) and target (code) languages. Roughly speaking, the generator reads the specification and writes the code, i.e. it *transforms* the specification into the code. Program generators are often written in conventional languages such as C or Java; they manipulate

data structures that encode abstract syntax trees of the source and target languages. The pattern matching featured by languages like ML and Haskell provides a convenient way to implement syntactic transformations. Languages like Refine [73] and Stratego [134] provide even more convenient features to implement syntactic transformations in a more declarative way, by means of rewriting rules, strategies, and quotation/anti-quotation pattern matching.

5.2 Relation to Model-Driven Development

The premise of Model-Driven Development (MDD) [21, 26, 136] is that a program has multiple representations, expressed as models. Transformations will update models and map models to other models, and compose models.[3] Since code is the most important kind of model in MDD, MDD falls within the scope of the program generation area.

5.3 Motivation for Program Generation

Program generation is useful for at least two reasons [40]. One is *productivity*: instead of writing the code directly, the developer writes and maintains the specification, which is supposedly shorter and easier to read and write than the code. The other reason, which is more relevant to our context, is that the code can be generated in such a way as to be *automatically verified*; that is, it will be correct with respect to the specification. The research directions advocated here aim at automatic verification.

Program generation also fits well with the use of software product lines. A software product line describes a family of programs [22, 40]. Using a product line gives a significant reduction in artificial complexity, more regularity and structure in a program's modules, and leads to modules are more likely to encapsulate increments in program functionality. All three are key requirements for module reusability, large scale synthesis, and verification. Showing how to verify software product lines would illustrate the connection between scale, design, and verification.

5.4 Problem: Verified Program Generation

The problem is that even when using the most declarative syntax transformation languages available, the semantics of the source specification and of target code are not directly "represented" in the program generator. Thus, it is very possible to generate code that is incorrect with respect to the specification, by doing "wrong" syntactic transformations. Achieving correctness is thus the overriding research problem for program generation with respect to the grand challenge.

5.5 Problem: Scalability

There has been significant progress in algorithm synthesis and automatic design optimization [129], especially in restricted domains; examples include Planware [23], Amphion [133], and AutoBayes [52]. While continued progress in the generation of moderate size programs can be expected, a scalable approach to program generation must also focus on how to generate verified compositions of reusable modules. A vast majority of practitioners and researchers who are automating parts of program development are building tools that are compositional in nature. COM, Java server pages, and Enterprise Java Beans are examples. These tools stitch code modules together to synthesize larger modules. Most code modules are written by hand, but some (e.g., parsers or boiler-plate interfaces) are generated by simple tools. In effect, the specification languages for these code synthesizers are akin to module interconnection languages.

[3] Thus, roughly speaking, a model is an object and a transformation is a method.

A module is more than just code; it encapsulates several different kinds of information: specifications, code, formal models from which properties can be inferred, documentation, performance models, etc. Specifications and performance models are especially important for verification. It is thus important to synthesize such information for generated compositions of modules [22].

A well-known example of the above is the work on query optimization in relational databases [123]. An optimizer maps a declarative specification (e.g., a SQL SELECT statement) to an efficient implementation. A SELECT statement is first mapped to a relational algebra expression, the expression is optimized, and then code is generated from the optimized expression. Each relational algebra operation is a module, and a relational algebra expression is a composition of modules that represents a query evaluation program. Each module (operation) encapsulates two different representations: a performance model (which evaluates the efficiency of the operation) and code (to implement the operation). The query optimizer uses only the performance model of an operation to deduce the most efficient composition. The program synthesizer uses only the code representation to generate the implementation. A similar organization (i.e., modules containing multiple formal models) will be needed for program verification.

5.6 Short-Term Research Goals

The following are some short-term (5-7 year) research goals in the area of program generation.

5.6.1 Formalizing Language Semantics

The first step to establish the correctness of generated code is to formalize the semantics of the source and target language, along with a notion of what it means for an artifact in the target language (the code) to be correct with respect to an artifact in the source language (the specification). For example, the correctness notion could be that the two artifacts have the same observable behavior (where the notion of observable behavior must be also formalized). These formalizations should be developed in a suitably expressive logical language with a formal proof theory, such as the languages found in mundane theorem provers. Examples include Project Bali [112] and the LOOP Project [75, 137], both of which formalize Java.

5.6.2 Tool Development

Current (meta-)languages and tools [73, 134] do not deal with the semantics and proof aspects of transformations, but only with their syntax. Thus, an important research direction is to design languages and tools, by which one can more directly represent semantics and generate proofs and code in an integrated fashion.

5.6.3 Certified Code Generation

Instead of directly verifying the generator, a promising approach is to have the generator produce, along with the code, a machine-checkable proof of the correctness of the output code with respect to the input specification [36, 37, 107]. The proof should use the inference rules of the logical language in which the semantics of source and target language, as well as the notion of correctness, are formalized.

Then, as in the well-known proof-carrying code technique [108], the proof is checked by a simple proof checker, so that trust is shifted from a large and complex generator to a small and simple checker.

5.6.4 Transformation Patterns

Proof-generating transformation patterns, which will emerge from applying program generation in practice should be cataloged; e.g.

taxonomies of algorithm theories and datatype refinements [130]. These catalogs will help others apply the ideas and build tools more quickly.

5.6.5 Better Algorithms to Aid in Program Generation

To apply general design principles and transformations to a concrete specification requires some analysis (to verify applicability conditions) and constructive inference (to extract expressions to fill in design templates).

More practical program generation requires low-order polynomial time algorithms for analysis and constraint solving. A promising approach is to compose constraint-solvers and decision-procedures for various specialized theories. Static analysis can also sometimes provide a fast alternative to search-based theorem provers.

5.7 Long-Term Research Goals in Program Generation

The following are some long-term (8-15 year) goals for research in program generation.

5.7.1 Scalability

To allow scalability of program generation, techniques for generating compositional, well-structured designs are needed in each application domain. A complementary need is for techniques for composing properties, specifications, and other non-code information in modules. It must be clear how such compositions preserve (or affect) properties of interest.

5.7.2 Taxonomy of Proof-Generating Transformations

A collection of proof-generating patterns (or templates) should be made into a library, categorized by various dimensions, such as application domain, source and target language, etc. This knowledge would make it easier to develop future program generators.

5.7.3 Better Tools and Frameworks

Researchers could design better languages, tools, and frameworks, to ease the task of building future program generators. Such tools could both more directly support proof generation and could also ease the proof of correctness for the program generator itself.

Such tools and languages could also more directly support proof-generating patterns.

5.7.4 Factoring the Certification Process

Establish sound techniques for incorporating formal proofs into the certification process for program generators, in order to eliminate some testing and reduce the need for other kinds of testing. (Current practice is to perform extensive and expensive testing, both to validate the generated code's functionality and performance, and to test for vulnerabilities and flaws along various code paths.) Given a complete specification from which the code is generated, together with a proof of consistency between code and specification, there should be little need to perform path testing to reveal flaws. There will still be a need to test that the specification meets intentions, but that can be a more specialized activity. Also, those requirements that are not treated during generation or refinement (e.g. performance concerns) would also still need to be tested.

5.7.5 Allow Update of Running Systems

For embedded systems, it is often necessary to update (fix) the code while the system is running. Supporting such updates in a system where code is generated may be a matter of generating the code to allow for eventual update.

5.7.6 More Manual Control

To allow users to operate outside a limited domain to some extent, program generators could be designed to allow more manual input, making them a blend of a program generator and a correctness by construction system, as described in the next section.

6. Research in Correctness by Construction

This section was mainly written by: Michael Butler, Gary T. Leavens, Eric Hehner, Murali Sitaraman, Jean-Raymond Abrial, and Cliff Jones.

6.1 Motivation

Much discussion on the need for a powerful program verifier seems to contain the following underlying assumptions:

- That a program verifier will be used mostly to verify completed programs.
- That when verification fails it is because the program contains errors.

While a powerful program verifier is a very valuable tool for programmers, it does not help them construct a correct program in the first place, nor does it help document and explain decisions (e.g., those motivated by efficiency considerations) made in existing code.

Equally important, the correctness of any verification is dependent on the validity of the formal properties against which a program is checked. Since we cannot, in general, guarantee that such properties are what users really want, we will, in the remainder of this section use the phrase "verification by construction," instead of the more common phrase "correctness by construction," to emphasize the potential problems with the initial specification.

The verification by construction approach helps developers who want to construct verified software systems by addressing the following questions:

Q1 How do we construct models and properties against which to verify our software?

Q2 How do we ensure that our models and properties properly reflect the requirements on the system?

Q3 How do we take account of the environment in which our software is intended to operate?

Q4 How do we construct our software so that the verification will succeed?

In the following, we will largely ignore question Q2, since it too large and important to be included in our grand challenge; it would constitute a grand challenge on its own.

As can be seen from the other questions, the verification by construction approach broadens the focus away from just verifying a finished *product* to analysis of models at all stages of the development *process*. It encourages verification of designs and not just verification of programs. Verification of designs may lead to a greater payoff than just verifying programs. Introducing formal modeling early in the development cycle helps to identify problems earlier, long before any code is developed, thus helping to avoid expensive later rework.

As well as supporting verification of designs and implementations, the formal modeling languages used in verification by construction encourage a rational design process. We contend that the use of good abstractions and simple mathematical structures in modeling, and reuse of modules with specifications can lead to cleaner, more rational system architectures that are easier to verify

(and maintain) than architectures developed using less disciplined approaches.

6.2 How is Verification by Construction Achieved?

Verification by construction can be achieved by having a formal framework in which models are constructed at multiple levels of abstraction; each level of abstraction is refined by the one below, and this refinement relationships is documented by an abstraction relation (typically in the form of a gluing invariant) [1, 3, 14, 42, 61, 77, 83, 100, 101, 102, 128]. The highest levels of abstraction are used to express the required behavior in terms of the problem domain. The closer it is to the problem domain, the easier it is to validate against the informal requirements, i.e., ensure that it is the right specification. The lowest level of abstraction corresponds to either an implementation, a specification from which an efficient implementation can be derived automatically, or to a specification realized in hardware.

Also critical in this framework are mechanisms for composing and decomposing models. Composition can be useful for building up specifications by combining models incorporating different requirements. Decomposition is important for relating system models to architectures of subsystem models and also for subsequent separate refinement of subsystems [5, 2, 15, 16, 30, 43].

Ensuring that a model $M2$ refines or implements $M1$ requires bridging the abstraction gap between them. Typically there is a large abstraction gap between a good formal specification, i.e., one that is easy to validate against the requirements, and an efficient implementation.

Verification by construction does not require that such abstraction gaps be bridged by a series of (small) transformations, done at the time that $M2$ is derived from $M1$, each step of which guarantees refinement. While this kind of *transformational approach* is valuable [61, 100, 101, 102], verification by construction also includes a *posit-and-prove approach*, in which the developer provides both $M1$ and $M2$ and uses tools to verify that $M1$ is refined by $M2$ [1, 3, 77, 83]. The difference is not great, especially since in the transformational approach, the transformation applied might result in the generation of side conditions that will need to be verified. Conversely, if the abstraction gap between $M1$ and $M2$ is small enough, or if the properties involved are limited, a tool can generate proof obligations that can be verified, perhaps automatically using model checkers or powerful theorem provers. Tools are important for the transformational approach, but tools are also useful in the posit-and-prove approach, for example, to help one discover ancillary properties, such as invariants.

Through refinement it is often possible to model and reason about how a strategy solves a problem in an abstract way using abstract specifications that encapsulate algorithms and data structures. At higher levels of abstraction one can focus reasoning on design choices closely related to the problem domain and less on coding details. These abstract specifications can then be optimized through refinements that select implementation modules, or that introduce more concrete algorithms and data structures. Reasoning about these optimizing refinements no longer requires reasoning about the original problem as this will have been dealt with by the earlier refinement.

In this way, by keeping the models as abstract as possible at each level, or by reusing modules, one will often have simpler proof obligations to discharge. This contrasts with the situation that obtains when one verifies a program (without annotations) and without intermediate refinement steps. In doing such a proof, one must reason about a number of issues simultaneously: the problem to be solved, the data structures, and the algorithms used in the solution. Using a series of refinement steps helps factor out and modularize such decisions, allowing them to be dealt with separately. This often simplifies proof obligations and helps make reasoning made more manageable.

When using refinement, one does not necessarily distinguish between properties and models. Essentially we are working with models in a modeling language and the important property to be proved of some model $M2$ is that it is a refinement of some other model $M1$. So the answer to the question "what properties should we prove of a model?" is "those properties that help show that it is a refinement of its abstraction." For the most abstract models, the important property is that they satisfy the requirements of the problem domain. This is an informal check which can sometimes be aided by checking required ancillary properties. With a refinement approach the "creative" input in a development is a collection of explicit models at different levels of abstraction. The invention of ancillary properties is dictated by the need to prove refinement between these explicit models. Creating models at different levels of abstraction, or reusing previously-available modules with specifications, fits well with an engineering approach.

6.3 The Goal of Verification by Construction

Existing theories, languages, proof techniques and tools for verification by construction need to be evolved to address more fully questions Q1, Q3, and Q4 above. This will lead to powerful tools that will:

- Support the construction of models (specifications, designs, programs) at multiple levels of abstraction,
- Support the verification of refinement between models,
- Support the verification of modules built from other modules, and
- Support verified construction of complex systems consisting of software and environments in which software operates.

The feasibility of these results will be demonstrated through their application to the development of complex software systems. The long term directions described later are intended to lead toward these goals. We also suggest some short-term directions which can build immediately on existing work in the area and will contribute to elaboration of the longer term problems and their solutions.

6.4 Short-Term Research Directions

The following are some short-term (5-7 year) research goals.

6.4.1 Range of Case Studies

Develop and open for scrutiny several case studies of verification by construction, using existing techniques and tools. These case studies should be selected from the class of verification problems considered for the grand challenge project, and might include some of the overall project's challenge problems. Some case studies should focus on verification of modules. In all cases, the studies will help identify particular areas for improvement in the approaches.

Researchers should consider developments in which not every part of a design is mapped down to fresh code, rather some parts are implemented by legacy systems. The specifications of the legacy parts need not appear at the highest level, rather they could be introduced in later refinement steps. The correctness of the overall system implementation with respect to the abstract specification would be conditional on the assumption that any legacy parts satisfy their specification; an assumption whose discharge may be tackled by other parts of the grand challenge.

Existing research projects and efforts have made requirements documents and formal specifications available and these could be used as starting points and built on further [95, 118, 132].

6.4.2 Links between tools

Build links between existing tools to support verification by construction. In particular, build links between proof obligation generators for refinement checking (as found in B and Z for example) and

- the latest powerful theorem provers, model checkers and SAT solvers, and
- automated invariant generation tools (such as Daikon [48]).

Existing work that could be used as a basis for tool integration work includes the Eclipse-based Rodin platform for refinement [118] and the Community Z tools initiative [41].

These experiments will guide the long term direction of a unified tools framework for verification by construction.

6.4.3 Programming Language Mappings

Models at low levels of abstraction need to be converted to executable software. The effective way of doing this is through tool-supported mappings to existing programming languages such as Ada, Eiffel, Java and C#. In the medium term these mappings should be pragmatic and their soundness provided through informal arguments. To increase confidence in the resulting code, the mappings should also generate appropriate formal annotations (e.g., SPARK, Eiffel, JML or Spec# assertions) from the models and ancillary properties. This allows the generated code and annotations to be analyzed using existing program analysis tools. For some applications or domains it may be appropriate to consider mapping low-level models direct to byte code by-passing the compiler. Since the code generation problem is essentially the problem of program generation, the research directions pointed out in Section 5 also apply to this problem.

Examples of automated mapping of models to code are found in AtelierB [35], which supports generation of C and Ada code from low level B models, and the B-Toolkit [13], which supports generation of C code from low level B models.

6.5 Long-Term Research Directions

The following are some long-term (8-15 year) research directions in the verification by construction approach.

6.5.1 Evolution + Refinement

Refinement is never purely top down from most to least abstract, because it is difficult to get the abstract model precisely right. One usually starts with an idealistic abstract model because that is easy to define. As refinement proceeds and more architectural and environmental details are addressed it often becomes clearer how the ideal abstract model needs to be modified to reflect reality better. Modifications to some level of abstraction will ripple up and down the refinement chain. This is not a weakness of the refinement approach per se, rather a reflection of the reality of engineering of complex systems. The theories, languages, proof techniques and tools need to support evolution of designs during and after development with minimal effort.

6.5.2 Complex system design

Control systems, interactive systems, and distributed systems involve multiple agents (users, environments, new programs, legacy code) all of which contribute to the correctness of a system. Individually the agents may be very complex, so reasoning about compositions of agents in all their gory detail may be infeasible. Instead, there is evidence that it will be feasible to reason about complex systems through good use of abstraction, refinement and module composition [31, 32, 60].

The extent to which one must consider the operating environment when developing software depends on where one draws the boundaries of the system. To reason about the validity of any fault tolerance mechanisms, it is useful to include some abstraction of the environment in the formal models in order to verify the effectiveness of these mechanisms. For example, when reasoning about the effectiveness of a security protocol, it is usual to include some abstraction of an attacker. The goal is not to implement the attacker, rather it is to show that the protocol achieves its security goal even in the presence of an attacker, under some assumptions about attacker behavior. These assumptions about attacker behavior can be encoded in the formal abstraction of the attacker.

6.5.3 Richer Refinement Theories

Within a particular framework there may be differing strengths of refinement. A weaker notion might capture the preservation of safety behavior, while stronger notions might capture preservation of liveness and/or fairness.

Another important dimension is resource usage. A theory of refinement should ideally allow one to prove tight bounds on resources, while still permitting abstract reasoning. Specifications of resource usage should also not require reverification when the computing platform is changed.

The refinement relation should enjoy some form of transitivity. Refinement is based on comparing models according to some notion of what can be observed about them, and it is useful to be able to modify what can be observed at different levels of abstraction. In particular, the interface to a system is usually described abstractly and may need to be made much more concrete at decomposition or implementation levels. In such cases, the observable behavior is not directly comparable, but needs to be compared via some mapping and transitivity of refinement is via composition of mappings.

6.5.4 Refinement Patterns

A halfway house between transformational and posit-and-prove can be envisaged, where certain patterns of model and refinement can be captured and used in the construction of refinements. This is a more pragmatic idea than transformational refinement in that the pattern might not guarantee the correctness of the refinement. Instead $M2$ would be constructed from $M1$ by application of a pattern and the correctness of the refinement would be proved in the usual posit-and-prove way. Ideally the pattern should provide much of the ancillary properties (e.g., invariants, tactics) required to complete the proof, or at least an indication of what kinds of properties might be needed.

The aim of using such patterns is to minimize verification effort when applying refinement. A research goal is to identify such patterns through a range of case studies and supporting the application of the patterns with tools.

6.5.5 Integrated Tools Framework

To a large extent the theory needed to support verification by construction already exists. The challenge is to provide a powerful set of tools to support abstraction, refinement, decomposition and proof. Tools should strive to achieve as much integration as possible and avoid isolation. Such tools should also exploit as much of the existing work in theorem proving and model checking as possible and should be designed in anticipation of future advances in these areas. The same can be said for using state-of-the-art methods in programming language design, program verification, and automated program generation. As they evolve, the support tools should be applied to the development of interesting software-based systems.

7. Research in Programming Languages

This section was mainly written by Gary T. Leavens, Simon Peyton-Jones, Dale Miller, and Aaron Stump.

7.1 Assumptions and Scope

In this section we assume that imperative languages are of interest. This is not meant to exclude research on other paradigms. For example, functional languages and domain-specific languages each have their own advantages for verification.

Also, this roadmap assumes that verifying a compiler (or other programming language tools) is not a goal of the grand challenge. This is not to say that researchers in programming languages are not concerned about correctness of the tools they produce. On the contrary, it is standard, for example, for all type systems in programming language research papers to come with a formal proof of correctness. (The recent POPLmark challenge calls for such proofs to be written in machine-checkable form [12].) However, it seems likely that such verification problems will be outside the emphasized areas of the grand challenge.

7.2 Programming Language Approaches to Verification

Aside from using refinement to derive programs that are "correct by construction," program generation (including certifying compilers [103]), and direct use of semantics[4] we know of the following main approaches that directly aid the verification of software.

7.2.1 Type systems

Types are weak specifications [72] that are automatically checked by compilers.

Type systems are a long-standing topic of interest in programming language research. Early work in type theory [38, 114] showed how dependent types allow a type system to express complete functional specifications as well as constructive proofs of program correctness, at many levels of detail. Examples of dependently typed programming languages where this idea is explored include ATS, RSP1, Ωmega, Epigram, Cayenne, and Martin-Löf type theory [11, 34, 96, 115, 125, 142]. Work by Voda has similar goals [139].

7.2.2 Program Analysis

Program analysis gathers information that safely approximates what programs will do at runtime. Static type systems are a special case of static analysis [109], but program analysis is not restricted to obtaining information about types. Like type checking, program analysis can be seen as a way of doing weak verification; for example shape analysis can be seen as a way of "computing a safe approximation to a statement's strongest postcondition" [121, p. 284].

Many interesting formal methods tools have checked various properties using static analyses of various sorts. Examples include partial correctness (checked by, e.g., TVLA [88]), conformance to API protocols (checked by SLAM [17]), memory safety (checked by Prefix and Prefast [85] and LCLint [49]),and absence of race conditions (checked by Autolocker [97]). (There are also several systems that look for error patterns, including Metal [47] and Findbugs [71].)

7.2.3 Assertions

Assertions are logical properties of a system, usually expressed in some extension of predicate logic or temporal logic. Assertions can specify post-conditions for methods, invariant properties for objects, and protocols that API calls should obey.

There has also been a historical strand of work that directly adds Hoare-style specification and verification to programming languages. Gypsy [7] and Alphard [64, 93, 124] are early examples. The Euclid language [84, 92] was notable along these lines; Euclid omitted or restricted several features of Pascal, as an aid to formal verification. For example, Euclid introduced the notion of heap regions as a way to get some control on aliasing, and also prohibited overlap among the parameters to procedure calls. The SPARK subset of Ada [18] continues this tradition. Perhaps the most successful such language is Eiffel [98, 99], which takes a very pragmatic approach to specification and focuses on run-time assertion checking. The ESC system [44] is an interesting hybrid, since it uses assertions, but in some ways is more like a static analysis system.

7.3 Problems with Current Approaches

We see several overall problems with the above approaches to directly aiding verification.

7.3.1 Effort Needed for Verification

Programmers are less likely to use a technique if it does not allow them to suppress proofs or details.

For example, when using a dependent type system, the need to provide proofs of correctness along with executable code limits the appeal of dependent type systems, since this demands substantially more work than needed in currently popular programming languages, and the proofs are not optional. A potential way out of this difficulty for dependent types is shown by Dependent ML, which, while also based on dependent types, has the goal of checking properties without programmer-supplied proofs [146]. Thus one research direction would be to explore how to gain the advantages of dependent type systems without the need to explicitly supply proofs.

Similarly, when using assertions, one often has to specify many properties in addition to the property of interest. The Bandera system [39] and SLAM [17] both use slicing [135, 141] before model checking to avoid state space explosion. An interesting research direction would be to use slicing more extensively in other kinds of verification.

7.3.2 Lack of Extensibility

Current programming languages often fix a particular notation and verification technique, and do not allow users to modify or add to it. For example, it is hard to find a single level of specification beyond types that all programmers would agree is worthwhile. Indeed one might criticize most languages where types play a central role for taking an important concept and freezing it. That is, if types are so important, why do languages (like Java, Standard ML, and Haskell) allow for just one type system? It would seem more valuable to first see a programming language as describing an untyped computation and then allow for various ways to infer the various kinds of typings as well as other static properties. Also, types are open-ended: there is no one best type system, and researchers will always be making new proposals for better systems. Similar remarks apply to assertion languages and static analysis frameworks.

Thus a research direction would be to find a more open architecture for programming language definition (and implementation) that allows the use of multiple type systems, multiple static analyses and multiple different kinds of assertions. Ideally, it would be best to allow these different kinds of annotations to interact with each other. For example, it would be great if specifications written using assertions could refer to properties (such as what variables are assigned) that are covered by a static analysis.

[4] Besides use of Hoare logic, or "axiomatic semantics" [65] one can also specify and verify software using denotational [122] or operational semantics [10]. However, these styles are not typically well-suited for specification purposes, at least for imperative programs.

7.4 Short-Term Research Directions

In this section we describe some ideas for research directions in the short term (5-7 years), with two goals: directly supporting specification and verification, and eliminating much of its drudgery by eliminating common problems.

7.4.1 Supporting Specification and Verification Annotations

Basic language features for supporting specification and verification have been discussed above, in the section on specification languages. These should be investigated for their interactions with programming languages and systems. For example to what extent can optimizing compilers and other kinds of static analysis make use of such information?

There is one important aspect of programming language designs that could greatly ease specification and verification, which is to design languages so that expressions (or at least some identifiable subset of expressions) have no side effects. Side effects in expressions make it difficult to follow Eiffel's lead in using programming language expressions in assertions [98, 99]. While some languages in the Pascal family (including Euclid [84] and Ada [74]) already do this, based on Pascal's separation of functions and procedures [76, 145], it deserves to be more widely followed.

Tools for programming languages could also be designed to better support specification and verification annotations. Ideally annotations should be provided in an open manner, which would allow users and tool providers to add to the set of annotations. Meta-information such as the annotations of Java and C# are useful for this purpose, but are weak in that they do not allow full use of the language's expression syntax and are not hierarchical, and thus do not support rich syntax for specification. Furthermore, to support typing and verification, annotations must be permitted at all levels of syntax; for example, adding annotations to statements is necessary to specify the effect of a loop.

Another way that programming languages could aid working with annotations is if they would allow annotations to substitute for code. That is, a tool should be able to manipulate a program in which some parts are not implemented in the language, but are merely specified with some annotations. Achieving this kind of "specification closure" would help researchers working on compilers and interface specification.

7.4.2 Eliminating Drudgery in Specification and Verification

Programming language design can reduce the cost of specification and verification by keeping the language simple, by automating more of the work (e.g., by propagating type information), and by eliminating common errors. (Eliminating common errors would also help make programs more reliable, even if programmers do not use verification techniques.) Historical examples include elimination of dangling references by the use of garbage collection, encapsulation of iteration idioms (such as map or for loops), type systems that avoid null pointer dereferences (as in Lisp or CLU [90] and the work of Fähndrich and Leino [50]), and SPARK's elimination of conditional data flow errors (such as reading from uninitialized variables) [18].

It seems like a fruitful research direction to try to eliminate other common errors, such as array indexing errors, perhaps by using dependent types or by using modulo arithmetic to map all integers back to defined array elements.

It is perhaps also useful to look closely at verification technology and to see what features of programming languages cause the most trouble for verification efforts. Following the lead of Euclid [84, 92], and SPARK [18], it may be interesting to try to design languages (or subsets) without such features. Another way of putting this research question is: what features that are not in languages like SPARK can now be handled without causing difficulty for verification?

Some common errors may not be problems with language itself, but may be problems with use of libraries or simply mistakes that programmers commonly make. Can rules for automatically finding such common errors, as is done in Metal [47] and Findbugs [71], be added to a programming language, under the control of tool builders or users? One simple direction for achieving allowing such extensions may be to add features like `declare error` and `declare warning` from AspectJ [9, 80], although such mechanisms may be too simple to handle all the kinds of bugs detected by such tools.

7.5 Long-Term Directions

In the longer term (8-15 years), one can contemplate more integration instead of just promoting extensible tools to aid specification and verification.

7.5.1 Integration of Tools and Languages

Make the programming language's compiler a platform that makes it easier to build and integrate multiple specification and verification tools. Eclipse may be an example of the kind of development platform that is headed in the right direction, but it would need to be substantially enhanced to allow for the addition of multiple tools and to support their integration.

7.5.2 More Integration of Types and Specifications

Another goal is to find potential "sweet spots" that are intermediate between full functional (or control) specifications and type systems. Dependent types might be helpful as a technology for verification of such partial specifications, but they must be made much more accessible to programmers.

7.5.3 Integration of Rich Static Checking

Support the integration of rich static checking (verification of partial specifications) in the programming language. Researchers could explore taking some existing programming languages and providing support for flexible deduction to be allowed on source code and any assertions that are associated with that code (either in the code as type declarations, loop invariants, etc.) or separately.

Allow for possible community-based inference to be performed on a module-by-module level. Provide the elements of a computational logic that could help in performing basic source-level manipulations such as substitutions and unification. An example of such a scheme can be found in the work of the Ciao system [63].

8. Conclusions

This roadmap has described ways that researchers in four areas — specification languages, program generation, correctness by construction, and programming languages — might help the verified software grand challenge project. Researchers in these areas need challenge problems to be described in many different ways, including requirements, source code, and test cases.

In the short term, a common research goal shared by all four areas is building extensible tool frameworks that would allow researchers to more easily implement specification and verification tools. This could lead to the exploration of more research ideas and to more careful evaluation of these ideas.

In the long term, researchers can try to consolidate the best of these ideas into new theories and tools.

Acknowledgments

Thanks to the members of IFIP Working Group 2.3 (Programming Methodology) for discussions and for comments on an earlier

draft of this material, presented at the Brugges meeting in March 2006. Special thanks to Michael Jackson (the one involved in IFIP WG 2.3) for his advice on narrowing the scope of this roadmap: "specialize!" (Yes, it was even broader previously.) Thanks to Shriram Krishnamurthi for several discussions and suggestions. Thanks to Rod Chapman for comments, ideas, and corrections relating to SPARK. Thanks also to the participants at the SRI Mini-Conference on Verified Software (April 1–2, 2006) and to the Dagstuhl workshop on "The Challenge of Software Verification" (July 10–13, 2006) for additional comments and suggestions. Thanks to the US National Science Foundation for grants supporting these meetings and for supporting, in part, the work of Leavens (CCF-0428078 and CCF-0429567), Fisler (CCR-0132659 and CCR-0305834), Sitaraman (CCR-0113181), and Stump (CCF-0448275).

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. Technical Report 29, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, Aug. 1988. A shorter version appeared in *Proceedings of the LICS Conference*, Edinburgh, Scotland, July 1988.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.

[3] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[4] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[5] J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, XXI, 2006.

[6] Ádám Darvas and P. Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[7] A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Choen, C. G. Hoch, and R. E. Wells. Gypsy: a language for specification and implementation of verifiable programs. *ACM SIGPLAN Notices*, 12(3):1–10, Mar. 1977. Proceedings of the ACM Conference on Language Design for Reliable Software.

[8] K. Araki, S. Gnesi, and D. Mandrioli, editors. *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[9] AspectJ Team. The AspectJ programming guide. Available from `http://eclipse.org/aspectj`, Oct. 2004.

[10] E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 51–136. Springer-Verlag, New York, NY, 1991.

[11] L. Augustsson. Cayenne — a language with dependent types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 239–250. ACM, June 1999.

[12] B. E. Aydemier, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, Lecture Notes in Computer Science. Springer-Verlag, June 2005.

[13] B-Core (UK) Limited. B-toolkit manuals. `http://www.b-core.com`, 1999.

[14] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[15] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, LNCS 430. Springer-Verlag, 1990.

[16] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13(2-3):133–180, 1990.

[17] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, Jan. 16–18, 2002.

[18] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, New York, NY, 2003.

[19] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[20] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[21] D. Batory. Multi-level models in model driven development, product-lines, and metaprogramming. *IBM Syst. J.*, 3, 2006.

[22] D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[23] M. Becker, L. Gilham, and D. R. Smith. Planware II: Synthesis of schedulers for complex resource systems. Technical report, Kestrel Technology, 2003.

[24] M. Bidoit and P. D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer-Verlag, 2004.

[25] H.-J. Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Trans. Prog. Lang. Syst.*, 7(4):637–655, Oct. 1985.

[26] G. Booch. Growing the uml. *Software and Systems Modeling*, 1(2):157–160, Dec. 2002.

[27] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[28] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Fifth International Joint Conference on Artifical Intelligence, MIT, Cambridge, Mass.*, volume 2, pages 1045–1058. IJCAI-77, Department of Computer Science, Carnegie Mellon, Pittsburgh, Aug. 1977.

[29] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. In *Abstract Software Specification, Copenhagen Winter School*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer-Verlag, New York, NY, 1980. Also University of Edinburgh, Department of Computer Science, Internal Report, CSR-65-80, Feb, 1980.

[30] M. J. Butler. Stepwise refinement of communicating systems. *Sci. Comput. Programming*, 27(2):139–173, 1996.

[31] M. J. Butler. On the use of data refinement in the development of secure communications systems. *Formal Aspects of Computing*, 14(1):2–34, 2002.

[32] M. J. Butler, E. Sekerinski, and K. Sere. An action system approach to the steam boiler problem. In Abrial et al. [4], pages 129–148.

[33] J. Charles. Adding native specifications to JML. In *Workshop on*

Formal Techniques for Java-like Programs (FTfJP), July 2006.

[34] C. Chen and H. Xi. Combining programming with theorem proving. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, Sept. 2005.

[35] ClearSy. Atelier B, user and reference manuals. `http://tinyurl.com/lkj72`, 1996.

[36] A. Coglio and C. Green. A constructive approach to correctness, exemplified by a generator for certified Java Card appplets. In *Proc. IFIP Working Conference on Verified Software: Tools, Techniques, and Experiments*, Oct. 2005.

[37] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 95–107, New York, NY, USA, 2000. ACM Press.

[38] R. L. Constable. Assigning meaning to proofs: a semantic basis for problem solving environments. In M. Broy, editor, *Constructive Methods in Computing Science*, volume F55 of *NATO ASI Series*, pages 63–91. Springer-Verlag, New York, NY, 1989.

[39] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, New York, NY, June 2000. ACM Press.

[40] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[41] CZT Partners. Community Z tools. `http://czt.sourceforge.net/`, 2006.

[42] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, NY, 1998.

[43] W. P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[44] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec 1998.

[45] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. Part II: Specifying components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):29–39, Oct 1994.

[46] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, New York, NY, 1985.

[47] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th Symp. OS Design and Int'l (OSDI 2000)*, pages 1–16. ACM, 2000.

[48] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[49] D. Evans. Static detection of dynamic memory errors. *ACM SIGPLAN Notices*, 31(5):44–53, May 1996. Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[50] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented langauge. In *OOPSLA '03: Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 38(11) of *ACM SIGPLAN Notices*, pages 302–312, New York, NY, Nov. 2003. ACM.

[51] K. Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, Feb. 1996.

[52] B. Fischer and J. Schumann. Generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(3):483–508, 2003.

[53] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, Mar. 2001.

[54] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, Jan. 1985.

[55] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, MA, 1996.

[56] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.

[57] J. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[58] J. V. Guttag, J. J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[59] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), Apr. 2000.

[60] I. J. Hayes, M. Jackson, and C. B. Jones. Determining the specification of a control system from that of its environment. In Araki et al. [8], pages 154–169.

[61] E. C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. Available from `http://www.cs.utoronto.ca/~hehner/aPToP`.

[62] E. C. R. Hehner. Formalization of time and space. *Formal Aspects of Computing*, 10:290–306, 1998.

[63] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Program.*, 58(1-2):115–140, 2005.

[64] P. Hilfinger, G. Feldman, I. K. Robert Fitzgerald, R. L. London, K. V. S. Prasad, V. R. Prasad, J. Rosenberg, M. Shaw, and W. A. W. (editor). (preliminary) an informal definition of Alphard. Technical Report CMU-CS-78-105, School of Computer Science, Carnegie Mellon University, 1978.

[65] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580,583, Oct. 1969.

[66] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[67] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, Aug. 1987. See corrections in the September 1987 CACM.

[68] C. A. R. Hoare, N. Shankar, and J. Misra, editors. *Proc. IFIP Working Conference on Verified Software: Tools, Techniques, and Experiments*, Zürich, Switzerland, Oct. 2005.

[69] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, Jan. 2003.

[70] T. Hoare, J. Misra, and N. Shankar. The IFIP working conference on verified software: Theories, tools, experiments. `http://tinyurl.com/nrhdl`, Oct. 2005.

[71] D. Hovemeyer. *Simple and Effective Static Analysis to Find Bugs*. PhD thesis, University of Maryland, July 2005.

[72] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Inc., New York, NY, 1980.

[73] R. Inc. Software Refinery, Mar. 2006. `http://www.reasoning.com/`.

[74] International Organization for Standardization. *Ada 95 Reference Manual. The Language. The Standard Libraries*, Jan. 1995. ANSI/ISO/IEC-8652:1995.

[75] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 329–340. ACM, Oct. 1998.

[76] K. Jensen and N. Wirth. *PASCAL User Manual and Report (third edition)*. Springer-Verlag, New York, NY, 1985. Revised to the ISO Standard by Andrew B. Mickel and James F. Miner.

[77] C. B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[78] I. T. Kassios. *A Theory of Object-Oriented Refinement*. PhD thesis, University of Toronto, 2006. To appear.

[79] Kestrel Development Corporation. *Specware System and documentation*, 2004. `http://www.specware.org/`.

[80] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Berlin, June 2001.

[81] J. Krone, W. F. Ogden, and M. Sitaraman. Modular verification of performance constraints. In *ACM OOPSLA Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pages 60–67, 2001.

[82] L. Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32(1):32–45, Jan. 1989.

[83] L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.

[84] B. W. Lampson, J. L. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. Technical Report CSL-81-12, Xerox Palo Alto Research Centers, Oct. 1981. Also *SIGPLAN Notices*, 12(2), February, 1977.

[85] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21:92–100, 2004.

[86] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, Mar. 2005.

[87] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, June 2004.

[88] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In J. Palsberg, editor, *Static Analysis, 7th International Symposium, SAS 2000, Proceedings*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer, 2000.

[89] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986.

[90] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, Aug. 1977.

[91] B. Liskov and W. Weihl. Specifications of distributed programs. *Distributed Computing*, 1:102–118, 1986.

[92] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. *Acta Informatica*, 10(1):1–26, 1978.

[93] R. L. London, M. Shaw, and W. A. Wulf. Abstraction and verification in Alphard: a symbol table example. Technical report, Information Sciences Institute, USC, Dec. 1976.

[94] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, NY, 1992.

[95] Matisse Partners. Matisse: Methodologies and technologies for industrial strength systems engineering. `http://www.matisse.qinetiq.com/`, 2003.

[96] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

[97] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *Proceedings of the 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 06)*, volume 41, 1 of *ACM SIGPLAN Notices*, pages 346–358, New York, Jan. 2006. ACM Press.

[98] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[99] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[100] C. Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[101] C. Morgan and T. Vickers, editors. *On the refinement calculus*. Formal approaches of computing and information technology series. Springer-Verlag, New York, NY, 1994.

[102] J. B. Morris. Programming by successive refinement of data abstractions. *Software—Practice & Experience*, 10(4):249–263, Apr. 1980.

[103] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Trans. Prog. Lang. Syst.*, 21(3):527–568, May 1999.

[104] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[105] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, Feb. 2003.

[106] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. Technical Report 424, ETH Zurich, Mar. 2005.

[107] A. Narayanan and G. Karsai. Towards verifying model transformations. In R. Bruni and D. Varró, editors, *5th International Workshop on Graph Transformation and Visual Modeling Techniques, Vienna*, pages 185–194, Apr 2006.

[108] G. C. Necula. Proof-carrying code. In *Conference Record of POPL 97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*, pages 106–119, New York, NY, Jan. 1997. ACM.

[109] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[110] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*. Elsevier, July 2001.

[111] T. Nipkow, L. Paulson, and M. Menzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[112] T. Nipkow, D. von Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer

and R. Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[113] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.

[114] B. Nordström and K. Peterson. Types and specifications. In R. E. A. Mason, editor, *Information Processing 83*, pages 915–920. Elsevier Science Publishers B.V. (North-Holland), Sept. 1983. Proceedings of the IFIP 9th World Computer Congress, Paris, France.

[115] B. Nordström, K. Peterson, and J. M. Smith. *Programming in Martin-Lof's Type Theory*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, New York, NY, 1990.

[116] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct. 1994.

[117] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

[118] Rodin Partners. Rodin: Rigorous open development environment for complex systems. `http://rodin.cs.ncl.ac.uk/`, 2006.

[119] E. Rodríguez, M. B. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In A. P. Black, editor, *ECOOP 2005 — Object-Oriented Programming 19th European Conference, Glasgow, UK*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576. Springer-Verlag, Berlin, July 2005.

[120] G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[121] M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Syst.*, 24(3):217–298, May 2002.

[122] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., Boston, Mass., 1986.

[123] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[124] M. Shaw. *ALPHARD: Form and Content*. Springer-Verlag, New York, NY, 1981.

[125] T. Sheard. Languages of the future. In D. Schmidt, editor, *OOPSLA '04: Proceedings of the 19th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 39(11) of *ACM SIGPLAN Notices*, New York, NY, Oct. 2004. ACM.

[126] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy. Performance specifications of software components. In *Proceedings of the 2001 Symposium on Software Reusability (SSR-01)*, volume 26, 3 of *SSR Record*, pages 3–10, New York, May 18–20 2001. ACM Press.

[127] M. Sitaraman, T. J. Long, B. W. Weide, E. J. Harner, and L. Wang. A formal approach to component-based software engineering: Education and evaluation. In *Twenty Third International Conference on Software Engineering*, pages 601–609. IEEE, 2001.

[128] M. Sitaraman, B. W. Weide, and W. F. Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering*, 23(3):157–170, Mar. 1997.

[129] D. R. Smith. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[130] D. R. Smith. Mechanizing the development of software. In M. Broy and R. Steinbrueggen, editors, *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, pages 251–292. IOS Press, Amsterdam, 1999.

[131] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, NY, second edition, 1992.

[132] S. Stepney, D. Cooper, and J. Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.

[133] M. E. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, editor, *12th Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[134] Stratego documentation. `http://tinyurl.com/nr2c5`, Mar. 2006.

[135] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[136] D. E. Turk, R. B. France, B. Rumpe, and G. Georg. Model-driven approaches to software development. In *OOIS Workshops*, pages 229–230, 2002.

[137] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer-Verlag, 2001.

[138] A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 5–19, New York, NY, June 2000. ACM Press.

[139] P. Voda. What can we gain by integrating a language processor with a theorem prover. unpublished; available from the author's web site, 2003.

[140] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley Longman, Reading, Mass., 1999.

[141] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[142] E. Westbrook, A. Stump, and I. Wehrman. A language-based approach to functionally correct imperative programming. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, 2005.

[143] J. M. Wing. Writing Larch interface language specifications. *ACM Trans. Prog. Lang. Syst.*, 9(1):1–24, Jan. 1987.

[144] J. M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, Sept. 1990.

[145] N. Wirth. The programming language pascal. *Acta Informatica*, 1(1):35–63, 1971.

[146] H. Xi. Facilitating program verification with dependent types. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, pages 72–81, 2003.

[147] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, Jan. 1997.