Proceedings of the Resolve 2007 Workshop:

Techniques and Tools for Verification and Instruction

Clemson, South Carolina, USA

June 11–13, 2007

Edited by Jason O. Hallstrom
School of Computing
Clemson University
Clemson, SC 29634-0974

Available as Technical Report CU-DSRG-06-07-00
(`http://dsrg.cs.clemson.edu/`)

Available as Technical Report RSRG-07-01
(`http://www.cs.clemson.edu/~resolve/`)

Table of Contents

Preface

The goal of the Resolve 2007 Workshop is to bring together researchers and educators interested in formal approaches to software engineering. The emphasis of the workshop is on techniques and tools for software verification, and computer science pedagogy, with an emphasis on tool-assisted collaborative approaches.

The workshop serves as a forum for participants to present and discuss recent advances, trends, and concerns in these areas, as well as to formulate a common understanding of emerging research issues and possible solution paths. The topics of interest solicited from participants include:

- Verifying compiler technology
- Specification and verification of performance properties
- Modular approaches to detecting component interface violations
- Trade-offs among testing, formal verification, and model checking
- Combining concurrency-oriented and model-based behavioral specification approaches
- Formal characterization of user interfaces
- Formal modelling of file system behavior
- Formal characterization of mathematical and program types
- Formal semantics and proofs of correctness
- Resolve language and implementation issues
- Software engineering environments and tools
- Component-based software
- Client-view-first pedagogy
- Industrial insights for Resolve research
- Integrating analytical reasoning principles in computer science education
- Collaborative instruction in computer science education

The workshop is chaired by Jason O. Hallstrom and Murali Sitaraman (Clemson University). The education panel is supported through a grant from the National Science Foundation (DUE-0633506). Local arrangements are provided courtesy of the Clemson University School of Computing.

We would like to thank education panelists Stephen H. Edwards, Peter B. Henderson, Joseph E. Hollingsworth, and Richard Pak for their valuable contributions. We would also like to thank the anonymous reviewers for their careful review of student submissions. Finally, special thanks are due to Andrew R. Dalton for his assistance typesetting the proceedings.

The Resolve Software Verification Vision

William F. Ogden, Joseph E. Hollingsworth, Joan Krone, Murali Sitaraman, and Bruce W. Weide

THE GRAND VISION

THE overall Resolve vision is that of a future in which no production software is considered properly engineered unless it has been fully specified, and fully verified as satisfying these specifications.

This wouldn't mean that all such software is absolutely correct because software may not have been properly specified to meet the objectives of larger systems in which it is embedded. At the limits, larger systems are real world systems, so the analysis of requirements and subsequent specification of their software will almost certainly require more complex techniques than those for specifying and verifying correctness within the relatively pristine world of imperative programming. However, complete specification and verification of software would imply a clear separation of concerns. Questions about the correctness of systems in which software is embedded could be fully addressed by looking only at the specifications for that software, with absolutely no consideration of coding details, and questions about the correctness of software could be fully addressed with absolutely no consideration of the systems into which it is to be embedded.

ELABORATING THE VISION

Development of verifiable software ultimately requires an *integrated language* within which it is possible to write both predicates that record external specifications and code that is appropriately annotated with embedded mathematical assertions. Only then can software warrant the full confidence of its users that it meets its specifications and thereby guarantee its integrity. If systems that don't share a common design are combined to write and verify software, the slightest of inconsistencies in their semantics could easily vitiate apparent correctness results. Consequently, the entire language for developing software processing system must be considered as an integrated whole. In particular, software's specifications should be viewed as an essential part of the software, and not as an add-on sideshow that might or might not describe the actual code.

An absolutely crucial consideration for the verification vision is *scalability*. Many approaches that may work for small examples cannot be scaled up to industrial size software, and this has many ramifications. A primary technique for achieving intellectual manageability of large systems is to follow the divide-and-conquer approach, so good modularization mechanisms are central to the scalability of software verification. But modularization can be effective only if the language enforces a strong separation of concerns so that problems solved in one module remain solved independent of what happens in other modules. Such decoupling will allow software developers to work independently on their assigned modules and will support the scaling up of verification by allowing modules to be specified and verified individually as they are completed.

Language features that allow uncontrolled aliasing, in particular, don't adequately support this separation of concerns. A programming language for verifiable software must be designed to support modularity not just in the usual syntactic sense of separate compilation but also in this stronger semantic sense. It must be clean in the sense of preventing aliasing and other strong coupling deficiencies.

One consequence of the scalability requirements is that most current programming languages are not suitable as a basis for verifiable software, since they don't adequately support separation of concerns: They allow aliasing and other strong coupling deficiencies, and they lack syntactic slots for specifications and code annotations. So, if well-engineered software is to be verified software, then legacy software and legacy languages are best viewed as being merely useful prototypes.

A further consideration to specify behavior formally is the need to incorporate a language for describing mathematical theories into any adequate verification system. If a verification system prescribes any fixed set of mathematical theories for writing the specifications of all software, then the size and complexity of specifications for code must increase in proportion to the size and complexity of that code, and intellectual manageability goes out the window. This problem has been attacked at the intuitive level by introducing into programming languages the notion of objects, which can be composed into ever larger objects. However, if we formalize such objects by modeling them by simply combining models

This research is funded in part by a grant from the National Science Foundation (CCR-0113181).

William F. Ogden and Bruce W. Weide are with the Department of Computer Science and Engineering at The Ohio State University (ogden@cse.ohio-state.edu, weide@cse.ohio-state.edu).

Joseph E. Hollingsworth is with the Department of Computer Science at Indiana University Southeast (jholly@ius.edu).

Joan Krone is with the Department of Mathematics and Computer Science at Denison University (krone@denison.edu).

Murali Sitaraman is with the School of Computing at Clemson University (murali@cs.clemson.edu).

of their constituent parts, then formal specifications lose the primary advantage that object formation was intended to provide. The only way to control this combinatorial explosion of specifications is to include mechanisms that allow introduction of progressively more sophisticated mathematical models in terms of which the behavior of large and complex programming objects can be specified in concise and simple ways. In short, the specificational machinery must not only be clean, but it must be rich in modeling possibilities as well.

For verified software to be trustworthy, proofs of correctness must be *credible*. Foolproof mechanical verification must rest upon an industrial-strength mathematical basis, rather than the traditional informal basis [1], which is inherently vulnerable to accepting the occasional incorrect result. Verifying programs that involve large objects may well require invoking difficult mathematical results about the sophisticated models that have been used to describe such objects. So it will be necessary to to develop new mathematical models for large objects in addition to redeveloping and formalizing the ordinary mathematics used for programming in the small. The credibility of software verification will then rest upon the demonstrated correctness of those mathematical notions and results.

A central idea in effective and credible verification is to *reuse* libraries of components. As the evolution of other engineering fields illustrates, an efficient and predictable methodology for creating reliable, upgradeable systems depends critically upon carefully designing and extensively employing reusable components. The development of such reusable components for software engineering has a symbiotic relation with the development of software verification technology in that it is vital for widely deployed components to be accurately specified and fully verified. From a verification perspective, reusing software components amortizes the cost of specification and verification over a much larger base of applications.

Just as the reliable construction of software depends heavily upon libraries of components, the construction of software specifications depends heavily upon libraries of theories. The quality of the theories/components in these libraries is indicated by the availability both of simple constituents that are frequently used and of more sophisticated constituents that are tricky to develop. Parallel organizational principles apply to mathematical libraries and to component libraries. A client of a mathematical theory needs to see an interface containing only a summary of the definitions and theorems, just as a client of a component needs to see a module containing only the specifications of its objects and operations. Beyond their existence, the implementation details for the constituents of a component are irrelevant to a client, and to promote separation of concerns, they should be relegated to separate modules. Similarly, the proof details for the theorems of a theory should be relegated to a separate module as well.

To verify software is correct and is based on sound libraries, a *mechanical verifier* must be available to check that mathematical theorems follow from their proofs and that code satisfies its specification. A language for developing verifiable software must be simultaneously cognizant of the limits of mechanization and of the needs of language users. For example, both because mathematicians generally will be needed to develop the non-trivial proofs upon which some software relies and because traditionally educated software engineers will be required to read and write specifications, it is essential that the software verification system present theories and specifications in standard mathematical notation. Also, because general automatic theorem proving is both theoretically and (currently) practically undoable, the software verification system should provide theorem provers with a subsystem that checks the correctness of putative proofs presented in a natural deduction notation that is familiar to mathematicians but at the same time in a syntactic form that permits automated checking.

Making the verification of production software routine depends on a taxonomic thesis about how software engineers create software that they "know" is correct. The thesis is that most of such code is straightforward and it's plain to see that it is correct. The remaining not-so-obvious parts are separable from the rest, and certainty of the correctness of each such part is developed through a serious individual process of abstract reasoning. If this thesis is correct, then a software verification system can achieve its objectives using two qualitatively different subsystems. The first addresses the not-so-obvious and is the general mathematics subsystem discussed above that handles theory and proof modules. The second is a code justification checker that examines the specifications embedded in code to determine whether they are "obviously" correct, based on the specifications and annotations in the code, and the theorems and definitions provided by the theories in use.

Software verification is a grand challenge [2]. Realizing the challenge demands much of language designers and implementers. It also comes with responsibilities for language users and their educators. Developers of a well-engineered piece of software should be competent and should be capable of supplying to the justification checker the internal code specifications, such as loop invariants, etc., that are the basis of their belief that the straightforward parts of their code are plainly correct. Moreover, they should also be responsible for making sure that the results of the serious reasoning that was needed for any non-trivial parts of their code are included as theorems and definitions in the available mathematical theories. Given these theorems and definitions and a bit more code annotation, the "tricky" parts of the software become straightforward for the justification checker. One effect, then, is to move all the difficult reasoning over to the theorem proving activity, where it is likely to be reusable in other software development. This leaves the code justification checker with a task that's only a bit harder than sophisticated compiling and makes it a reasonable constituent of a verifying compiler.

Adequate specification of software requires distinct mechanisms for describing its characteristics with respect to

functionality, termination, and performance. Describing functionality in general involves employing the rich spectrum of mathematical theories discussed above, while establishing termination involves the use of progress metrics, which map from the spaces supplied by these theories to the space of ordinals. Performance actually includes two software characteristics: durations for procedures and displacements for objects and operations. Specifying these involves mappings from object spaces to, respectively, real numbers and integers. These various mechanisms somewhat complicate the verification rules for software, but don't make verification materially more complex than it is for functionality alone.

REFERENCES

1. Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM 22*, 5 (1979), 271-280.
2. Tony Hoare, The verifying compiler: a grand challenge for computing research. *JACM 50*, 1 (2003), 63-69.

Resolve 2007: Current State of Verification

J. Krone, W. F. Ogden

***Abstract*—At the Resolve '06 Workshop, a new way of thinking about verification was introduced. There we identified two main tasks that must be performed in order to achieve verification of Resolve programs. One is proof checking: automated checking of proofs submitted as constituents for libraries of mathematical components. The other is code justification, which involves the automated application of proof rules to generate and process mathematical clauses from the code to see if there is sufficient mathematical support to establish the correctness of the code. Here we present a report on progress made by the group toward the goal of verification.**

***Index Terms*—software correctness, proof rules, semantics, verification**

I. INTRODUCTION

At the Resolve '06 workshop [1] we discussed Resolve in the context of the Verifying Compiler Grand Challenge and concluded that our approach to software, as opposed to other existing approaches, has the potential to succeed. We advanced the thesis that there is no prospect for taking current software and retrofitting formal specifications to it with the expectation of verifying its correctness. We observed that Java and C++, the most widely used contemporary programming languages, include features that make verification at best difficult and generally impossible.

Twenty five years of addressing the software verification challenge has confirmed our belief that any successful approach must include automated proof checking, automated code justification, and most critically, a language with syntactic support not only for executable constructs, but for mathematical specifications and theories as well.

Believing that Resolve provides the best approach to verification, we continued to develop a prototype system for Resolve, including a compiler, a proof checker, and a code justifier.

Experience has shown us that, while it is tempting to select a single problem and work on it to completion, the reality is that most problems here interconnect with other problems, and so we must simultaneously confront several intertwined problems.

Manuscript sent May 22, 2007.

J. Krone is the Benjamin Barney Professor of Computer Science at Denison University in Granville, Ohio, 43023 (krone@denison.edu).

W.F. Ogden is Professor Emeritus at The Ohio State University, Columbus, Ohio 43210 (ogden@CSE.Ohio-State.edu).

For example, only when considering use cases involving the three most general constituents of Resolve did it become clear that we need type checking rules for mathematical constituents that are different from the type checking rules for executable code. More subtle discoveries concerning when to place mathematical definitions in a realization as opposed to a mathematical theory unit, for example, emerged only when we examined use cases.

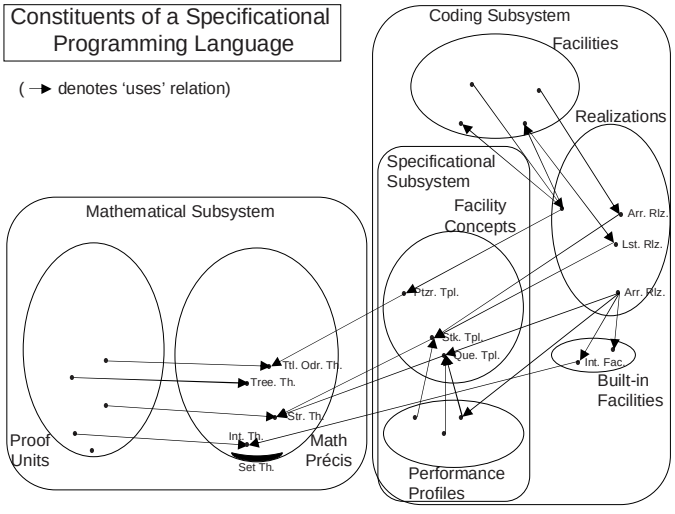
Here we identify which particular aspects of the Resolve project have been addressed during the past year, noting what progress has been made on those aspects. In some cases there have been major steps made, while others have seen more modest progress.

II. PROGRESS SINCE THE LAST WORKSHOP

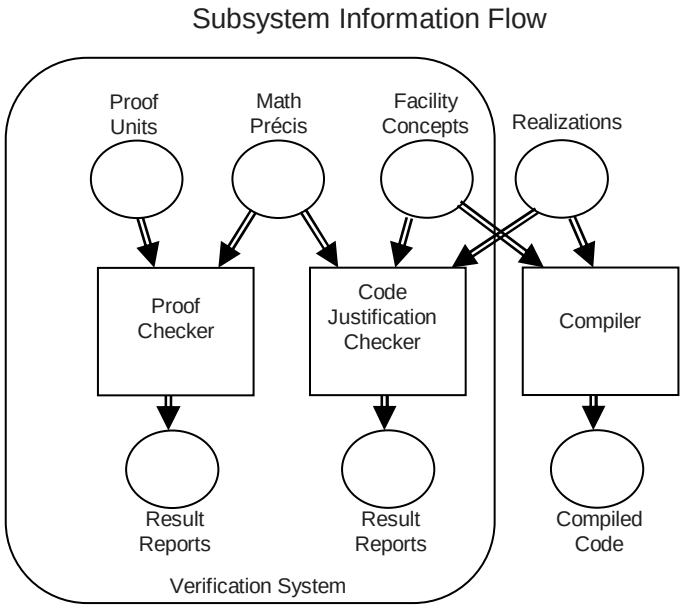
We will leave it for those who have worked directly on the compiler, the proof checker, and the code verifier to present demos of their works in progress. Our contributions have been to provide necessary support for those components.

A. Understanding of the Overall Verification System

The first diagram below shows the interconnections among the constituents in the Resolve verification system, indicating which constituents use which other constituents.



The second diagram indicates the flow of information among the verification components and the realization components of Resolve. Here the relationship among the three major Resolve components currently under construction are shown.



B. *Proof Rules*

Certainly one critical need when automating code justification is a complete set of sound, automatable proof rules. By examining some use cases we found that, although our program proof rules were sound, they had, in some cases, forms that did not lend themselves well to efficient automation. In response to our discoveries we updated our collection of proof rules, leading us to rules that omit some redundancies, and thereby making the rules more efficient when automated.

For example, our **while** rule had been written as a three hypothesis rule, primarily to make the rule understandable to students. However, by changing it to a two hypothesis rule and expressing one of the hypotheses as a conditional, we made it possible for programmers to employ less cumbersome loop invariants than would have been needed in many cases for the old form.

It was also important to update the operation rules to address all of the parameter modes that have been defined during the past few years.

Additionally, when doing one of our use cases, we discovered that our old rules were not adequate, because they did not automatically include the constraints on the types of each parameter, something that is necessary, but that did not show up until we attempted to automate the proof process.

C. *A New Parameter Mode*

While updating the operation rules, we noticed that some parameters are specified to change by functional postconditions given strictly in terms of the old values of variables. By introducing a new parameter mode, which we call "reassigns," we distinguish these parameters from others such as updates and alters, since those changes may be relational and the clauses generated for those changes may need universal quantification.

This discovery was made in the context of trying to simplify what the code justifier produces when applying proof rules. We noted that for some parameter modes, certain clauses can be promoted to the beginning of the generated hypotheses, while others cannot. For the reassign, evaluates, restores, and preserves parameters, such promotion can be done, while for the other modes, it cannot.

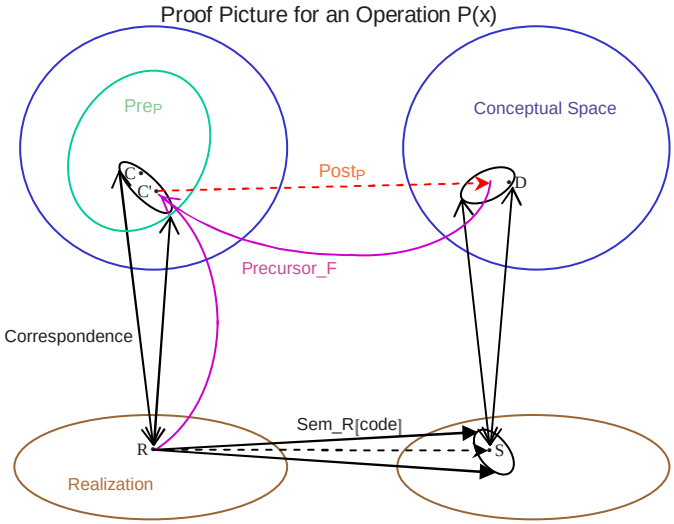
D. *Prototypes for Concepts and Realizatoins*

Part of the task of updating proof rules for **Concepts** and **Realizations**, was the clarification of the syntax of both constructs. Care was then taken to include both local and global variables, operations with various kinds of parameters, and the introduction of syntax that allows for both functional and relational operations. Respecting the principle of clean semantics, we introduced a construct "alludes to" to make explicit the use of globals that do not appear on the parameter list.

E. *Precursor Function for Relational Correspondences*

Another update was required when we realized that our proof rule for realizations with relational correspondences involved an existential quantifier. Our general thesis is that code justification will be much simpler if all existential quantifiers are removed in favor of Skolem functions. Here that meant adding a syntactic slot for a user supplied **Precursor_for** function.

In the accompanying diagram, for any conceptual value D that might correspond to a result S of applying the realization code to state R, the programmer must identify a particular initial conceptual value C' that corresponds to R and could result in D.



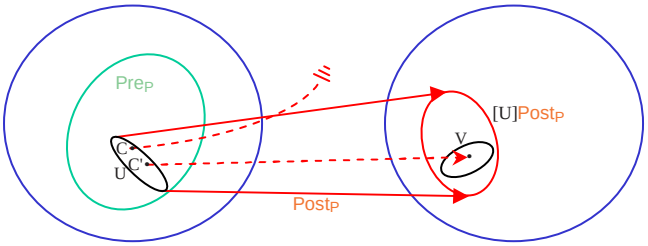
F. *Terminability Condition for Relational Specifications*

A further observation about the relational correspondence situation is that the notion of termination becomes trickier to characterize for operations with relational post conditions. Note that while a realization may terminate for all allowable input states, in the corresponding conceptual space, there may be some states C that aren't specificationally permitted to

advance. So the obvious characterization of a terminating I/O relation as being a total relation is clearly too restrictive.

The notion of compatible collections of uncertainty regions turns out to be the appropriate generalization to capture the proper notion of a terminating I/O relation. So in the denotational semantics for Resolve, when we want to filter out I/O relations for conceptual operations that require termination, we use this compatible collections notion instead of totality.

Implication for Termination Semantics



∀ U: Uncertainty_Rgn, **if** Pre$_P$[U] = {True},

then ∃ V: Uncertainty_Rgn ∋ V ⊆ [U]Post$_P$.

Not the same as:

∀ C: State, **if** Pre$_P$(C), **then** ∃ D: State ∋ C Post$_P$ D.

An additional complication of allowing the I/O relations for operations not to be total is that guaranteeing the existence of minimal fixed points for relational semantics requires the use of total relations. The patch for this difficulty is to introduce another magical semantic state, SI, which denotes a state that is Specificationally Impossible to reach. The SI state results when the code preceding an operation invocation, P(a, b), could produce states that go nowhere under P's I/O invocation.

G. SpecificationsSimplified Communal Concept Specifications

Current specifications for communal facilities involve talking about all instances of objects provided by the facilities, but this approach doesn't accord well with verifying client code. This approach was developed long before we explicitly elaborated the Clean Semantics objective for Resolve, and it was intended to preserve specificational completeness in the face of the possibility that code for an operation might side-effect facility objects that were not syntactically targeted by the operation. Of course, we now recognize that such side effects must be outlawed if Resolve is to support a verifying compiler.

Accordingly, while specificational mechanisms for talking about all instances of a programming type (e.g., Stack.**Last_Specimen_Num** and Stack.**Denoted_by**) must be retained for verification of the "nothing else changes" frame property in realizations, such mechanisms can no longer be used in the conceptual descriptions of facilities. Instead, suitable facility global variables must be introduced. When done appropriately, this seems to lead to much simpler specifications for most current communal concepts.

H. Forms Allowed in Mathematical Proofs

In order to automate the process of checking proofs that have been submitted to serve as mathematical support for code, we have prescribed particular syntax for writing those proofs. Moreover, we have identified appropriate logic principles, such as substitution and modus ponens, to be built into the justifier.

I. Facility Declarations within Objects

It has long been known that Resolve must allow declarations of local facilities inside realizations. However, we recently found a situation for which there is an advantage to declaring a facility within an object. This advantage occurs when creating a communal module, since each individual object within the communal structure may have its own realization object description distinct from other objects in the same communal structure.

J. Depository Metaphor for Partial Maps

The Partial Map and the Almost_Constant_Function concepts were introduced to serve as a basis for searching. A variety of realizations have been written for them, and more recently, the issue of what mathematical definitions and theorems would be needed to prove that the realizations are correct has been addressed. In addition, profiles for these concepts have been written to specify performance requirements.

However, one important question that arises when putting all of these ideas together is whether there might be a better metaphor and nomenclature for these concepts. Of course, associated with the metaphor question is the matter of whether the existing modules have the right sets of operations.

There is now a newly developed concept using the metaphor of a Depository, along with the appropriate nomenclature for the objects and operations.

III. What Remains?

Having examined a summary of progress made during the past year, we turn now to those items that we call "frontier activities," activities that are now at a point where sufficient background material exists to enable them to move forward.

A. A Resolve Vision Statement

The draft of a vision statement was begun a year ago, laying out an overview of the Resolve philosophy and putting the project in perspective of the Verifying Compiler Grand Challenge. The progress of the past year may serve as motivation to complete this document.

B. Exploration Tree Realization for the Depository

Now that the specifications for a Depository Concept have been written, it is reasonable to take up the challenge of providing an efficient realization for it, the Exploration Tree being a likely candidate.

C. Rules for Subtyping

Although some rules for mathematical typing have been decided, there are still unanswered questions about how much explicit typing is necessary in order to determine what operations may be performed on what objects.

D. Compiler Development

Significant progress toward a working compiler for Resolve has been achieved and momentum exists for the work to continue.

E. Resolve Books

Eventually, we will need at least five books for the RESOLVE project:

- Rationale for Features of Resolve for Experts in Software Engineering and Programming Languages (Vol. I, II, ⋯)
- Introduction to Resolve for CS 1 & 2
- Data Structures in Resolve
- Programming Language Theory based on Resolve
- Discrete Math in Resolve

The first, and most critical, of these books is partially written and progress on it continues. Once it has been completed, the others can be done in parallel.

F. Additional Performance Profiles

Although we have completed a number of profiles, all related to the Spanning Forest problem, we need to continue developing more profiles so that we can convert existing code which still includes performance information in realizations to profile form. It is also necessary to write proof rules for profiles and work out some examples that include performance.

G. CS Encyclopedia Articles on Specification and Verification

At least two articles are in draft form.

H. Experiments with Isabelle

Having been asked for a sorting example to give to Isabelle, we prepared not only the clauses generated for an insert sort based on the List_Position concept, but also the context in which those clauses would need to be processed. We did the proof manually and sent the clauses and context to a person who plans to submit the example to Isabelle and then to let us know what happens.

Although Isabelle has performed adequately on examples involving integers, this sorting example will serve as a way to see whether Isabelle can handle more general objects. In particular, it will help us find out just how complicated it may be to put in the necessary mathematical content for Isabelle to use in its attempt at finding a proof.

I. Caching Optimization and Scratchpad Memories

Memory management has been a topic of interest to the Resolve community for many years, but a topic for which we have no immediate answers for what concepts are needed and how to specify them. We have looked at communal modules as a possible attempt at addressing the problem of how to specify shared storage. Further study in this area must figure out what, if any, support communal modules may provide.

IV. Summary

Without verification, Resolve is just another language. The past year has been a particularly active one with respect to making progress toward the goal of Resolve verification. If we were to be optimistic, we might conjecture that a year from now, there will be the prototype of a working compiler with a proof checker and a code justifier.

References

[1] J. Krone, W.F. Ogden, "Software Verification is Not Dead, But It Needs a New Way to Express Mathematics," Resolve'06, March, 2006.

Mechanical Proof Checking and Its Role in Establishing Software Correctness

Kimberly E. Roche

***Abstract*—A complete automated software verification system will need to provide the functionality to identify type and other semantic errors within proof modules. This paper describes the creation of a mathematical type checker and proof checker for the Resolve verifying compiler. It outlines the operation of those tools and the state of their development at the time of this writing.**

***Index Terms*—Software tools, software verification and validation, theorem proving**

I. Introduction

ADEQUATE systems for the mechanical verification of software components must rely on theorems (both those already defined in mathematical theories and those generated by the user of the system) to demonstrate the correctness of code. An essential part of the reliability of such a system will be a mechanical proof checker, capable of automatically verifying the correctness of proofs of those theorems.

Such a checker would allow a user to establish and include new theorems in mathematical units, verified by proofs, and then allow those theorems to be used in checking the correctness of implementation code.

As an illustration of the necessity of including a proof checker in a fully-developed verification system, we examine the following abbreviated piece of code and a proof obligation that might be associated with it:

```
Var I, K, L, M: Integer;
I := Sum(K, L);
I := Sum(I, M);
Confirm I = K + (L + M);
```

To prove the correctness of this code, after applying proof rules and making appropriate substitutions, a verifying compiler must prove the equivalence of the statements "(K + L) + M" and "K + (L + M)". This task would be quite straight-forward assuming the authors of the mathematics libraries had included a theorem describing the associativity of the operator "+" for Integers. The verification would not be "complete", however, unless that theorem of associativity could itself be proven.

To date, proof checking approaches have proceeded in two general directions, resulting in the creation of tools we will call "proof assistants" and those we will refer to as "proof checkers". So-called proof assistants involve the interactive development of a proof, typically requiring that the user "guide" the process of proof creation. These tools may be able to participate in the development of the proof in a limited capacity by making simple inferences. On the other hand, proof checkers aim to be largely autonomous in their operation. Supplied with a proof and a library of mathematical definitions, a proof checker attempts to verify the proof, developing any further proof obligations that may be necessary. We intend that the Resolve proof checker indeed be a "proof checker" in this sense but with one important distinction.

Some existing mechanical proof checkers rely on static libraries of mathematical types, definitions, axioms, theorems, and lemmas. An update to these libraries requires an update to the proof checker itself. The Resolve proof checker intends to be as dynamic as possible. With the exception of axiomatic set theory and a portion of Boolean theory, both of which are built-in but extensible, all other mathematical libraries are intended to be created, modified, and updated by the user. For illustration of the approach, basic versions of set theory, natural number theory, integer theory, real number theory, and string theory have been created as a foundation from which further development can be made. As mathematicians are intended to be the creators of these proofs and not programmers, the syntax of Resolve-language proofs has been designed to adhere closely to the style of formal mathematical proofs. An excerpt from an example proof, as well as an explanation of proof syntax are provided in the proof checking section of this paper.

The remainder of this paper will outline major points in the design and implementation of the Resolve proof checker. Since many errors in mathematical typing can be detected using a traditional type checker, we will first examine the operation of the Resolve mathematical type checker in Section II. In Section III, the proof rules on which the Resolve proof checker operates will be explained and finally, a detailed description of the operation of the proof checker itself as well as an account of prior research will be given.

This research is funded in part by a grant from the National Science Foundation (CCR-0113181) and a grant from the NASA SC Space grant Consortium.
Kimberly Roche is with the School of Computing at Clemson University (kroche@cs.clemson.edu).

II. The Resolve Type Checker

The mathematical type checker serves to identify any potential errors in the type consistency of Resolve mathematical expressions and in so doing, simplifies the work of the proof checker.

The compiler employs two independent type checking systems: one which performs traditional type checking on Resolve code and one which checks for type errors in Resolve mathematical units (libraries), proof modules, and specification modules, and annotations within implementations (e.g. loop invariants). Each module specifies any other modules that are visible to it and these referenced modules are compiled first.

The Resolve mathematical type checker possesses two features not present in the Resolve program type checker. Firstly, the mathematical type checker is able to resolve the mathematical types of not only specification variables but program variables. Resolve program types are defined as aggregations of one or more mathematical types. For example, a stack is modeled as a mathematical string. It is allowable in some mathematical contexts then, to name to a program variable when referring to that variable's mathematical type.

Next, the nature of mathematical types themselves demands that the type checker be able to handle what we will refer to as "sub types". In mathematics, it is often acceptable to use a natural number where, for example, an integer is expected. Consider the following assertion:

```
For all x: N, there exists y: Z, such that, x + y = 0;
```

In this case, if N and Z represent the set of all natural numbers and the set of all integers respectively and the expression should be allowed by the type checker. This essentially amounts to an implicit cast of the specification variable x to type Z. But how should the type checker know what is acceptable? In this case we expect the unit Integer_Theory to include a special "subtype theorem" that essentially identifies a natural subset of integers with N and shows that definitions, such as + on N correspond to + on Z.

Each expression is evaluated within its context to determine if it should be type checked by the program type checker or the mathematical type checker. The Resolve type checker then performs a bottom-up walk of the syntax tree of that expression in order to verify its type consistency. Sub types are explicitly declared in theory modules and that information is stored in the symbol table. For a given module, the symbol table holds a list of mathematical types and the membership of those types within other types. If a match cannot be made between two types and if it is appropriate in the context of the statement to implicitly cast one or both of those types to some other type, the type checker attempts to match the types again using any substitutions that can be made.

In the example above, the type checker would try to establish the existence of a definition "+" parameterized by an N value and a Z value and fail. It would then cast the first argument (of type N) to type Z and perform the search again, this time, successfully.

It should be mentioned that a sub type declaration introduces a unique proof obligation. Each definition to which the sub type is a parameter must be mapped to an equivalent definition parameterized by the super type. The current version of the Resolve type checker allows for sub type declarations to be made and for the implicit casting described above but is not yet able to verify the correctness of sub type declarations.

Some axioms are built-in to the Resolve type checker, including basic set theory definitions of the empty set, power set, and concept of membership. As they are so fundamental, these axioms are essentially "hard-coded" into the type checker. All other mathematical theories (including any extensions to set theory and Boolean theory) must be provided by the user and are loaded into the symbol table dynamically.

III. Proof Rules

The current version of the Resolve proof checker allows the author of a proof to use 16 different proof rules (or justifications). These rules and their mathematical elaborations were introduced by Bill Ogden. At present, these justifications seem to be sufficient to describe any single step in a formal proof. Each justification requires up to two references, which may be theorems, definitions, lemmas, corollaries, suppositions, or previously verified lines in the same proof. The justifications requiring two references are as follows:

- **Modus ponens**
- **And rule**
- **Contradiction**
- **Alternative elimination**
- **Common conclusion**
-

The justifications requiring one reference are:

- **Equality**
  - *For ease of use, the keyword "equality" may be omitted from the justification.*
- **Reductio ad absurdam**
- **Universal generalization**
- **Existential generalization**
- **Or rule**
- **Conjunct elimination**
- **Quantifier distribution**
- **Definition ∃!**
- **Universal instantiation**
- **Existential instantiation**
-

Finally, the only rule requiring no references is:

- **Excluded middle**

These rules and any necessary references can be cited in certain expressions within the proof, providing *direction* to the proof checker when verifying a given proof line. The types of statements allowed in a Resolve-language proof will be explained in the next section.

IV. The Resolve Proof Checker

In order to illustrate the format of a Resolve-style proof, let us present the following excerpt of a proof of the associativity of the operator "+" over the set of natural numbers. The proof references the definition of "+", part of which has been included for reference.

```
Inductive Definition (a: N) + (b: N): N is
    (i) a + 0 = a;
    (ii) …

Proof of Theorem N4
    Goal for all k, m, n: N, k + (m + n) = (k +
    m) + n;
    Definition S1: Powerset(N) = {n: N, for all
    k, m: N, k + (m + n) = (k + m) + n};
    Goal S1 = N;
    Goal for all k, m: N, k + (m + 0) = (k + m)
    + 0;
    Supposition k, m: N;
        k + (m + 0) = k + m  by equality & (i) of
        def of +;
        k + m = (k + m) + 0  by equality & (i) of
        def of +;
    Deduction if k in N and m in N then k + (m +
    0) = (k + m) + 0;
    For all k, m: N, k + (m + 0) = (k + m) + 0
    by universal generalization;
    id1: 0 in S1  by equality & def of S1;
    …
```

The above proof demonstrates each of the four types of expressions allowed in a Resolve language proof:

- **Goal expressions.** Preceded by the keyword "goal", these are ignored by the proof checker and are intended for the benefit of the user alone. Goal expressions serve as comments within the proof.
- **Definitions.** These follow the syntax of standard Resolve definitions. Definitions made within the proof are only visible inside the proof.
- **Supposition/deduction expressions.** Preceded by the keywords "supposition" or "deduction", these consist of a supposition, deduction, and several intervening statements which use the supposition to state the deduction. The deduction must be in *if-condition-then* form where the supposition must be referenced as the condition of the deduction.
- **Justified expressions.** Each justified expression has two parts: the body, a Boolean-valued expression, and a justification statement. The justification specifies a proof rule and any references required by that rule. Together, the rule and its references must allow the proof checker to verify that the body of the expression is true. Additionally, any line in the proof may be labeled with a unique identifier and referenced in these justified expressions.

For a given statement within a proof, each previous line in the same proof is considered to be a part of the context of that statement. The proof checker begins with the left hand side of original relationship or relationships to be proven and follows the proof line by line, transforming the original statement as it travels through the proof. Each supposition, deduction, or justified statement is individually verified and then applied to the transformed expression if possible. At the end of a valid proof, the transformed expression should be represented in the original theorem to be proven.

For each justified line, the associated references (if any) must first be located and kept available until the line has been verified. A justified line may cite one of the rules described above and as many references as are necessary to support the verification of the line by that rule. Users are allowed to reference any visible definitions, theorems, axioms, lemma, suppositions, and any prior statements in the proof.

For illustration purposes, we will include a pair of examples. The first demonstrates the usage of the *equality* proof rule and the second utilizes the *and* proof rule.

```
Inductive Definition (a: N) + (b: N): N is
    (i) a + 0 = a;
    …

Proof of Theorem N4
…
    k + (m + 0) = k + m  by equality & (i) of
    def of +;
…
```

Expressions of equality or inequality, such as the line included above from a proof, represent transformations; one side of the equality, when transformed by the justification rule, should either be equivalent (in the case of an equality) or nonequivalent (in the case of an inequality) to the other side of the expression. Context determines how expressions are judged for equivalence; they may be evaluated on the basis of type and expression structure alone or type, structure, and identifier equivalence in the strictest case. The following example demonstrates the verification of a single statement in the above proof of the associativity of the operator "+" over natural numbers and includes some necessary context. Some omissions have been made in the definition and in the proof for the sake of brevity. Theorem N4 in this case is the associative property of the operator "+" over natural numbers.

The justification implies that the basis case of the above definition of "+" should be applicable to the justified line. The matching begins by attempting to find the left hand side of the definition's basis case, "a + 0", somewhere in the left hand side of the justified statement. "a + 0" will be matched with "m + 0" and, importantly, the variable identifier "a" will be mapped to the sub-expression "m" for the remainder of this matching attempt. Next, the proof checker isolates the sub-expression of the justified line transformed by the application of the rule, namely "m + 0", and checks to make sure that no other part of the expression has changed. In this case, the original "k + …" statement present in the left hand side of the expression should remain the same on the right hand side and indeed, it has. The proof checker then tries to match the right hand side of the basis case of the definition with the transformed portion of the expression on the right hand side of

the justified line (i.e. matching "a" in the definition with what appears in the place of "m + 0" on the right hand side of the justified line: "m"). Since "a" is mapped to "m" for this round of matching, it is satisfied only upon encountering an "m".

If the initial application of the rule to the justified line fails, the proof checker tries (in a breadth-wise manner) all other possible applications of the rule to the line. Matching on an inequality is performed similarly, although in that case we must make sure that there is *no* satisfying application of the rule to the justified line.

A subsequent line from the same proof illustrates the semantics of the "and" proof rule.

```
0 in S1 and for all n: S1, suc(n) in S1  by id1,
id2 & and rule;
```

The references provided here denote proof steps marked by the identifiers "id1" and "id2". In this case, the statement of 0's membership in the set S1 must be cited somewhere in the context. Once found, the proof checker must locate a previously verified statement matching the second clause of the "and" expression. Only if both the left hand and right hand sides of the "and" statement can be found will the line be verified.

V. Examples of Use

The following are some excerpts from a single Resolve proof, modified in such a way as to cause errors. The complete output of the compiler is given for each, as well as an informal explanation of the error.

Excerpt:

```
Supposition k, m: N;
  Goal for all k, m: N, k + (m + 0) = (k + m) + 0;
  k + (m + 0) = (k + m)  by equality & (i) of def
+;
  k + m = (k + m) + 0  by equality & (i) of def +;
Deduction if k is_in N and m is_in N then k + 0 =
k + (m + 0);
```

Error Message:

```
Construct record: Natural_Number_Theory_Proofs
Construct record: Boolean_Theory
Complete record: Boolean_Theory
Construct record: Set_Theory
Import file Boolean_Theory.mt has already been
successfully compiled.
Construct record: Natural_Number_Theory
Import file Boolean_Theory.mt has already been
successfully compiled.
Complete record: Natural_Number_Theory
Complete record: Set_Theory
Import file Natural_Number_Theory.mt has already
been successfully compiled.

Error: Natural_Number_Theory_Proofs.mt(47):
Could not verify deduction from supposition and
intervening statements.
        Deduction if k is_in N and m is_in N then
k + 0 = k + (m + 0);
        ^

Abort compile: Natural_Number_Theory_Proofs
```

This fails because the proof checker cannot trace the equality of expressions "k + 0" and "k + (m + 0)".

Excerpt:

```
k + (m + 0) = m + 0  by equality & (i) of def +;
```

Error Message:

```
Construct record: Natural_Number_Theory_Proofs
Construct record: Boolean_Theory
Complete record: Boolean_Theory
Construct record: Set_Theory
Import file Boolean_Theory.mt has already been
successfully compiled.
Construct record: Natural_Number_Theory
Import file Boolean_Theory.mt has already been
successfully compiled.
Complete record: Natural_Number_Theory
Complete record: Set_Theory
Import file Natural_Number_Theory.mt has already
been successfully compiled.

Error: Natural_Number_Theory_Proofs.mt(45):
Unable to verify the transformation from the
information provided.
            k + (m + 0) = m + 0  by equality & (i)
of def +;
                                ^

Abort compile: Natural_Number_Theory_Proofs
```

This fails because there is no application of the cited rule that will allow such a transformation.

Excerpt:

```
k + m = (m + k) + 0  by equality & (i) of def +;
```

Error Message:

```
Construct record: Natural_Number_Theory_Proofs
Construct record: Boolean_Theory
Complete record: Boolean_Theory
Construct record: Set_Theory
Import file Boolean_Theory.mt has already been
successfully compiled.
Construct record: Natural_Number_Theory
Import file Boolean_Theory.mt has already been
successfully compiled.
Complete record: Natural_Number_Theory
Complete record: Set_Theory
Import file Natural_Number_Theory.mt has already
been successfully compiled.

Error: Natural_Number_Theory_Proofs.mt(46):
Could not apply the rule to the proof expression.
            x1: k + m = (m + k) + 0  by equality &
(i) of def +;
                  ^

Abort compile: Natural_Number_Theory_Proofs
```

This fails because the proof checker expects the relation "k + m = (k + m) + 0". Because the commutative property has not been explicitly included in the justification for the transformation, the proof checker cannot yet *infer* the equivalence of "k + m" and "m + k".

VI. Related Work

Isabelle

Written in standard ML and using the specification language "Isar", Isabelle is equipped with a library of formally verified theories contained within a library called "Main". Users can create their own theories, proofs of which must be stored in separate ML files. Isabelle recognizes a standard array of functional programming types including Booleans, natural numbers, sets, lists, function types, and type variables and contains a built-in type checker. Proofs are written as "sequences of commands" to the proof checker and can be checked interactively. Isabelle is a generic environment and can be paired with several different systems of logic. Standard schemes include:

- Isabelle/HOL – higher-order logic
- Isabelle/HOLCF – higher-order logic augmented with domain theory
- Isabelle/FOL – first-order logic
- Isabelle/ZF – first-order logic extended by Zermelo-Fraenkel set theory

Alternatively, the user is permitted to add a new logic by making a number of additions to the original source as outlined by Isabelle's developers [1].

The primary distinction between Isabelle/* and the Resolve proof checker is that the Resolve proof specification language, unlike Isar, is design to adhere as closely as possible to the traditional syntax of mathematics.

Coq

The proof assistant Coq, developed by a team at INRIA-Rocquencourt, uses a unique specification language called "Gallina". Operating on a small logical kernel, Coq allows the user to interactively create proofs, mechanically checking as they go. Coq comes with a standard library of theories covering the properties of Booleans, natural numbers, integers, real numbers, lists, maps and more, which can be extended by the user. In addition, the user can define new sets and functions within his or her proof. Coq is capable of participating autonomously in the proof creation process, albeit in a limited fashion. It can search proofs, make limited first-order logic decisions, and simplify expressions, among other things but is designed to rely heavily on user intervention [2].

PVS

Based on typed higher-order logic, PVS is labeled by its creators as being more powerful than a traditional proof assistant but not quite as capable as a theorem prover. PVS uses libraries containing definitions and lemmas to autonomously evaluate arithmetic and statements of equality within proofs with the aid of the SMT solver *Yices*. Additionally, it allows the user can modify these libraries while the proof is being written. While PVS exhibits a high degree of automation in comparison with other proof checkers and proof assistants, the proofs written in PVS, the developers admit, are not easily human-readable. PVS contains a type checker and proof checker that work in tandem. During the type checking phase, the proof checker may be called upon to address some proof obligations generated by ambiguous type conditions. The standard PVS package includes specification libraries, called "theories", which are built-in but which can be extended by the user. PVS recognizes base types which include some traditional numerical types and Booleans and allows the user to create new base types and sub-types. These types may then be combined with "type constructors" – functions, records, tuples, sets, enumerations, and more complex ADTs – to create new types [3], [4], [5].

In contrast to PVS proofs, Resolve-style proofs are designed to be immediately readable to anyone familiar with formal mathematical proofs.

Nuprl

Another ML proof assistant, created at Cornell University, it allows the user to interactively verify proofs. From an initial goal, Nuprl tracks a set of sub-goals which are refined using a library of tactics. Nuprl recognizes integers, function types, and sets, among others, which can be extended by means of Cartesian Products (tuples), unions, and lists. Unlike the Resolve proof checker, however, Nuprl assumes the type-correctness of the statements it evaluates and performs no type checking on its own. Built-in libraries include definitions over Booleans, sets, lists, array, and basic arithmetic operators and additional libraries can be added by the user [6], [7].

By contrast the designers of the Resolve proof checker have avoided (to the extent possible) building mathematical theories themselves into the proof checker. The most basic elements of set theory and Boolean theory are the only mathematical theories native to the proof checker.

References

[1] Nipkow, T., L. C. Paulson, and M. Wenzel, *A Proof Assistant for Higher-Order Logic.* New York: Springer-Verlag, 2002, pt. I1-2.

[2] Huet, G., G. Kahn, and C. Paulin-Mohring, *The Coq Proof Assistant: A Tutorial.* INRIA, 2004, pp. 3-18; 45-47.

[3] Shankar, N., S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, *PVS Prover Guide: Version 2.4.* Menlo Park, CA: SRI International, 2001, pp. 1-24; 103-110 .

[4] Shankar, N., S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, *PVS Language Referencee: Version 2.4.* Menlo Park, CA: SRI International, 2001.

[5] Dutertre, B. and L. de Moura, "The Yices SMT Solver," August 2006, http://yices.csl.sri.com/documentation.shtml.

[6] PRL Project, "The Nuprl Book," September 1995, http://www.cs.cornell.edu/Info/Projects/NuPrl/book/doc.html.

[7] PRL Project, "Nuprl Basics – Nuprl Primitives," September 2003, http://www.cs.cornell.edu/Info/People/sfa/Nuprl/NuprlPrimitives/.

When an Assertive Program Is Wrong

Wayne D. Heym

***Abstract*—When verifying compilers come into use, the usual case will be that the program is found to be incorrect. Hence, the requirements for such compilers should emphasize strong support for users in this common case.**

***Index Terms*—requirements, verification condition generation, verifying compiler**

I. Position

Prominent among the requirements for verifying compilers and verification condition (VC) generators should be that good hints be provided the user about what is wrong (in the case when something is wrong, which is the usual case). Of course, users should expect reasonable hints when, considered in isolation, either implementation code or specification assertions are syntactically incorrect. Going further, the position here is that the user should also be provided with reasonable hints when the assertive program is wrong. That is to say, when the program cannot be proven correct with respect to its specification, the compiler should give the user some strong hints as to what has gone wrong.

One such hint would be to point out which particular obligation cannot be discharged. Is it an implemented operation's post-condition? Is it the precondition for a called operation? Given that the troublesome obligation is isolated, it would, of course, be helpful to further state which of the conjuncts of this obligation is giving trouble.

The VC generator should be implemented in such a way that isolating troublesome obligations is possible, even easy. As the generator builds the VC assertions, the choice of names of mathematical variables in these assertions can aid the later task of isolating the troublesome parts. It is important to note here that the tableau method and its related, formal indexed method of generating VCs [1] (to which the author has devoted much time in the past) are *not necessary* for making good variable-name choices. Not even the way the indexed method names variables is necessary.

If the backward sweep method is used to generate VCs, the different name it gives to an actual parameter in "replaces" mode could very likely be enough of a hint to aid the isolation of a troublesome obligation. On the other hand, simplification of the VC as it is generated may make it more difficult to trace its troublesome parts back to the original obligations that gave rise to them. This problem could be addressed by attaching metadata to the VC, thus providing a kind of audit trail. Whatever techniques are adopted and/or invented, the position here is a reminder and an appeal, when VC generators and proof checkers for a verifying compiler are built, to keep in mind the requirement of informing the user what has gone wrong.

Manuscript received May 23, 2007.

W. D. Heym is with The Ohio State University, Columbus, OH 43210 USA (phone: 614-247-6903; fax: 614-292-2911; e-mail: w.heym@ieee.org).

References

[1] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S., and Hollingsworth, J.E., "Reasoning About Software-Component Behavior," in Frakes, W.B., ed., *Software Reuse: Advances in Software Reusability (Proceedings Sixth International Conference on Software Reuse)*, Springer-Verlag LNCS 1844, 2000, 266-283.

Use of Unprovable Verification Conditions for Debugging

Heather K. Harton

***Abstract*—A large portion of the time that goes into forming correct code is devoted to debugging. Given a verification system that can either guarantee that an implementation is correct or show which obligations cannot be discharged, it is possible to debug code and assertions. Using a sequence of examples, this paper experiments with the alternative of using a verifier instead of testing for finding errors.**

***Index Terms*— Verification, Debugging**

I. Introduction

THE purpose of a verification system is to guarantee that an implementation is correct. In the process of developing correct code, however, the verification system will be employed commonly on code that is unprovable for some reason. Thus an important step in the creation of correct code is finding errors when the verification system cannot show correctness.

Typical debugging practices rely on testing to both show the errors and to find the cause. Using specific test cases and observing which test cases fail (and which succeed) and relying on debugging tools to identify where the test cases seem to fail is the normal debugging method. However, it is possible that when verification shows that an error definitately exists, the same false assertions that show an error exists could provide debugging assistance.

There are two possible causes of an error in the verification process. An incorrect specification can cause the verification process to fail. Therefore, it seems that the first step in debugging a program should be to check the correctness of the specification (and other annotations including invariants). Secondly, an incorrect implementation can cause the program not to behave as specified. A verification system can show which proof obligation failed – if the proof obligation does not result from a flawed specification, the implementation must contain errors. Based on the specific proof obligation that failed, it may point to the location of the error.

In this paper, we will explore various examples (with errors) and use the generated assertions from the verification process to assist in finding the cause of the error. The assertions in this paper have been generated using the Resolve verifier available at www.cs.clemson.edu/~resolve.

This research is funded in part by the National Science Foundation grant CCR-0113181 and a grant from the NASA/SC Space Grant Consortium.

H. K. Harton is a student at Clemson University. (e-mail: hkeown@clemson.edu).

II. Examples

A. Example One

The specification of Integer_Template is given in the Appendix. Following is a first example that demands that nothing be done to the Integer that is supplied as a parameter. However, the generated assertion is unprovable. The specification, implementation, and generated assertion follow.

Operation Do_Nothing(**restores** I: Integer);
Procedure Do_Nothing(**restores** I: Integer);
  Decrement(I);
  Increment(I);
end Do_Nothing;

Assertion:
(((min_int <= 0) and (0 < max_int)) implies
  ((((min_int <= I) and (I <= max_int))) implies
    (**(min_int <= (I - 1))** and
    ((((I - 1) + 1) <= max_int) and **I = ((I - 1) + 1)**)));

The highlighted obligations in the assertion must be proved to show the correctness of the code.

- I = I -1 + 1
  - simplifies to the statement, I = I, which is obviously true
- I – 1 + 1 <= max_int
  - simplifies to the statement, I <= max_int, which is true based on the assumption (I <= max_int)
- min_int <= (I – 1)
  - simplifies to the statement, min_int <= I – 1, which cannot be proved

(((min_int <= 0) and (0 < max_int)) implies
  ((((min_int <= I) and (I <= max_int))) implies
    ((**min_int <= (I - 1)**) and
    ((((I - 1) + 1) <= max_int) and
    I = ((I - 1) + 1)))));

Looking at the assertion again, knowing the failed obligation, is it possible to more easily deduce the cause of the error? It seems somewhat clear that if I was not decremented, then this obligation could be guaranteed based on the assumption, min_int <= I. Thus, this seems to point to possibility that if I =min_int, then our error will occur. Thus, adding a requires clause to the Do-Nothing specification or change the code to check for the boundary condition should lead a correct implementation. In this relatively simple example, our assertion seemed to point to a (somewhat

obvious) boundary problem, but will this continue in more complex examples?

B. Max Example

The specification and implementation of a Find_Max procedure follow. The generated assertion is also shown.

```
Operation Find_Max(
                    restores I: Integer; restores J: Integer;
                    replaces K: Integer);
          ensures (K = I or K = J) and
                    (K >= I and K >= J);

Procedure Find_Max(
                    restores I: Integer; restores J: Integer;
                    replaces K: Integer);

      If (Greater(I,J)) then
          I :=: J;
       end;
       K := Replica(J);
end Find_Max;
```

Assertion 1:

```
((((min_int <= 0) and (0 < max_int))) implies
    ((K = 0 and (((min_int <= J) and (J <= max_int)) and
    ((min_int <= I) and (I <= max_int)))) implies
       ((I > J) implies
          (((((I = J or I = I) and ((I >= J) and (I >= I))) and
           I = J) and J = I))));
```

Assertion 2:

```
((((min_int <= 0) and (0 < max_int))) implies
   ((K = 0 and (((min_int <= J) and (J <= max_int)) and
   ((min_int <= I) and (I <= max_int)))) implies
      (not((I > J)) implies
         ((((J = I or J = J) and ((J >= I) and (J >= J))) and
           I = I) and J = J))));
```

The second assertion is easily proved. The following obligations must be showed:

- J = I or J=J
  - True
- J >= I
  - True based on assumption not(I > J)
- J >= J
  - True
- I=I
  - True
- J=J
  - True

The first assertion is not provable. The following obligations must be showed:

- I = J or I = I
  - True
- I >= J
  - True based on assumption I > J
- I >= I
  - True
- I=J
  - False
- J=I
  - False

Looking at the unprovable obligations in the first assertion, we can see because of the assumption, I > J, that this assertion assumes the if part of the rule was executed. Thus, this code fails when I > J. It also becomes apparent if we watch the formation of the assertion, that the obligations I=J and J=I result from the restoration mode of I and J. Now, it becomes clear that I and J are not restored if I > J.

C. Multiplication Example

The specification and implementation of an integer multiply operation follow:

```
Operation Multiply(alters I, J: Integer;
                   replaces K: Integer);
   requires (min_int <= I * J) and
            (I * J <= max_int);
   ensures K = #I * #J;

Procedure Multiply(alters I, J: Integer;
                   replaces K: Integer);
   While (Is_Not_Zero(I))
      changing I, K;
      maintaining K = (#I - I) * #J;
      decreasing |I|;
   do
      K := Sum(K, J);
      Decrement(I);
   end;
end Multiply;
```

Assertion 1:

```
((((min_int <= 0) and (0 < max_int))) implies
   ((K = 0 and (((min_int <= J) and (J <= max_int)) and
   ((min_int <= I) and (I <= max_int)) and
   ((min_int <= (I * J)) and ((I * J) <= max_int))))) implies
      (K = ((I - I) * J) and
      ((?K = ((I - ?I) * J) and ?P_val = |?I|) implies
         (?I /= 0 implies
            ((min_int <= (?K + J)) and
            ((?K + J) <= max_int) and
            ((min_int <= (?I - 1)) and
            ((?K + J) = ((I - (?I - 1)) * J) and
            ((|?I – 1|) < ?P_val)))))))));
```

Assertion 2:

```
((((min_int <= 0) and (0 < max_int))) implies
```

((K = 0 and (((min_int <= J) and (J <= max_int)) and
(((min_int <= I) and (I <= max_int)) and
((min_int <= (I * J)) and ((I * J) <= max_int))))) implies
(?K = ((I - ?I) * J) implies
(?I = 0 implies **?K = (I * J)**)));

The highlighted obligations in the assertion must be proved to show the correctness of the code.

- K = (I - I) * J - Provable
  - simplifies to the statement, 0 = 0 * J based on the assumption, K = 0.
- min_int <= (?K + J) - Not Provable
- (?K + J) <= max_int - Not Provable
- min_int <= (?I – 1) - Not Provable
- ?K + J= (I - (?I - 1)) * J - Provable
  - Simplifies to ((I - ?I) * J) + J = (I - ?I + 1) J
  - I*J - ?I*J + J = I*J -?I*J = J
- |?I -1| < ?P_val – Not Provable
- ?K = I * J -Provable
  - Based on the Assumptions
    - ?K = ((I - ?I) * J
    - ?I = 0

The obligations that were not provable in this situation were the pre-condition clauses for add and decrement. This realization can lead to the conclusion that most likely this is a problem with the boundary values when calling these operations. After closer observation, it becomes clearer that if I < 0, the loop will continue to decrement I until it reaches min_int and at that point an invalid call to Decrement will be made and the program will fail. Thus, the following requires clause will fix that issue:

I >= O and (min_int <= I * J) and (I * J <= max_int).

With that modification, the following assertions are generated:

Assertion 1:
((((min_int <= 0) and (0 < max_int))) implies
((K = 0 and (((min_int <= J) and (J <= max_int)) and
(((min_int <= I) and (I <= max_int)) and (((I >= 0) and
(min_int <= (I * J))) and ((I * J) <= max_int))))) implies
(K = ((I - I) * J) and
((?K = ((I - ?I) * J) and ?P_val = |?I|) implies
(?I /= 0 implies
((min_int <= (?K + J)) and
((?K + J) <= max_int) and
((min_int <= (?I - 1)) and
((?K + J) = ((I - (?I - 1)) * J) and
(|(?I - 1)| < ?P_val))))))));

Assertion 2:
((((min_int <= 0) and (0 < max_int))) implies
((K = 0 and (((min_int <= J) and (J <= max_int)) and
(((min_int <= I) and (I <= max_int)) and (((I >= 0) and
(min_int <= (I * J))) and ((I * J) <= max_int))))) implies
?K = ((I - ?I) * J) implies
(?I = 0 implies **?K = (I * J)**)));

Once again, the highlighted obligations in the assertion must be proved to show the correctness of the code.

- K = (I - I) * J - Provable
  - simplifies to the statement, 0 = 0 * J based on the assumption, K = 0.
- min_int <= (?K + J) - Not Provable
- (?K + J) <= max_int - Not Provable
- min_int <= (?I – 1) - Not Provable
- ?K + J= (I - (?I - 1)) * J - Provable
  - Simplifies to ((I - ?I) * J) + J = (I - ?I + 1) J
  - I*J - ?I*J + J = I*J -?I*J = J
- |?I -1| < ?P_val – Not Provable
- ?K = I * J -Provable
  - Based on the Assumptions
    - ?K = ((I - ?I) * J
    - ?I = 0

At this point, it is observed that our pre-conditions still cannot be proved. It seems that inside the while loop, the assumption that I >= 0 (which is what guarantees those preconditions) is not available. Thus, the following invariant should be used: K = (#I - I) * #J and I >= 0. The following assertions are generated and each condition can be proved:

Assertion 1:
((((min_int <= 0) and (0 < max_int))) implies
((K = 0 and (((min_int <= J) and (J <= max_int)) and
(((min_int <= I) and (I <= max_int)) and
(((0 <= I) and (min_int <= (I * J))) and
((I * J) <= max_int))))) implies
((K = ((I - I) * J) and **(I >= 0))** and
(((?K = ((I - ?I) * J) and (?I >= 0)) and
?P_val = |?I|) implies
(?I /= 0 implies
((min_int <= (?K + J)) and
((?K + J) <= max_int) and
((min_int <= (?I - 1)) and
(((?K + J) = ((I - (?I - 1)) * J) and
((?I - 1) >= 0)) and **(|(?I - 1)| < ?P_val)**)))))));

Assertion 2:
((((min_int <= 0) and (0 < max_int))) implies
((K = 0 and (((min_int <= J) and (J <= max_int)) and
(((min_int <= I) and (I <= max_int)) and
(((0 <= I) and (min_int <= (I * J))) and
((I * J) <= max_int))))) implies
((?K = ((I - ?I) * J) and (?I >= 0)) implies
(?I = 0 implies **?K = (I * J)**)));

Again, the highlighted obligations in the assertion must be proved to show the correctness of the code.

- K = (I - I) * J - Provable
  - simplifies to the statement, 0 = 0 * J based on the assumption, K = 0.

- min_int <= (?K + J) - Provable
  - Using the following assumptions
    - I*J <= max_int
    - ?K = (I - ?I)*J
      - I*J = ?K + (?I * J)
    - min_int <= I*J
  - We get the assumption (min_int <= ?K + (?I * J)) which allows us to prove (because ?I >= 0 and) that min_int <= (?K + J)
- (?K + J) <= max_int - Provable
  - Using the following assumptions
    - I*J <= max_int
    - ?K = (I - ?I)*J
      - I*J = ?K + (?I * J)
    - (I * J) <= max_int
  - We get the assumption (?K + (?I * J) <= max_int) which allows us to prove (because ?I >= 0) that (?K + J) <= max_int
- min_int <= (?I – 1) - Provable
  - because ?I >= 0 and ?I /=0, then, ?I - 1 >= 0 and thus, provable
- ?K + J= (I - (?I - 1)) * J - Provable
  - Simplifies to ((I - ?I) * J) + J = (I - ?I + 1) J
  - I*J - ?I*J + J = I*J -?I*J = J
- |?I -1| < ?P_val – Provable
  - Based on the assumptions
    - ?P_val = |?I|
    - ?I >= 0
- ?K = I * J -Provable
  - Based on the Assumptions
    - ?K = ((I - ?I) * J
    - ?I = 0

III. Discussion

From the examples in this paper, it seems clear that debugging based on unprovable assertions is a reasonable idea. Without the unprovable assertions, good test cases could have discovered some of the problems, but likely would not have caught every error.

In the first example, if the test case that supplied min_int as a parameter was used (and the tester was aware that the value was min_int), the error would most likely have been caught. However, it was clear and quickly found with an observation of the incorrect obligation rather than with extensive testing of all tested inputs.

In the Find_Max example, the assertions easily showed that the code was incorrect. It is questionable if testing would have found this error or helped debug the program because the code did correctly find the maximum value, which is most likely what testing would look for. However, the unprovable obligations easily show which variables are incorrectly set.

In the last example, the first problem discovered was an incorrect pre-condition. It seems likely that testing would have discovered that the program failed if I < 0. At that point, the programmer could have either added the pre-condition 0 <= I or modified the program to handle negative inputs for I. Testing would not have shown the needed modification for the invariant. Although, the code was correct without this modification, it is necessary to prove the correctness and also very helpful for the implementor of the code to have a complete understanding of the behavior of the code which is one positive side effect of requiring an invariant.

It appears that in general, it is helpful to use the unprovable obligations to find errors in the code. Primarily because it is possible that the problem is with the specification instead of the implementation, but also because it is obvious that if in the process of debugging the programmer understands the obligations and how they are formed, it can become simpler to debug.

APPENDIX

```
Concept Integer_Template;
  uses Integer_Theory, Std_Boolean_Fac;

  Defines min_int: Z;
  Defines max_int: Z;

  Constraint min_int <= 0 and 0 < max_int;

  Type Family Integer is modeled by Z;
    exemplar i;
    constraint min_int <= i and i <= max_int;
    initialization ensures i = 0;

  Operation Is_Not_Zero(evaluates i: Integer): Boolean;
    ensuresIs_Not_Zero = ( i /= 0 );

  Operation Increment(updates i: Integer);
    requires i + 1 <= max_int;
    ensuresi = #i + 1;

  Operation Decrement(updates i: Integer);
    requires min_int <= i - 1;
    ensuresi = #i - 1;

  Operation Greater(evaluates i, j: Integer): Boolean;
    ensures Greater = ( i > j );

  Operation Sum(evaluates i, j: Integer): Integer;
    requires min_int <= i + j <= max_int;
    ensuresSum = ( i + j );

  Operation Replica(restores i: Integer): Integer;
    ensures Replica = (i);

end Integer_Template;
```

A Multiparadigm Programming Case Study with RESOLVE Components

Matthew Thornton (thorntom@vt.edu)

***Abstract*—A formal specification language for programs written in a multiparadigm fashion would combine the best of two worlds: a programming platform that allowed developers the freedom to write programs in whatever methodology fits the problem with an unambiguous specification for the program behavior. A proof of concept for a behavioral specification language has already been devised by extending the syntax of the Java Modeling Language. A theoretical validation of this language is already underway. However, a more—practical validation of the language would be ideal, one that would show that the specification language would be "useful" to developers and specifiers alike.**

To demonstrate this, a case study will be developed to show that the programs specified using behavioral specification language devised (JML-MP) have similar attributes in terms of specified collected metrics as an existing specification language—in this case, RESOLVE. This paper will discuss some of the issues that will need to be addressed and raise discussion points for discussions during the workshop.

I. Introduction

A multiparadigm programming language is a programming language that allows a developer to program in more than just one paradigm. Programming languages such as Oz, Leda, JavaMP, and others like it that include Object-Oriented, Functional, Logical, and Imperative paradigms give the developer a rich tool chest to develop software that uses the best tool for the job [1]. At the moment, there is no behavioral specification language that allows one to specify the behavior of programs written in a multiparadigm fashion.

JML-MP is a multiparadigm behavioral specification language that allows one to model behavior of Object-Oriented, Functional, Logical, and Imperative programming paradigms in an extension of Java called JavaMP[7]. JML-MP functions basically as an extension to the Java Modeling Language (JML) [6].

A syntactic definition of the language JML-MP has been completed [9, 10] and a theoretical validation of the language against the programming language it was written for (JavaMP) is currently underway. In many cases, however, a programming language can have all of the desirable theoretical attributes of soundness and completeness and still be a worthless language if people won't use it. Ideally, there would be some way of comparing JML-MP against existing specification tools and techniques and see that writing specifications in that language is similar to writing specifications in another specification language. The question, then, is whether or not this is actually possible. The following sections will discuss our position on this issue and bring to light some of the issues that will need to be addressed moving forward.

II. Position

The position that this paper takes is that it is possible to develop a comparative study of JML-MP using specifications and implementations of components that already exist. This will be accomplished through a case study of an existing program or software component written in RESOLVE, which has many characteristics of existing specification languages [4]. A comparable multiparadigm specification and implementation will then be created and data regarding both sets of specifications/implementations will be gathered and metrics will be generated against the data and the results will be compared. Ideally, the results should show that a JML-MP-specified program or software component is comparable to the comparable RESOLVE software component.

III. Related Work

The idea of comparing and contrasting *programming* languages is nothing new but evaluations of formal specification languages are much harder to come by. Consequently, it will be necessary to make use of some of the work that has already been done with evaluating programming languages. Countless books have been written on the subject [8]. The concepts discussed in books like this are questions of semantic differences, differences in features between languages, the paradigms that the language provides, and others. However, in these cases, the comparison is more of a "black and white" feature comparison. Some work has been done in the area of testing the *usability* of a language [2], but not a lot of work has been done in evaluating more empirical data that can be collected about comparing one language against another.

IV. Case Study Setup

The case study will, again, involve comparing an existing

specification and implementation of a RESOLVE software component. What follows is a brief synopsis of the current plan for the case study.

A. RESOLVE Component to be Evaluated

The RESOLVE component that was chosen to be evaluated is the *Assertion_And_Query_Machine* component. The *Assertion_And_Query_Machine* component is a unit of software that takes in as input assertions. These assertions consist of a label and a sequence of values that that label can be. After that, a structured query can be fed into the machine. The query can have bound and unbound variables in it. The machine then generates the appropriate bindings based on the query and allows the client to process the bindings by arbitrarily removing the bindings from the machine.

There are several reasons why this component was chosen. For one, the component is sufficiently long and complex so as to allow for a useful case study to be done. Another (potentially more important reason) is that the *Assertion_And_Query_Machine* component has many attributes that would lend itself to a multiparadigm design. Processing a list is something that the Functional programming paradigm is used for. Executing search and query functionality is something that immediately lends itself to a logical programming paradigm [1]. Developing a JavaMP component based on the RESOLVE component would allow for many interesting multiparadigm design issues to be addressed.

B. Multiparadigm Design and Specification

In Section A, it was stated that there were several interesting design issues that would have to be addressed in implementing a design and specification for the *Assertion_And_Query_Machine*. First of all, class design differs considerably between RESOLVE C++ and Java (and its JavaMP variant). To implement a *proper* class design in Java, appropriate design patterns [3] should be instituted, where appropriate. This brings up the immediate concern of how to integrate additional paradigms into the Java design methodology. In [5], the authors address the issue by providing several examples of patterns in a multiparadigm fashion, including iteration patterns, command patterns, and generators. These concepts and others need to be investigated to see how to best integrate them into the design of the *Assertion_And_Query_Machine* for JavaMP.

The specification of the JavaMP class(s) that are defined in creating a JavaMP counterpart to the RESOLVE component will be done using the draft version of JML-MP. This will allow us to iron out the bugs of the specification language and verify that we actually have everything we need to effectively write a specification for such a component. Many of the issues that will need to be addressed include exactly how specific a specification should be given to the lambda functions that are created as members of a class or that are parameters to member methods.

C. Data Collection

Once the design, specification, and implementation of the JavaMP counterpart to the *Assertion_And_Query_Machine* is completed, we will begin to gather data about both components. This data will include easily-accessible data such as the number of lines of code, numbers of lines of specification, number of methods, and other data. It would also include calculation of complexity measures of the implementation. For *that* matter, some form of complexity measure could be devised for the specification, as well (including some calculation involving the number of nested quantifiers, etc.). Measures taken from the specification such as the number of quantifiers, number of decision paths in the specification, and other will be included. Additionally, as much information related to the design and implementation *process* will be gathered as possible (time it took to design, specify, and implement, measures of numbers of iterations/versions, etc). This data could then be used in gathering useful information about comparing and contrasting the two specifications and their implementation.

D. Metrics Evaluation

There are a number of interesting metrics that have already been considered for the evaluation of the two specifications. The goal of the evaluation is to see that writing the specification in JML-MP is comparable to writing it in a language such as RESOLVE. What do we mean by "comparable"? It would be nice if the specifications were about the same length, complexity, difficulty to write, and other measures. Some of the metrics that would allow us to gleam more information out of the data that was collected would be metrics like complexity per line of code/specification (this would give us a measure of how "dense" the specification was), how long the implementation is versus how long and/or complex the specification is, how long it took to write a line of the specification, and other calculations.

If it were possible to gather similar metrics of other RESOLVE specifications and then do a statistical analysis of the results gathered and then include the JML-MP specification in the mix and we see that the metrics and data we collected from the JML-MP specification is within tolerance or an acceptable alpha, then we could say that JML-MP is comparable to writing a specification for a RESOLVE component.

V. Conclusions and Future Work

Coming up with a case study that shows how *similar* two languages are seems to be hard to come by. Furthermore, coming up with a way of showing how similar two *specification* languages are seems even more difficult. The case study approach that has been adopted should allow for us to see how similar the JML-MP specification is to other similar RESOLVE components.

The preceding has just been a proposal. Since the case study has not yet begun, it would be nice to get feedback on

this paper. Particularly, information in the following topic areas would be useful:

- Suggestions on design and how to incorporate other paradigms into design and implementation of a JavaMP *Assertion_And_Query_Machine* class.
- Suggestions on data points to collect in the design, specification, and implementation of the *Assertion_And_Query_Machine* class.
- Recommendations on how to do the analysis of the data and metrics collected.
- Suggested metrics to take based on the data that has been collected.

There is already a theoretical model for the viability of writing program specifications for a multiparadigm program. If we can demonstrate that it is also *practical* to write such a specification, it could go a long way to improving the multiparadigm community of adopting formal methods into their research.

REFERENCES

[1] Budd, T., T. Justice and R. Pandey, *General-Purpose Multiparadigm Programming Languages: An Enabling Technology for Constructing Complex Systesms*, Engineering of Complex Computer Systems 1995, IEEE, Ft. Lauderdale, FL, 1995, pp. 334-337.

[2] Clarke, S., *Evaluating a New Programming Language*, Psychology of Programming Interest Group, Bournemouth University, UK, 2001.

[3] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Indianapolis, IN, 1995.

[4] Heym, W. D., T. J. Long, W. F. Ogden and B. W. Weide, *Mathematical Foundations and Notation of RESOLVE--OSUCISRC-8/94-TR45*, in Department of Computer and Information Science--The Ohio State University, ed., 2000.

[5] Knutson, C. D., T. A. Budd and C. R. Cook, *Multiparadigm Patterns of Thought and Design*, Pattern Languages of Programs Conference, Allerton Park, Illinois, 1996.

[6] Leavens, G., E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Muller and J. Kiniry, *JML Reference Manual*, 2005.

[7] Naik, R., *Multiparadigm Programming with JavaMP*, Electrical Engineering and Computer Science, Oregon State University, Corvalis, 2003.

[8] Sebesta, R., *Concepts of Programming Languages*, Addison Wesley, 2003.

[9] Thornton, M., *Behavioral Specification of Multiparadigm Programs*, in Stephen Edwards, ed., RESOLVE 2006 Workshop, Virginia Tech, Blacksburg, VA, 2006.

[10] Thornton, M., *JML-MP: a Formal Specification Language for a Multiparadigm Programming Language (Abstract)*, Fall Conference of the Mid-Southeast Chapter of the ACM, Gatlinburg, TN, 2006.

The Collaborative Reasoning Paradigm

Jason O. Hallstrom, Joan Krone, Richard Pak, and Murali Sitaraman

The Paradigm

The central goal of the *Collaborative Reasoning* paradigm is to teach analytical reasoning skills that help students learn not only how to develop correct programs, but to reason and understand why their programs are correct. This will complement the more common learning pattern of developing programs through a trial and error process of program execution. To engage and excite students about learning the central principles of reasoning, a collaborative reasoning paradigm is appropriate. This teaching paradigm is motivated, in part, by the conclusions of Martinez and Eisenhart after performing a case study of various pedagogical techniques in Physics [Martinez 04]:

> "The alternative instructional methods that show promise are various activities that allow students to be active class participants, procedures or devices that give instructors quick ways to assess their students' understanding during class, in-class opportunities to discuss students' understandings and difficulties, and activities specially designed to be fun, challenging, and relevant."

Under the new paradigm, teams of students are involved in problem solving; often, students work on detailed steps of reasoning individually and join their results together to learn from each other and to solve problems collectively.

Collaborative Learning

To engage students in the learning process, the Collaborative Reasoning paradigm leverages multiple modes of collaboration. The concept is a variation on the idea of learning through one's peers. It builds upon Mazur's *Peer Instruction* approach [Mazur 97] pioneered at Harvard in teaching calculus and introductory Physics courses.

Through over a decade of evaluation, Peer Instruction has been shown to be effective, with significant improvements in the understanding of concepts in the area of *Force Concept Inventory* [Crouch 01]. In some cases, almost twice as many students understood a concept when using the approach as compared to those taught using traditional instructional techniques. One key to the success of Peer Instruction is its activity-based nature. Students are given a problem solving activity on which they're asked to work individually. After they've had an opportunity to formulate their answers, they either try to convince their peers of the validity of those answers, or work to debug the answers with the help of their peers. Whether or not their answers are correct, and whether they're working to convince others or to debug their solutions, understanding of the target concept is reinforced. One variation of Peer Instruction involves the instructor, and relies on classroom discussion. Whether or not the instructor is included, group cooperation in the learning process is a key reason why the approach has been shown to engage and excite students.

The Collaborative Reasoning paradigm, like the Peer Instruction approach, is intended to enhance cooperative learning from peers, and thus engage and excite students. In teaching reasoning principles in computing, there are several interesting places for pairs and/or teams of students to collaborate, and to give and receive feedback.

Reasoning Workbench

Principles of reasoning can be taught using a collaborative approach without any computers at all. However, in modern Computer Science classrooms, collaborative learning can be significantly enhanced through computer-assisted feedback mechanisms. This is because a variety of guided, interactive exercises become possible when computer-aided reasoning assistants are integrated in the classroom. Further, if the computers are network-enabled, an additional class of exciting collaborative activities becomes possible. Pargas and Shah [Pargas 2006] report the benefits of applying a variation of Peer Instruction using wireless "*clickers*" that allow students to submit solutions to multiple choice questions in a CS classroom. Using their tool, an instructor may choose to share aggregate response data with the class, giving students a chance to reformulate their answers collaboratively through discussions with their peers. Pargas and Shah attribute improved awareness of deficiencies in content understanding, by both students and instructors, to the use of instant feedback and response collection.

A computer-aided approach can be tuned to provide instant or delayed feedback. Some studies have documented the benefits of delayed feedback on learning and performance in some situations [Brackbill 62, Kulhavy 72], while others have discussed the merits of immediate feedback [Azevedo 95,

This research is funded in part by a grant from the National Science Foundation (DUE-0633506).

Jason O. Hallstrom and Murali Sitaraman are with the School of Computing at Clemson University (jasonoh@cs.clemson.edu, murali@cs.clemson.edu).

Joan Krone is with the Department of Mathematics and Computer Science at Denison University (krone@denison.edu).

Richard Pak is with the Department of Psychology at Clemson University (rpak@clemson.edu).

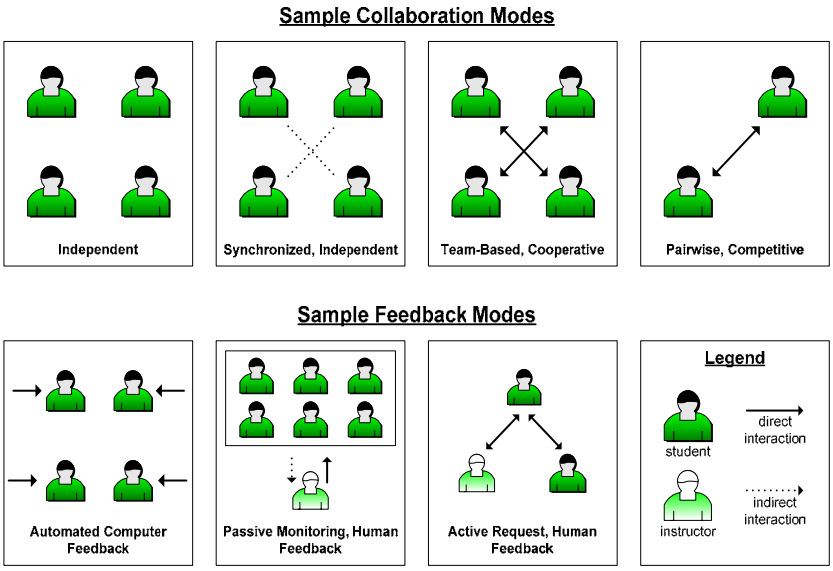


Figure 1. Sample Collaboration and Feedback Modes

Bangert 91]. The argument for immediate feedback is that it prevents learners from floundering and can lead to the same level of learning in less time; if corrected immediately, students may not encode incorrect knowledge into memory [Anderson 95, Corbett 01]. The benefits of rapid, in-class feedback are also documented among the findings of Martinez and Eisenhart [Martinez 04]. The argument for delayed feedback is that while learners may not benefit from the immediate correction offered by rapid feedback, they learn more working through each error. Ultimately, the most suitable feedback mechanism (*i.e.*, delayed or rapid) may depend on individual differences among students.

Our goal is to design a c*ollaborative reasoning workbench* that can guide collaboration and provide rapid, timed, or delayed feedback to students. In some cases, after a student attempts to solve a particular problem, the corresponding *reasoning assistant* will be able to determine the correctness of the student's solution automatically. In other cases, the assistant will help to bring together individual solutions used in solving a larger puzzle, highlighting any inconsistencies among the team members in the process. In other activities, the instructor will be involved, providing human feedback to supplement machine feedback. The instructor might also detect a common misconception among the collaborating teams, and project individual or group solutions on a screen to help resolve the common difficulty. There are several other modes of collaboration that will be supported, each of which involves feedback, solution sharing, and assistance from automated and human reasoning assistants. Some of the key collaboration and feedback modes that will be supported by the workbench are illustrated in Figure 1.

The workbench will be used to teach principles of reasoning about the operational semantics of programming languages, specification-based testing, specification-based reasoning, and (even) performance analysis. The student interface will be similar to existing step-wise debugging tools, but will guide students in *predicting* the behavior of programs based on their understanding of the techniques under study. If one or more predictions are incorrect, the student will be notified of the error. We expect that the approach will be especially valuable in teaching students the concept of loop invariants, as advocated by Henderson in [Henderson 03].

The instructor interface will allow professors in the classroom, and designated teaching assistants outside of the classroom, to monitor the individual and aggregate progress of their students, and to provide real-time or delayed feedback when necessary. The instructor interface will allow a professor or a designated teaching assistant to monitor the progress of their students as they work through their exercises. The instructor view will provide information concerning the individual progress of each student, as well as the aggregate progress of a (potentially remote) class or study group. When a point of confusion in reasoning is noticed, the instructor will be able to provide feedback in the form of instant messages, and to direct the feedback to an individual or to a group. Alternatively, or in combination, if the students and the instructor are co-located, the instructor might choose to disable the exercise temporarily and discuss the common difficulty. The active collaboration features will allow students to initiate feedback requests directly. Depending on the settings specified for the current exercise, students will be able to request help from their instructor or from a randomly selected student who has already completed the exercise. The second alternative is especially interesting, as it provides the foundation for group learning environments where students can practice their exercises without an instructor and receive feedback from peers as they work. Further, the workbench will provide support for turn-based collaboration, allowing students to complete their reasoning exercises cooperatively.

Students can take turns predicting the behavior of each program statement, and judging the correctness of their peers' predictions. In this mode of operation, the reasoning becomes a kind of game, and provides an exciting opportunity for learning.

REFERENCES

1. Anderson, J. R., Corbett, A. T., Koedinger, K. R., and Pelletier, R., Cognitive Tutors: Lessons Learned, *The Journal of the Learning Sciences*, 4, 167-207, 1995.

2. Azevedo, R., and Bernard, R. M., "A Meta-Analysis of the Effects of Feedback in Computer-Based Instruction", *Journal of Educational Computing Research*, 13(2), 111 – 127, 1995.

3. Bangert-Drowns, R. L., Kulik, C. C., Kulik, J. A., and Morgan, M, "The Instructional Effect of Feedback in Test-Like Events", Review of Educational Research, 61, 213-238, 1991.

4. Brackbill, Y., Bravos, A., and Starr, R. H., "Delay-Improved Retention of a Difficult Task", *Journal of Comparative and Physiological Psychology*, 55, 947-952, 1962.

5. Corbett, A. T., Anderson, J. R., "Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes", *Proceedings of the 2001 SIGCHI Conference on Human Factors in Computing Systems*, 245-252, 2001.

6. Crouch, C. H. and Mazur, E., "Peer Instruction: Ten Years of Experience and Results", *American Journal of Physics*, Vol. 69, No. 9, September 2001, 970–977.

7. Henderson, P. B., "Mathematical Reasoning in Software Engineering Education", *Communications of the ACM 46*, September 2003, 45-50.

8. Kulhavy, R. W., and Anderson, R. C., "Delay-Retention Effects with Multiple Choice Tests", *Journal of Educational Psychology*, 63, 505-512, 1972.

9. Martinez**,** K., Eisenhart**,** M., *L.E.A.P. – Literature Review of Best Practices in College Physics and Best Practices for Women in College Physics***,** January 2004; available at**:** http://advance.colorado.edu/researcg_best-_practices.html.

10. Mazur, E., *Peer Instruction: A User's Manual*, Prentice–Hall, Upper Saddle River, NJ, 1997.

11. Pargas, R.P., Shah, D.M., "Things are Clicking in Computer Science Courses", *Proceedings of the. 37th SIGCSE Technical Symposium on Computer Science Education*, ACM, 2006.

The Sulu Tools for Automated Testing

Roy Patrick Tan
Department of Computer Science
Virginia Tech
Blacksburg, Virginia
Email: rtan@vt.edu

***Abstract*— The Sulu programming language is a programming language inspired by Resolve; it uses the software component and an embedded design-by-contract formal specification as inputs to a unit testing system that provides a framework for automatically generating, executing and evaluating test cases. This paper describes the tools we constructed for the automated testing system, and a summary of an evaluation done to show the effectiveness of this approach.**

I. Introduction

The state of the art in unit testing, as popularized by the Test-Driven Development practice, requires manually writing test cases as well as manually asserting what the correct state of the program should be after a test case is executed. Because the developer's time is limited, a programmer usually writes a relatively small number of test cases that attempt to meaningfully capture the behavior of the software under test.

We believe that we can complement this process with an automated testing system that provides an additional layer of dynamic error checking, and frees the programmer to write tests for more difficult to find bugs. In the previous Resolve workshop we described a programming language called Sulu [1]. In summary, Sulu is a programming language that takes several features inspired by Resolve such as:

- Use of generics as a composition mechanism
- Separation of specifications from implementation
- Ability to have multiple implementations for one specification
- Support for alternative data movement operations

However, Sulu also has more object-oriented features than Resolve, and has a design-by-contract style of executable specification that can be used for automated testing. In this workshop we describe the tools created that completes the automated testing system for Sulu.

II. The State of the Art

Software testing in various stages of the development lifecycle constitute three parts: *selection* or *generation* of specific test cases, *execution* of these test cases, and *evaluation* of the quality of the software under test, but also of the test cases themselves.

Test cases generation involves selecting a particular set of test cases (a *test suite*) within an often practically infinite domain of program execution. To the extent that software developers actually perform unit testing, most test cases are written manually, in a format that can be run by an automated test case execution tool such as JUnit. Included in these test cases are statements that assert the correct state of the software under test after the test case is executed. The execution tool then reports whether the test cases passed or failed.[1]

However, knowing whether a set of test cases found bugs or not is only one part of the evaluation process. The test cases themselves should be evaluated to determine how thoroughly they test the software. Increasingly the metric used for this determination is code coverage [2]. Statement coverage, for example is a metric that tells the tester the percentage of statements in the software under test that was executed by the test suite.

Commercial applications that generate test suites automatically such as Parasoft's Jtest tool are available, although it is unclear how effective such tools are.

III. An Integrated Approach

We believe there is space for greater automation of unit testing. We envision the automatic generation, execution, and reporting of baseline test cases as becoming an integral part of the software development lifecycle. We thus built a suite of software tools that include:

- A programming language called *Sulu* that supports design-by-contract style specifications.
- An interpreter for the Sulu programming language that collects code coverage information
- A pluggable architecture for various test case generation strategies
- An automated test case execution tool
- An extensible mutation testing tool

These tools constitute a proof-of-concept system that integrates automated testing into the software development process. Using these tools (and a text editor) a set of 10 reference software components were developed. Figure 1 is a diagram of the overall architecture of the Sulu tools for automated testing.

The only input required by these tools is the component under test, which contains an embedded design-by-contract specification of its own behavioral requirements. Pluggable automatic test case generators then take this information and produce one or more test suites, which can be run by the test execution tool.

[1] In this paper we follow the JUnit convention where a test case is said to *pass* if the software under test executed the test case correctly, and *fail* if the software behaved incorrectly (i.e., when a bug is found).

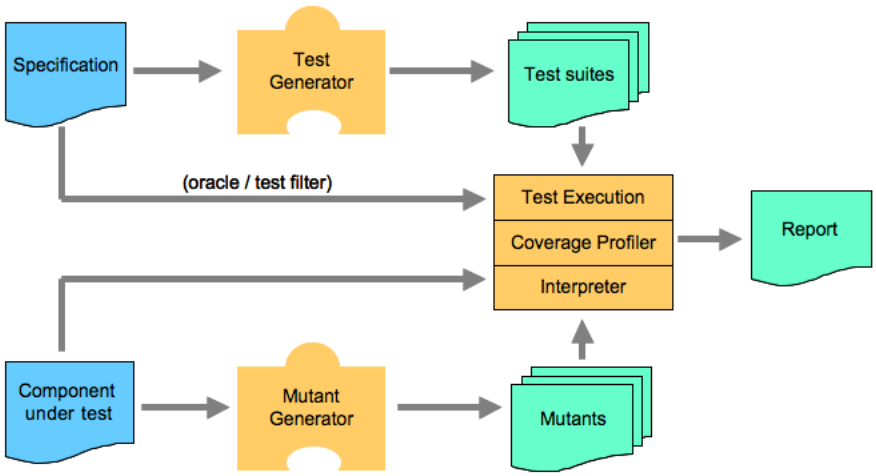


Fig. 1. The Sulu automated testing system

The test suites generated by the test generation tool are in the same format as manually generated test suites, and thus can be run in exactly the same way. A key question with automatically generated test cases is deciding whether for a particular test case, the software under test behaved correctly or not. This is known as the *oracle problem* [3]. In our system, we use the design-by-contract specification, particularly the postconditions, as the test oracle. If a postcondition fails because of a test case, we flag that test case also as failed, i.e. a bug was found. If the postcondition does not fail, we say that the test case passed, the software under test behaved correctly. A variant of the oracle problem is the problem of deciding whether test cases are valid. In many instances, a test case may be invalid because it violates a precondition; with the Sulu system, we also use the DBC specifications as a *test filter*. Test cases that cause a precondition failure are considered *invalid*.

The Sulu interpreter includes a code coverage profiler that collects three code coverage measures: statement, condition, and condition/decision coverage [4]. In addition, a mutation analysis tool is also included. Using the mutation analysis tool, we can generate a set of software components each of which differs from the original in exactly one way (such as, for example, one plus operator changed to a minus operator). Thus, a test suite may be run against a mutant, if the test suite fails, the mutation is detected by the test suite, and the mutant is said to be *killed*.

Our mutation analysis tool generates mutants, runs each mutant against a test suite, and reports the number of mutants killed versus the total number of mutants. Our mutation analysis tool also has pluggable mutant generators, so that the tester may add different kinds of mutant generators as needed.

Thus, the Sulu tools for automated testing provides all the necessary infrastructure for generating, executing, and evaluating test suites, given the specification and implementation of a software component. We believe that taken together, the Sulu programming language and its testing tools allows us to have an effective layer of automated software testing that can reduce bugs and increase the reliability of software.

IV. Issues Addressed

Our integrated approach to software testing addresses several issues that arise with automated testing:

a) Manually written test cases are better than automated ones. The Sulu tools do not seek to replace human-written tests, rather, augment and complement them. Automatically generated test cases provide an extra baseline layer of tests. Sulu allows people to write manual unit tests, and the test execution tool will run the manual tests along with the automatically generated ones. Additionally, the evaluation tools will help the tester find the areas of the code that is inadequately tested, and guide him to write tests for those.

b) Automatically generated test cases don't work for all kinds of software components. Sulu enables programmers to create new test generators as plug-ins to the testing system. Thus, programmers will be able to choose from a set of test case generators that is effective for the kind of software he writes.If none of the predefined test-case generators are appropriate, our framework enables programmers to develop their own.

c) Mutation testing does not reflect real life bugs. Although recent research shows evidence to the contrary [5], the mutation tool also allows people to easily plug-in new mutants that reflect other kinds of bugs.

d) It is difficult to compare different automated testing strategies. A recurrent problem in automated testing research is the evaluation of the relative effectiveness of differing testing strategies. By providing a robust evaluation subsystem, and a pluggable test case generation mechanism, Sulu provides a *benchmark* upon which different testing strategies can be compared against each other.

Concept	Realization
BinaryTree	Standard
List	TwoStacks
Map	Hashtable
Stack	LinkedList
Sorter	BubbleSort
Sorter	HeapBased
Sorter	ListBased
Sorter	MinFinding
Vector	ArrayBased
Vector	LinkedList

TABLE I
SULU REFERENCE COMPONENTS USED FOR EVALUATION

V. EVALUATION

To evaluate the effectiveness of our approach, we implemented 10 different collection components, many of which correspond to traditional Resolve components, as shown in the table on Table I. Each of these components have a formal design-by-contract specification.

We developed a test case generator that generates sequences of method calls, based on the work by Edwards [6]. From this test case generator we produced 6 different test suites for every component in Table I. We then executed each test suite against the corresponding component and gathered code coverage information.

We also implemented 5 different mutation generators, corresponding to a set reported by Andrews and his colleagues [5], of mutation operators that have some evidence of corresponding to real bugs.[2] These are:

1) Change an integer constant to one of $0, 1, -1$, and its negation
2) Change an arithemetic operator to another arithmetic operator
3) Change a comparison operator to another comparison operator
4) Force an if statement or while loop to either *true* or *false*
5) Delete a statement

e generated mutants for each component under test using each mutation generator, and ran all test suites against each mutant. We then gathered the percentage of mutants killed.

At the time of writing, we are still completing the analysis for the data we gathered during this study. We do, however, have promising preliminary information. The results show that for the most comprehensive automatically generated test suite (all triples with 2 parameters), we achieve 90% statement coverage and nearly 80% decision and condition/decision coverage with the reference components. Mutation analysis also shows high coverage, with kill ratios of above or nearly 75% for 3 of the 5 mutation operators. In contrast, a recent conversation with a Microsoft employee revealed to the author that Microsoft's code coverage bar during the development of their software is 75% coverage (it is unclear whether this was statement coverage or decision coverage).

[2]Note that a sixth operation, the change of one boolean operation to another, is not yet implemented.

VI. RELATED WORK

There are a few different groups that have similar approaches to automated unit testing. The first is the work by Cheon and his colleagues [7], [8], on the JMLUnit tool which automatically generates test cases for Java using JML, the Java Modeling Language [9]. In an earlier paper [10], we describe our experience evaluating their earlier work. Their more recent work [8] includes a fairly sophisticated mechanism for generating test cases. However, the evaluation of the defect revealing capabilities rely on a small number of hand-seeded defects.

Leitner and his colleagues [11] use a similar mechanism of taking design-by-contract specifications in the Eiffel language and using them as test oracles. They do not report any mutation analysis results, although they do gather code coverage information. Their test case generation and execution mechanism was shown to find bugs in production-level code.

Korat [12] uses a white-box approach to generating test inputs, using invariants to filter out invalid states. For future work, we might want to explore a similar method to generate, for example, parameter values to test cases in the current test case generator.

Model-checking approaches to generating white-box tests have also been an active are of research. For example, Fraser and Wotawa [13] have an interesting approach that has a high level of mutation coverage. However, their mutation analysis uses mutants that are specifically targeted by their test cases generator.

A common thread among many of these is a lack of common measures for the bug detection strengths of these automated testing systems, much less provide a comparison with other automated testing systems. They also present the evaluation of the thoroughness of test suites as a separate task, rather than integrated into the automated testing system themselves. We hope that by presenting the Sulu approach to software testing, we provide a mechanism to evaluate an arbitrary automated testing strategy, and benchmark different test case generation strategies against each other.

VII. FUTURE WORK

Since one of the goals of Sulu is to be used for benchmarking different test cases generation strategies, 2 avenues for future research immediately comes to mind.

The first is to develop and plug in additional test case generators. Perhaps take the JMLUnit or Korat test generation strategy, apply it to Sulu components and gather code coverage and mutation coverage with currently existing code coverage, mutation operators, and software components.

A second path for future research is to add to the current set of software components, including components that are not collection types. For example, it is unclear how well the test case generator works for I/O bound software, or GUI software. Along the way, we need to explore the best ways to write good

specification and tests for I/O-bound and event-based software components.

Refinements to the programming language and the specification language may provide interesting insights into future research as well. In conclusion, we believe that the Sulu programming language and tools can provide a fertile platform for future research.

REFERENCES

[1] R. P. Tan, "An overview the sulu programming language," in *Proceedings of the Resolve Workshop 2006*. Virginia Tech, 2006.

[2] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997.

[3] M.-C. Gaudel, "Testing can be formal, too," in *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*. London, UK: Springer-Verlag, 1995, pp. 82–96.

[4] H. K. J., V. D. S., C. J. J., and R. L. K., "A practical tutorial on modified condition/decision coverage," NASA, Tech. Rep., 2001.

[5] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[6] S. H. Edwards, "Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential," *Software Testing, Verification and Reliability*, vol. 10, no. 4, pp. 249–262, 2000. [Online]. Available: citeseer.nj.nec.com/448240.html

[7] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," in *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2002, pp. 231–255.

[8] Y. Cheon, "Automated random testing to detect specification-code inconsistencies." The University of Texas at El Paso, Tech. Rep. 07-07, February 2007.

[9] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A notation for detailed design," in *Behavioral Specifications of Businesses and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Kluwer Academic Publishers, 1999, pp. 175–188. [Online]. Available: citeseer.nj.nec.com/leavens99jml.html

[10] R. P. Tan and S. H. Edwards, "Experiences evaluating the effectiveness of jml-junit testing," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–4, 2004.

[11] A. Leitner, I. Ciupa, B. Meyer, and M. Howard, "Reconciling manual and automated testing: The autotest experience," *hicss*, vol. 0, p. 261a, 2007.

[12] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM Press, 2002, pp. 123–133.

[13] G. Fraser and F. Wotawa, "Property relevant software testing with model-checkers," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 6, pp. 1–10, 2006.

Accessibility Preserving Operations for Pointers

Gregory Kulczycki and Amrinder Singh

Abstract— **The Resolve pointer component captures the efficiency of traditional pointers while allowing clients to reason about their behavior as they would with any other component. Proving the correctness of operations involving pointers usually includes proving that no memory leaks are created during the operation. One way to do this is to show that all locations accessible prior to an operation are still accessible after the operation. Swapping locations – or links that point to locations – preserves the accessibility of a system of linked locations. This paper explores how location-swapping operations can effect the implementation and reasoning of pointer-based operations and offers some preliminary observations on their usefulness.**

Index Terms—**Resolve, Swapping, Linked locations, Schorr-Waite**

I. Introduction

THE Resolve pointer component – described in [3] and used to demonstrate reasoning about a splice operation in [4] and [5] – provides various operations that are especially useful for implementing linked data structures such as lists and trees. Examples of its use so far tend to employ operations such as Relocate, Follow_Link, and Redirect_Link to move a position variable or a link from one location to another. These operations are simple to use because they have counterparts in the traditional programming world. For example, Relocate(p, q) would correspond to a pointer assignment, $p := q$, in a more traditional programming language where p and q are nodes containing objects and pointers to other nodes. Follow_Link(p, NEXT) would correspond to $p := p$.NEXT in such a language, while Redirect_Link(p, NEXT, q) would correspond to p.NEXT := q.

In contrast to these operations, the primary operation Swap_Locations has been used only sparingly. The call Swap_Locations(p, NEXT, q) ensures that p's NEXT link points to q's original location and q moves to the location originally pointed to by p's NEXT link, as in Figure 1. Since we wish to make it clear that we are swapping a link and a position variable, we will henceforth refer to this operation as Swap_Link_Pos instead of Swap_Locations.

Manuscript received May 22, 2007.

G. Kulczycki is an assistant professor at Virginia Polytechnic Institute and University in Falls Church, VA 22043 USA (e-mail: gregwk@vt.edu).

A. Singh is a Master's student at Virginia Polytechnic Institute and University in Falls Church, VA 22043 USA (e-mail: asingh05@vt.edu).

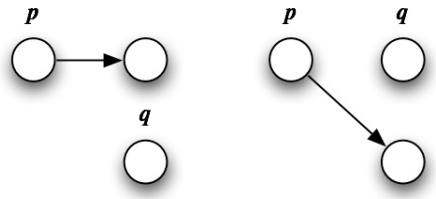


Fig. 1. Effect of call Swap_Link_Pos(p, NEXT, q);

A benefit of this operation is that the accessibility of the system of linked locations remains unchanged after the call. In other words, if a location could be accessed by a client before the call, it could also be accessed by a client after the call. This is because anything previously accessible from q is now accessible from p through its NEXT link, and anything previously accessible from p through its NEXT link is now accessible from q. Locations that were accessible from other variables remain accessible from them, and locations the were inaccessible remain inaccessible. The fact that the accessibility of the system does not change guarantees that the operation did not produce any memory leaks: locations that remain taken but become inaccessible. In this position paper we explore operations that do not change the accessibility of a system and show how they can be used when implementing various pointer-based operations.

II. Accessibility Preserving Operations

Before we talk more about the access preserving operations, let us convince ourselves that primary operations Relocate, Follow_Link, and Redirect_Link do not preserve accessibility and can indeed create memory leaks. Figure 2 shows the effect of the call Relocate(p, q) on a simple system consisting of two linked locations. After the call, p's original location is inaccessible, creating a memory leak. Note that the call Follow_Link(p, NEXT) will have the exact same effect.

Fig. 2. Effect of Relocate(p, q) or Follow_Link(p, NEXT);

The call Redirect_Link(p, NEXT, q) in Figure 3 also creates a memory leak. The location originally pointed to by p's NEXT link becomes inaccessible.

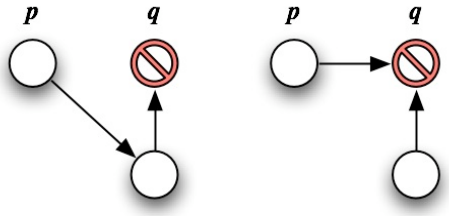


Fig. 3. Effect of Redirect_Link(*p*, NEXT, *q*);

Instead of these operations, we will explore using operations that preserve the accessibility of a system. The operation Swap_Link_Pos has already been described above. Another operation that preserves accessibility is simply the swapping of two pointers: $p :=: q$. After the operation, anything previously accessible from p is now accessible from q, and anything previously accessible from q is now accessible from p. Another operation that we have found useful does not appear in the current description of the pointer component. Swap_Links(p, m, q, n) simply redirects p's m-th link toward the original location of q's n-th link, and vice versa. Accessibility is preserved for reasons similar to those given for the other operations. Sometimes we will talk about accessibility (or reachability) with regard to the two position variables given as arguments to the call. For example, we might note that these operations preserve p-q-reachability. That is, all (and only those) locations reachable from either p or q before the operations will be reachable from either p or q after the operations.

A. *Introducing Cycles*

While these operations can make reasoning about general accessibility easier, they have their limitations. For example, one often wants to show that an operation does not create a cycle. In systems with only one link per location, this is often done by stating that the Void location continues to be reachable from some root position. Unfortunately, all the operations discussed so far that modify the *Target* variable can create new cycles, as illustrated in Figures 4 and 5.

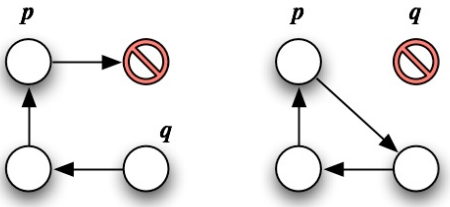


Fig. 4. Cycle created by Swap_Link_Pos(*p*, NEXT, *q*);

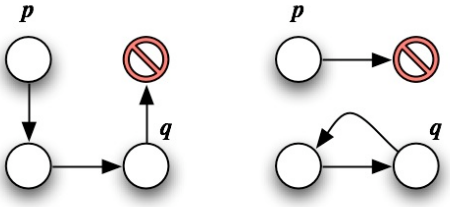


Fig. 5. Cycle created by Swap_Links(*p*, NEXT, *q*, NEXT);

B. *A Repeated Argument Problem?*

How do we handle the call Swap_Link_Pos(*p*, NEXT, *p*)? According to the semantics of repeated arguments, this would be equivalent to Swap_Link_Pos(*p*, NEXT, Void), which would give us the effect shown in Figure 6.

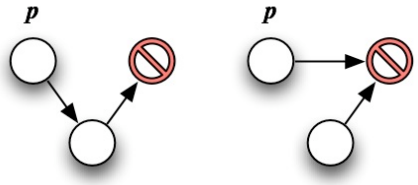


Fig. 6. Possible effect of call Swap_Link_Pos(*p*, NEXT, *p*);

The drawback of this operation is that accessibility is not preserved. We can make the case that pointer operations are treated differently for the sake of efficiency. In this case the operation would be like swapping *p*.NEXT with *p* in a traditional language, which would give us the effect shown in Figure 7.

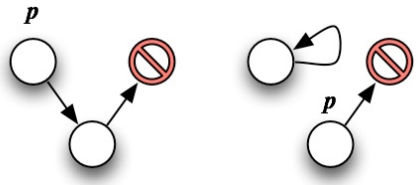


Fig. 7. Alternative possible effect of call Swap_Link_Pos(*p*, NEXT, *p*);

Again, accessibility is not preserved. The easiest solution would be to preclude cases in which formal parameters *p* and *q* share the same value (location), though this would rule out the potentially useful – and accessibility preserving – call Swap_Link_Pos(*p*, NEXT, *q*) when $p = q$.

III. Examples Using Access Preserving Operations

This section gives three examples that use the accessibility preserving pointer operations in place of more conventional operations. We show both implementations. Exclamation marks indicate operations that may alter system accessibility.

A. *List Insert Operation*

```
Procedure Insert(alters E: Entry; updates S: List);
    Var post, new: Position;
    Relocate(post, pre);                  // !!!
    Follow_Link(post, NEXT);              // !!!
    Take_New_Location(new);               // !!!
    Swap_Contents(new, E);
    Redirect_Link(S.pre, NEXT, new);      // !!!
    If S.rem_length = 0 then
        Follow_Link(S.last, 1);           // !!!
    end;
    S.rem_length := S.rem_length + 1;
end Insert;
```

Listing 1. Traditional implementation of list insert

The first implementation of the list insert operation (Listing 1) is given in [3]. There are five statements that potentially modify the accessibility of the system.

The modified code in Listing 2 eliminates some, but not all, of the accessibility changing code. One of the conventions in the list implementation is that ∀*q*: Location, Is_Accessible(*q*) **iff** Is_Taken(*q*), which guarantees no memory leaks or dangling references. Since Take_New_Location(*new*) creates one location that is both accessible and taken, our convention is still satisfied after the statement. The next few statements preserve both accessibility and taken status, so it is trivial to see that our convention holds even internally until we get to the Follow_Link invocation. At which point we must use more sophisticated logic to reason that all our taken locations are still accessible. Note that we were able to dispense with one local variable.

```
Procedure Insert(alters E: Entry; updates S: List);
    Var new: Position;
    Take_New_Location(new);     // !!!
    Swap_Conents(new, E);
    Swap_Links(pre, NEXT, new, NEXT);
    Swap_Link_Pos(pre, NEXT, new);
    If S.rem_length = 0 then
        Follow_Link(S.last, 1);     // !!!
    end;
    S.rem_length := S.rem_length + 1;
end Insert;
```

Listing 2. Implementation of list insert using accessibility-preserving operations

B. *Splice Operation*

```
Operation Splice(preserves p: Position; clears q: Position);
  updates Target;
  requires ( ∃k1, k2: N ∋ Is_Reachable_in(k1, p, Void) and
    Is_Reachable_in(k2, q, Void) and k2 ≤ k1 ) and
    ( ∀r: Location, if Is_Reachable(p, r) and
  Is_Reachable(q, r) then r = Void );
  ensures Is_Reachable(p, Void);
Procedure
  Var r: Position;
  Var s: Position;
  Relocate(r, p);
  While (not At_Void(q))
    maintaining Is_Reachable(p, Void);
  do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
  end;
end Splice;
```

Listing 3. Traditional implementation of splice operation

The next operation we illustrate is the splice operation, which was implemented in [4]. All of the statements here potentially modify the accessibility of the system.

The splice code implemented in Listing 4 is shorter than that in Listing 3. It uses only one local variable, *r*, which must advance twice at the end of each iteration so that the next element in *q*'s list is inserted into the appropriate position in *p*'s list. It is illustrated graphically in Figure 8. It is not clear whether the reasoning would be any simpler in this case. The invariant that needs to be proved is that *Void* is reachable from *p*, which essentially means that there are no cycles. However, both operations, Swap_Links and Swap_Link_Pos, may introduce cycles, as we have seen above.

```
Operation Splice(preserves p: Position; clears q: Position);
  updates Target;
  requires …
  ensures …
Procedure
  Var r: Position;
  Relocate(r, p);
  While (not At_Void(q))
    maintaining Is_Reachable(p, Void);
  do
    Swap_Links(r, NEXT, q, NEXT);
    Swap_Link_Pos(r, NEXT, q);
    Follow_Link(r);     // !!!
    Follow_Link(r);     // !!!
  end;
end Splice;
```

Listing 4. Implementation of splice operation using accessibility-preserving operations

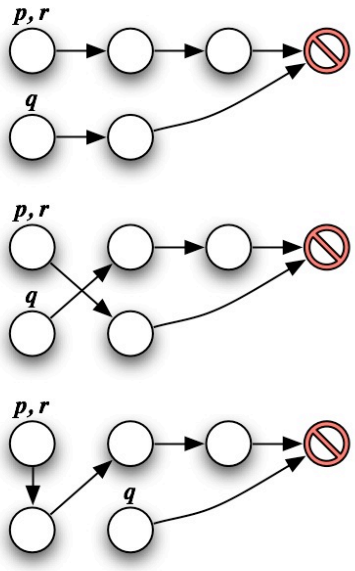


Fig. 8. Animation of the first part of the first iteration of the loop in the splice operation implemented with accessibility preserving operations

C. Observations about List-Like Code

Assume that a system of linked locations has only one link per location. If we define the location list of p (Loc_List(p)) as the longest acyclic string of locations that starts with p, we can say the following about a system with location lists p and q. If Loc_List(p) and Loc_List(q) do not share any locations other than *Void*, then invoking the accessibility preserving operations with p (or a member of p's list) and q (or a member of q's list) will not introduce any cycles and will not introduce any shared variables other than *Void*. Furthermore, invoking the operation Follow_Link using a position variable residing in position p's list will not change the locations accessible from p – as long as the operation is not invoked on position variable p itself. It may be possible to use these facts to simplify some part of the reasoning about both the Splice and the Insert operations.

D. Schorr-Waite Algorithm

The final example is the Schorr-Waite algorithm, which will appear in [6]. This algorithm is used to mark locations for a traditional mark and sweep garbage collector. Locations that are accessible from a root location are marked so that the garbage collector will *not* reclaim them. The implementation in Listing 5 is similar to the implementation found in [1]. The first conjunct of the loop invariant asserts that x is originally reachable from the root location if and only if it is currently reachable from either t or p – two of the three local variables used by this algorithm. When we originally planned on using this implementation in [6], we found this first conjunct the most difficult to prove – though it should be noted that we were using informal (math-style) proofs.

```
Procedure Mark_Accessible_From(preserves root: Position);
    Var t, p, q: Position;
    Relocate(t, root);
    While not Is_At_Void(p) or ( not Is_At_Void(t) and not
Is_Marked(t) )
        maintaining
∀x: Location,
x #is_reachable_from root iff x is_reachable_from t or x
is_reachable_from p
If not x #is_reachable_from root then Is_Marked(x) =
#Is_Marked(x)
If ⟨x⟩ is_substring_of Loc_Stack(p) then Is_Marked(x)
If x is_reachable_from root and not Is_Marked(x) then
x is_unmarked_reachable_from t or ∃y: Location such that
⟨y⟩ is_substring_of Loc_Stack(p) and
x is_unmarked_reachable_from Target(y, RIGHT)
If not ⟨x⟩ is_substring_of Loc_Stack(p) then
∀k: [LEFT, RIGHT], Target(x, k) = #Target(x, k)
Stack_OK(t, Loc_Stack(p));
        decreasing
3 * Unmarked_Loc_Count + Unset_CB_Count +
| Loc_Stack(root) |;

    do
        If Is_At_Void(t) or Is_Marked(t) then
            If CB_Is_Set(p) then
                Relocate(q, t);
                Relocate(t, p);
                Follow_Link(p, RIGHT);
                Redirect_Link(t, RIGHT, q);
            else
                Relocate(q, t);
                Relocate_to_Target(t, p, RIGHT);
                Redirect_to_Target(p, RIGHT, p, LEFT);
                Redirect(p, left, q);
                Set_Control(p);
            end;
        else
            Relocate(q, p);
            Relocate(p, t);
            Follow_Link(t, LEFT);
            Redirect_Link(p, LEFT, q);
            Mark_Location(p);
            Unset_Control(p);
        end;
    end;
end Mark_Accessible_From;
```

Listing 5. Traditional implementation of Schorr-Waite algorithm

```
Procedure Mark_Accessible_From(preserves root: Position);
    Var t, p: Position;
    Relocate(t, root);
    While not Is_At_Void(p) or ( not Is_At_Void(t) and not
Is_Marked(t) )
        maintaining …
        decreasing …
    do
        If Is_At_Void(t) or Is_Marked(t) then
            If Control_Is_Set(p) then
                Swap_Link_Pos(p, RIGHT, t);
                p :=: t;
            else
                Swap_Loc_Links(p, LEFT, RIGHT);
                Swap_Link_Pos(p, LEFT, t);
                Set_Control(p);
            end;
        else
            p :=: t;
            Swap_Link_Pos(p, LEFT, t);
            Mark_Location(p);
            Unset_Control(p);
        end;
    end;
end Mark_Accessible_From;
```

Listing 6. Implementation of Schorr-Waite algorithm using accessibility-preserving operations

Listing 6 shows an implementation of Schorr-Waite using the accessibility-preserving operations. The code is significantly shorter than that in Listing 5, and it does not require an additional local variable q. In addition, the first conjunct in the invariant becomes fairly easy to prove. It amounts to showing that if x is accessible (or inaccessible) from p or t at the begin-

ning of an iteration, it is still accessible (or inaccessible) from *p* or *t* at the end of that iteration.

IV. Conclusions

The operations Relocate, Follow_Link, and Redirect_Link can all change the accessibility of the system, while Swap_Link_Loc, Swap_Links, and swapping positions do not change the accessibility of the system. However, Redirect_Link, Swap_Links, and Swap_Link_Loc can all add cycles to a system where none existed before. Sometimes code size can be reduced and extra pointer variables can be avoided by implementing operations with the accessibility preserving operations.

Informally, accessibility preserving operations can simplify reasoning about accessibility and reachability. It seems reasonable to think that these operations could be beneficial in helping to *automatically* prove assertions for lightweight specifications involving accessibility. However, several open questions remain. Can we build a verifier that leverages the simplified informal reasoning about pointer operations? And what would such a verifier look like? In particular, would reachability and accessibility assertions be handled differently than other assertions? The Resolve community is beginning to focus on many of the issues involved in automated verification. In light of this, we believe these accessibility-preserving operations and their ability to simplify certain proofs deserve more attention.

References

[1] R. Bornat, "Proving Pointer Programs in Hoare Logic," in Mathematics of Program Construction, LNCS 1837, 2000.

[2] G. Kulczycki, M. Sitaraman, W. F. Ogden, J. E. Hollingsworth, "Component Technology for Pointers: Why and How," Technical Report RSRG-03-03, Clemson University, 2003. http://www.cs.clemson.edu/~resolve/reports/RSRG-03-03.pdf

[3] G. Kulczycki, M. Sitaraman, B. W. Weide, A. Rountev, "A Specification-Based Approach to Reasoning about Pointers," in Proceedings ESEC/FSE SAVCBS '05 Workshop, ACM Software Engineering Notes, Vol. 31, No. 2, 2005.

[4] G. Kulczycki, M. Sitaraman, H. Keown, B. W. Weide, "Abstracting Pointers for a Verifying Compiler," in Proceedings 31st Annual Software Engineering Workshop, Baltimore, MD. March, 2007.

[5] A. Singh, A Component-Based Approach to Reasoning about Pointer Programs, Master's Thesis, Virginia Tech. 2007. [to appear]

Duration Analysis of Finalization of Objects

Nighat Yasmin and Murali Sitaraman

***Abstract*—This paper examines the issues involved in estimating the time to finalize local objects within procedures, using performance profiles. It explains the need to strengthen specifications of operations and internal assertions, such as loop invariants, for effective performance analysis.**

***Index Terms*—Software Performance, Specification.**

I. Introduction

A formal specification of a software component can be implemented in multiple ways. Different implementations provide important space/time trade-offs to clients. Since clients should not be burdened with the internal details of implementations, developers must supply abstract, performance profiles for their implementations, so that clients could choose ones that best fit their performance needs and analyze the performance of their code. An introduction to performance profiles is given in [1]. This paper considers issues in writing and using duration expressions to finalize objects.

II. The problem

The execution time for a procedure depends on the time it takes to finalize its locally-declared objects, and the time to finalize these objects depends on their values. Given that typically we are not interested in values of objects that are just about to be finalized, what are the ramifications of accounting for finalization? This is the problem addressed in this paper.

```
Enhancement Flipping_Capability for Stack_Template;
    Operation Flip( updates S: Stack );
        ensures S = #S^Rev;
end Flipping_Capability;
```

Fig 1: Specification of a Stack reversal enhancement

Manuscript received May 22, 2007. This research is funded in part by a grant from the National Science Research Foundation (CCR-0113181) and a grant from the National Aeronautical and Space Administration through SC Space Grant Consortium.

Nighat Yasmin is a graduate student in the department of Computer & Information Science at The University of Mississippi University, MS 38677 USA (1-864-656-5756, ynighat@olemiss.edu).

Murali Sitaraman, is with department of Computer Science at Clemson University, Clemson, SC 29634 USA. (e-mail: murali@cs.clemson.edu).

```
Realization Obvious_F_C_Realiz for Flipping_Capability
                                        of Stack_Template;
    Procedure Flip( updates S: Stack );
        Var Next_Entry: Entry;
        Var S_Flipped: Stack;

        While ( Depth( #S ) /= 0 )
            changing S, Next_Entry, S_Flipped;
            maintaining #S = S_Flipped^Rev ∘ S;
            decreasing |#S|;
         do
            Pop( Next_Entry, S );
            Push( Next_Entry, S_Flipped );
        end;
        S :=: S_Flipped;
    end Flip;

end Obvious_F_C_Realiz;
```

Fig 2: A Stack reversal enhancement realization

III. Position

Objects are finalized when they exit their scope, and the time for their finalization must be accounted in performance analysis. This accounting affects assertions that are typically only concerned with functionality, such as loop invariants, but it is entirely tractable and uniform.

IV. Justification

To make the problem concrete, we discuss a specific example. Consider the functional specification and realization (implementation) of a stack reversal enhancement given in Fig. 1 and Fig. 2, respectively. The execution time of the Flip procedure depends on the time to finalize local objects Next_Entry and S_Flipped. How do we account for this finalization time? This is the focus of the rest of this paper.

Fig. 3 shows the skeleton of a formal functional specification of the Stack_Template, on which the enhancement is based. For further details, see [1]. The bounded Stack_Template can be realized using an array. The realizations may be either space conscious or time efficient. In a space-conscious realization, every entry in the array that is not part of the conceptual stack contains only the initial value of the *Entry*, as illustrated in Fig. 4. In other words, the implementation follows the space conscious convention of maintaining initial values for the part of the array that does not contribute to the conceptual stack. In a time-efficient realization, the entries in the array that are not part of the conceptual stack can contain arbitrary values as shown in Fig.

5.

```
Concept Stack_Template(type Entry; evaluates
                                    Max_Depth: Integer);
   uses Std_Integer_Fac, String_Theory;
   requires Max_Depth > 0;

   Type Family Stack is modeled by Str(Entry);
      exemplar S;
      constraint |S| <= Max_Depth;
      initialization ensures S = empty_string;

   Operation Push( alters E: Entry; updates S: Stack );
      requires |S| < Max_Depth;
      ensures  S = <#E> o #S;

   Operation Pop( replaces R: Entry; updates S: Stack );
      requires |S| > 0;
      ensures #S = <R> o S;
    .
    .
    .
end Stack_Template;
```

Fig 3: A skeleton of the Stack_Template specification

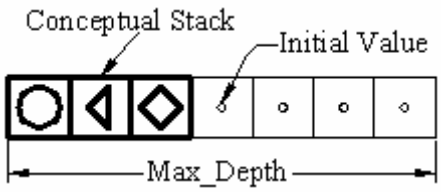


Fig 4: Internal array (space-conscious realization)

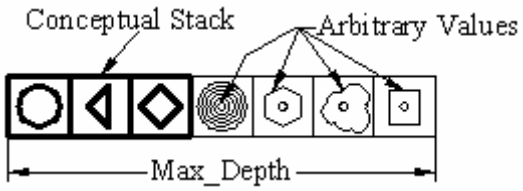


Fig 5: Internal array (time-efficient realization)

Performance analysis of the Flip procedure will depend on the profiles of the implementations of the Stack_Template. As a prelude to considering the time to finalize the local objects in the Flip procedure in Fig. 2, therefore, we first study the time to finalize Stack objects.

In the performance profiles, *Entry.**I_Dur*** represents the time needed to initialize an *Entry*; *Entry.**F_Dur(E)*** represents the time to finalize an *Entry E*; *Entry.**F_IV_Dur*** is the duration to finalize an *Entry* that has an initial value; and 'a·b' represents the product of a and b. If Max_Depth is the total depth of a stack and |#S| is the depth of the incoming conceptual stack then the Max_Depth - |#S| represents the entries in the array that are not part of the conceptual stack.

1. Space-Conscious Implementations

A profile for the space-conscious class of implementations is given in [1] and we summarize it here.

1.1. Duration for Stack Finalization

In performance profile, *SSC* for the space-conscious class of implementations shown in Fig. 6, the duration for finalization for a Stack S is given as:

duration SSC_{F1} + Cnts_Dur(#S) +
(SSC_{F2} + Entry.**F_IV_Dur**)·(Max_Depth - |#S|)

The duration expression has two parts. Part 1 is the time to finalize the entries in the array that are part of the conceptual stack. This part is given as the summation of SSC_{F1} (implementation dependent constant) and Cnts_Dur(#S). The term Cnts_Dur(#S) is the sum of the durations to finalize every entry in the conceptual stack. Part 2 is the time to finalize the entries in the array that are *not* part of the conceptual stack. In the space-conscious implementation, the second part of the array contains only the initial valued entries. Therefore, the duration for finalization of this part is captured by

(SSC_{F2} + Entry.**F_IV_Dur**)·(Max_Depth - |#S|)

where, SSC_{F2} is an implementation dependent constant.

```
Profile SSC short_for Space_Conscious
                              for Stack_Template;
   Defines SSC_F1, SSC_F2, SSC_Po, SSC_Pu, ...: ℝ^≥0;
    .
    .
    .
   defn Cnts_Dur( α: Str(Entry) ): ℝ^≥0 =
       ( ∑ Occurs_Ct(E,α)·Entry.F_Dur(E) );
         E: Entry
   finalization
         duration SSC_F1 + Cnts_Dur(#S) + (SSC_F2 +
              Entry.F_IV_Dur)·(Max_Depth - |#S|);

   Oper Push( clears E: Entry; updates S: Stack );
         duration SSC_Pu;

   Oper Pop( replaces R: Entry; updates S: Stack );
         duration SSC_Po + Entry.I_Dur +
                                Entry.F_Dur(#R);
    .
    .
    .
end SSC;
```

Fig 6: Performance profile of space-conscious realizations

1.2. Finalization of Local Objects in the Flip Procedure

In analyzing the duration for Flip procedure, we note that the *Stack* S_Flipped is empty at the end of the procedure. So the time to finalize it is straightforward: Simply substitute Λ for *Stack* S in the finalization duration expression above. This gives us the expression SSC_{F1} + Cnts_Dur(Λ) + (SSC_{F2} + Entry.**F_IV_Dur**)·(Max_Depth - |Λ|) and it simplifies to SSC_{F1} + (SSC_{F2} + Entry.**F_IV_Dur**)·(Max_Depth).

How long does it take to finalize Next_Entry? This depends on the values of Next_Entry that result after the call to *Push*. Notice now that the SSC profile specification strengthens the specification of the *Push* operation and asserts that it *clears* the pushed Entry. Hence, at the end of each loop iteration Next_Entry is reinitialized. Also, before the *while* loop, Next_Entry has an initial *Entry* value. So we can strengthen the loop invariant as shown below in Fig. 7.

Given the invariant at the end of the loop, and hence, at the

end of the Flip procedure Next_Entry will have an initial value. So the time to finalize it is just Entry.**F_IV_Dur**. This example illustrates the general impact of performance analysis on assertions, such as invariants.

```
Realization Obvious_F_C_Realiz for Flipping_Capability
      with_profile SSCF of Stack_Template with_profile SSC;
   .
   .
   .
   Procedure Flip( updates S: Stack );
      Var Next_Entry: Entry;
      Var S_Flipped: Stack;

      While ( Depth( #S ) /= 0 )
         changing S, S_Flipped, Next_Entry;
         maintaining #S = S_Flipped^Rev ∘ S and
                  Entry.Is_Init(Next_Entry);
         decreasing …
         elapsed_time …
      do
         Pop( Next_Entry, S );
         Push( Next_Entry, S_Flipped );
      end;
      S :=: S_Flipped;
   end Flip;
end Obvious_F_C_Realiz;
```

Fig 7: Stack reversal enhancement realization based on space-conscious realization for the Stack Template

2. *Fast-Array Implementation*

The analysis of the class of "non-initializing" implementations that allow arbitrary entries to be present in the array is slightly more complex, and it is the topic of this section. The implementation is shown in Appendix A.

2.1. Finalization Operation

A simple performance profile, *SFC*, for the time-efficient (or Fast_Clear) implementation is shown in Fig. 8. In this profile, the duration for the finalization is given as:

duration SFC_{F1} + Cnts_Dur(#S) +
(SFC_{F2} + Entry.**F_Dur**(Max_E))·(Max_Depth - |#S|)

This duration for finalization can also be divided into two parts. Part 1 is the time to finalize the entries in the array that are part of the conceptual stack, and it is the same for both the space- and time-conscious array implementations. Part 2 of the finalization duration is described with the help of a supplemental model.

The supplemental model introduces an *Entry* type variable, *Max_E*, to record an Entry that is "left behind" and that takes the maximum time to finalize out of all entries not contributing to the conceptual. Initially, all array entries are initialized, so *Max_E* is given an initial *Entry* value in the initialization ensures clause. Using *Max_E*, the second part of the finalization duration is given by:

(SFC_{F2} + Entry.**F_Dur**(Max_E))·(Max_Depth - |#S.**ipso**|)

where SSC_{F2} is an implementation dependent constant. Here, S.**ipso** is a generic notation to denote the conceptual model of a stack, a string.

2.2. Push Operation

In the profile of time-efficient implementation, the duration for the *Push* operation is given by the implementation dependent constant, SFC_{pu}. However, there is an extra **ensures** clause that is useful in analyzing the performance of client code. This clause guarantees that the finalization duration of the outgoing value of the entry E is no more than the time to finalize *Max_E*.

```
Profile SFC short_for Fast_Clear for Stack_Templat
   Defines SFC_F1, SFC_F2, SFC_Po, SFC_Pu, …: ℝ^≥0;
      .
      .
      .
      Supplement Max_E: Entry;
      Initialization
         ensures Entry.Is_Initial(Max_E);
         duration SFC_I1 + (SFC_I2 + Entry.I_Dur)·Max_Depth;
      defn Cnts_Dur( α: Str(Entry) ): ℝ^≥0 =
         ( ∑ Occurs_Ct(E,α)·Entry.F_Dur(E) );
           E: Entry
      finalization
         duration SFC_F1 + Cnts_Dur(#S) +
            (SFC_F2 + Entry.F_Dur(S.Max_E))·
                        (Max_Depth - |#S.ipso|);

      Oper Push( alters E: Entry; updates S: Stack );
         ensures Entry.F_Dur(E) ≤ Entry.F_Dur(S.Max_E);
         duration SFC_Pu;

   Oper Pop( replaces R: Entry; updates S: Stack );
         updates Max_E;
         ensures Max_E = #R if Entry.F_Dur(#R) >
                              Entry.F_Dur(#Max_E) and
                     Max_E = #Max_E otherwise;
         duration SFC_Po;
   .
   .
end SFC;
```

Fig 8: Performance profile for realizations that don't clean up

2.3. Pop Operation

In the profile of the time-efficient implementation, the duration for the *Pop* operation is given by the implementation dependent constant, SFC_{po}. However, the additional **updates** clause indicates that the outgoing value of *Max_E* may be different from its incoming value. To see why refer to the code for Pop in the implementation in the Appendix that merely swaps in the arbitrary Entry supplied as the parameter with the top Stack Entry. If this new Entry takes much longer to finalize, then the supplement *Max_E* has to be updated. Specifically, the **ensures** clause guarantees that if the finalization duration of the incoming entry (#R) is greater than the finalization duration of the current *Max_E* then *Max_E* is updated to *#R*.

2.4. Finalization of Local Objects in the Flip Procedure

How much time does it take to finalize Next_Entry and S_Flipped? To answer this question, we need to establish suitable loop invariants that will let us make assertions about their values at the end of the code. A realization of the *Flipping_Capability* is shown in Fig. 9.

First we consider the invariant conjunct (i),

Entry.**F_Dur**(S_Flipped.Max_E) <= Entry.**F_IV_Dur**. Before the *while* loop, *S_Flipped.Max_E*, has initial *Entry* value. The only operation that's called on S_Flipped is *Push*, and *Push* does not affect Max_E. So the most time, it could take to finalize S_Flipped.Max_E is the time to finalize an initial Entry, i.e., Entry.**F_IV_Dur**. So the invariant conjunct follows.

Now consider the conjunct (ii): Entry.**F_Dur**(Next_Entry) <= Entry.**F_IV_Dur**. Before the *while* loop, Next_Entry has initial *Entry* value, so the invariant holds at the beginning. That it holds in subsequent iterations follows from (1) specification of *Push* in the SFC profile, which guarantees that Entry.**F_Dur**(Next_Entry) ≤ Entry.**F_Dur**(S_Flipped.Max_E) and (2) invariant conjunct (i).

```
Realization Obvious_F_C_Realiz for Flipping_Capability
      with_profile SSCF of Stack_Template with_profile SFC;

   Definition SFCF_F1: ℝ^≥0 =
                ( Dur_Call(1) + ( SFC_Dp + ICA_/= ) + RUC_:=: );
   Definition SFCF_F2: ℝ^≥0 = ( SFC_Dp + ICA_/= + SFC_Po + SFC_Pu );

   Procedure Flip( updates S: Stack );
      Var Next_Entry: Entry;
      Var S_Flipped: Stack;

      While ( Depth( #S ) /= 0 )
         changing S, Next_Entry, S_Flipped;
         maintaining #S.ipso = S_Flipped.ipso^Rev ∘ S.ipso and
            Entry.F_Dur(S_Flipped.Max_E) <= Entry.F_IV_Dur
            and Entry.F_Dur(Next_Entry) <= Entry.F_IV_Dur
            and S.Max_E = #S.Max_E;
         decreasing |#S.ipso|;
         elapsed_time …
       do
         Pop( Next_Entry, S );
         Push( Next_Entry, S_Flipped );
      end;
      S :=: S_Flipped;
   end Flip;

end Obvious_F_C_Realiz;
```

Fig 9: Alternative Loop invariants for Flip

Conjunct (iii): S.Max_E = #S.Max_E follows directly from conjuct (ii) and the fact the **ensures** clause for *Pop* (the onlt operation called on Stack S) .

Given the invariants, now we are ready to analyze the time to finalize Next_Entry and S_Flipped. From the invariant, it is easy to see that the time to finalize Next_Entry is dominated by Entry.**F_IV_Dur**. S_Flipped and S are swapped after the loop so S_Flipped.Max_E becomes S.Max_E which remains the same as #S.Max_E, according to the invariant. So the time to finalize S_Flipped is simply Entry.**F_Dur**(#S.Max_E) * Max_Depth.

Intuitively, the analysis is easy to understand. Since the incoming Stack is swapped in the flipping code, the garbage in S needs to be cleaned up at the end of the procedure. At the same time, we can guarantee that S is clean at the end of the code,, indirectly. Therefore, we add an additional ensure clause to Flip profile to guarantee that Entry.**F_Dur**(S.Max_E) <= Entry.**F_IV_Dur**. This follows from the invariant Entry.**F_Dur**(S_Flipped.Max_E) <= Entry.**F_IV_Dur** before the swap statement. These observations lead us to the profile for Flipping_Capability shown in the next Fig 10.

```
Profile SFCF short_for Fast_Clear_Stack_Flip for
            Flipping_Capability for Stack_Template
                        with_profile SFC;

      Defines SFCF_F1, SFCF_F2: ℝ^≥0;

      Operation Flip( updates S: Stack );
         ensures Entry.F_Dur(S.Max_E) <=
                                 Entry.F_IV_Dur;
         duration SFCF_F1 + Entry.I_Dur +
            Stack.I_Dur + Entry.F_IV_Dur +
            (SFCF_F2)·|#S.ipso| +
            (Entry.F_Dur(#S.Max_E))·Max_Depth;

end SSCF;
```

Fig 10: Performance profile for the realization shown in Fig. 9

V. Discussion

There are several questions for discussion. In general, is it an appropriate trade-off to use the simplified supplemental model of remembering only one entry that takes most time to finalize? Given the cost of finalization, should the Stack used for flipping be a global variable in the enhancement? What are the ramifications?

VI. Acknowledgments

We thank Joan Krone, Bill Ogden, and Bruce Weide for discussions on topics contained in this paper

References

[1] J. Krone, W. F. Ogden, and M. Sitaraman. Performance analysis based upon complete Profiles. *Fifth International Workshop on Specification and verification of Component-Based Systems (SAVCBS 2006)*. Portland, Oregon, USA, November 2006.

Appendix A

```
Realization Array_Realiz for Stack_Template;
   Type Stack is represented by Record
         Contents: Array 1..Max_Depth of Entry;
         Top: Integer;
      end;
      convention 0 <= S.Top <= Max_Depth;
      correspondence
         Conc.S = Rev(Concatenation i: Integer
            where 1 <= i <= S.Top, <S.Contents(i)>);

   Procedure Push(alters E: Entry; updates S: Stack);
      S.Top := S.Top + 1;
      E :=: S.Contents(S.Top);
   end Push;

   Procedure Pop(replaces R: Entry; updates S: Stack);
      R :=: S.Contents(S.Top);
      S.Top := S.Top - 1;
   end Pop;
    …
end Array_Realiz;
```

Building Skills with Logic and Proof Argument

Bruce W. Weide, Brandon Mintern, and Harvey M. Friedman

Practice is the best of all instructors.
— Publilius Syrus (42 BC)

Position Statement

For nearly every major topic in an elementary science or engineering course, student assignments include many practice problems, all of a similar nature. As the saying goes, "practice makes perfect" because it helps students hone their skills in solving problems of a certain kind. Can software automatically produce good practice problems? Can software offering deep, serious interaction give students a qualitatively different and more effective experience than traditional homework or rote drill-and-practice?

Our first position is that these questions can be answered "yes" for an important aspect of mathematics and computer science (CS) education: skill with the formal notation of predicate logic and rigorous proof argument in mathematical domains of interest in CS—*as opposed to* skill in formal symbolic proof without the same mathematical theories as the underlying context, which might be taught in a philosophy course. The proposed approach for generating good practice problems and honing student skills in this area is to leverage powerful software theorem-proving assistants that have been developed for research purposes and are in the public domain. We are planning to build a suite of courseware called ***Syrus*** that adapts such existing tools to mediate student interaction scenarios that are far more flexible than can be achieved with canned practice problems from a textbook, with standard drill-and-practice courseware, or with fully scripted dialogs. A key technical feature of *Syrus* will be that the student's reasonably open-ended answers to practice problems will be checked by sophisticated back-end mathematical software in real-time during the interaction.

Our second position it that future CS graduates will find such skill increasingly important in order to reason soundly about the dynamic behavior of the software systems they will design and implement. Clearly, this ultimately will be true when the Resolve vision is realized. But it is becoming true anyway, even now. As more and more commercial software ventures follow the lead of the pioneers in using formal software specification technologies in actual practice—including large firms such as Microsoft [19] and smaller firms such as Praxis HIS [22, 21]—CS graduates will find employers increasingly interested in employees with a firm foundation in predicate logic, which is the basis for modern software specification languages that are coming into wider use in industry. Graduates also will be expected to know how to reason about statements written in such notations. To believe that this is not inevitable is to deny history, to ignore the relentless progression of formalization and sophistication of the mathematical models used *in practice* in all other hard science and engineering disciplines. To date, except for the obvious and direct connection to practical database design and querying, CS student knowledge of predicate logic and proof has been considered by many to be a matter of showing "mathematical maturity". Maturity will not be enough in the future; competence will be required of all CS graduates, and expertise of some. Achieving this will require practice making proof arguments that most students do not get today because it simply is not practical without computer assistance.

Bruce W. Weide and Brandon Mintern are with the Department of Computer Science and Engineering at The Ohio State University (weide@cse.ohio-state.edu, mintern@cse.ohio-state.edu).

Harvey M. Friedman is with the Departments of Mathematics, Computer Science and Engineering, and Philosophy at The Ohio State University (friedman@math.ohio-state.edu).

Logic and Proof Argument in Computer Science

An important learning outcome of a discrete mathematics course for CS is that students should be able to make rigorous proof arguments involving the validity of logical statements in predicate calculus. Experience over many years and many institutions has shown that it is notoriously difficult to get undergraduate students to think with the required mathematical rigor. The point in requiring this material for CS students has been evident since *Curriculum 68* [15]. The important new rationale, simply put, is that skill with predicate logic and proof argument gives software engineers control over the correctness of their programs: they need to be able to formulate carefully what correctness means and to be able to reason soundly about correctness. It requires a certain level of mathematical sophistication to understand formal specifications of software that use formal logic notation with predicate calculus [17, 27]. Regarding proof, however, CS majors do not necessarily need skill in performing formal proofs via symbolic manipulation—as emphasized in [12] and in symbolic logic courses taught in philosophy departments—but rather skill in making rigorous proof arguments. CS students do not require deep knowledge of logic theory and full-blown proof procedures, but rather the skill to understand and reason about and with formal logical sentences over theories of interest within CS. They need to be able to do this with sufficient "mathematical rigor" and sound intuition.

Innovations to Facilitate Sufficient Practice

The primary difficulty faced by instructors in a course covering logic and proof argument for CS is that the relevant

kind of mathematical rigor is reasonably subtle. For the overwhelming majority of students, a considerable amount of practice is required to achieve the skill level sought by faculty. Yet by a very large margin, the level of interaction needed in such practice exercises is just not practical without computer technology. A typical student learning logic and proof does a homework problem on paper and gets feedback about it days later, after tedious grading by a human. Even one round of revisions for re-grading—such as the student might experience in an English composition class—is certainly the exception in discrete mathematics courses today. However, several rounds of interaction per practice problem might be needed for the student to appreciate the subtle nature of the mathematical arguments involved in determining the truth value of a logical sentence in predicate calculus, and in explaining thoroughly why his/her answer is correct. Then, there is the issue of how to get such practice on many, many problems.

It is instructive to contrast the practice-feedback-rework situation in a discrete mathematics course with that in a typical programming course. A student learning to program gets virtually instant feedback on the syntactic correctness of a purported solution. In fact, rapid feedback about correctness of the solution goes far beyond mere well-formedness. The student might experience many rounds of interaction with the computer's execution of a program before that program is deemed correct. Usually, this interaction is not guided in any systematic way, except to the extent that the student has been taught to use specific techniques for testing and debugging [6, 7]. The computer does not ask the student questions or lead the student step-by-step through a process that an expert might use to debug and correct a program. Yet it does provide plenty of practice using short-cycle feedback and associated opportunities for learning-by-doing.

Syrus will provide practice that is more targeted and adaptive to student needs than is possible with the kind of unstructured and completely open-ended feedback that comes with compiling and executing programs in a programming course. The student will not be given a fixed set of answer choices for each problem, and will not merely be told that answers are right or wrong. Instead, he/she will have unprecedented flexibility to answer questions posed by *Syrus*: questions based in part on answers the student has given so far. Where will practice problems come from? *Syrus* will generate them systematically, off-line, from problem templates. It will automatically filter these sentences to acquire an inventory of pre-solved practice problems, by passing candidates to the back-end proof assistant and saving the solved ones with their answers. The interactive part of *Syrus* will randomly select practice problems with various instructor-specified characteristics from this pre-computed inventory.

In short, *Syrus* will allow a student to practice skills in making proof arguments about predicate logic sentences on a virtually endless supply of example problems meeting instructor-specified criteria, all within the context of mathematical theories that CS students should learn, and all with a serious level of flexible real-time interaction. Moreover, *Syrus* will be freely available to anyone, anywhere, using only a web browser. A demo of a prototype version is available at http://syrus.cse.ohio-state.edu.

Technical Issues

The general learning outcome targeted by *Syrus* is that students should be able to understand and reason rigorously about/with logical statements in predicate calculus over various mathematical theories that are of interest in CS. The examples handled by the prototype and discussed here involve Presburger arithmetic, where variables denote integers, and expressions may include integer constants (e.g., 0, –13), addition (+) and subtraction (–) as well as multiplication by constants (e.g., $3*x$), and relational operators (e.g., $x+3 = y$, $x \leq 18$). Students can be expected to know this underlying mathematical theory with no additional instruction. The educational focus for these examples, at least, is therefore on the predicate logic aspects, though it is evident that "domain knowledge" is also important in proof arguments.

An important technical property of Presburger arithmetic is that there is a decision procedure for it. Indeed, for the sorts of problems that *Syrus* will present to students, the prototype version shows that the decision procedure [5] implemented in John Harrison's "theorem proving examples" code [13]—though not the best available algorithm [10] and according to Harrison himself "not intended to be efficient"—is nonetheless acceptably fast. It can determine the truth value of a typical sentence in a fraction of a second, leading to comfortable interactions with a human.

There are, of course, theories other than Presburger arithmetic (or integers in general) that are of interest and importance in CS. Can *Syrus* be extended to handle these theories, too? It should be possible, where there are decision procedures for the fragments of mathematics involved.

Finite sets are important in CS because many mathematical models used in software specifications and databases involve them. It is well known that the sets of first order sentences true in any infinite atomic Boolean algebra are the same, and decidable. From this, it follows that for any infinite set X, the two-sorted system X, $\wp(X)$, with equality on both sorts, inclusion on $\wp(X)$, and union, intersection, and complement on $\wp(X)$, is decidable. The decision procedure is independent of the choice of infinite set X. Furthermore, it is known that if we add a finite cardinality function and the predicate "is finite" to this system, then we still get decidability—with the procedure independent of the choice of X [16]. Given the importance of generic mathematical sets in software specifications and elsewhere in CS, this decision procedure or a similar one should have substantial educational value as a component of *Syrus*.

Another theory of interest is that of finite strings $\Sigma^*(X)$ over an infinite set X. It is important in CS because many software specifications involve mathematical models of this sort and because it arises in the study of automata and language theory (though here, X is normally finite). The operators include equality on X and $\Sigma^*(X)$, string construction from X,

concatenation of strings, and a length function. With a small number of quantifiers, it is not clear whether this is decidable.

Related Work

The educational purpose of *Syrus* will be to take students who are novices at solving logic problems of certain kinds, and to help them develop more expert-like skills. The way to become an expert is through effortful practice [9]. The cognitive apprenticeship model [4] basically says that students need to be shown how to do something and be given the chance to do it with scaffolding in place, and that gradually the scaffolding can be taken away. *Syrus* will provide such scaffolding, tailored to each student in real-time interactions, for as many problems as the student wishes to try.

When trying to develop problem-solving skills, one needs to put the student in a situation where novice-like skills will not be successful, yet more expert-like skills will result in a reasonable chance of success. *Syrus* will accomplish this by constructing large inventories of problems of varying degrees of difficulty, and by requiring the student to carefully justify conclusions through rigorous proof arguments having the structure the instructor wants the student to learn, as well as by quickly detecting mistakes and then focusing interactions on their mitigation or correction.

Regarding back-end proof assistants [29], the *Syrus* prototype uses a set of OCaml modules [13] that are related to algorithms used in HOL Light [14]. If these modules prove inadequate for some reason in a production version of *Syrus*, HOL Light or Isabelle [2] are likely candidates to be substituted.

In terms of related educational software, the works of Suppes, *et al*., at Stanford, and Sieg, *et al*., at CMU, stand out as the best developed and the most closely related to *Syrus*. They are also among the very few such tools to have been evaluated with student subjects.

The CMU group observed [25] that resolution theorem provers gave little help to students in thinking about how to prove something, and hence set out to develop a proof search system based on natural deduction: the CMU Proof Tutor. Experiments with students using this system for structuring proofs in sentential (propositional) logic [24] showed the educational value of "naturalness", i.e., maintaining a close connection between formal proof structures and the cognitive processes involved in making proof arguments. This connection is key to the thesis underlying *Syrus*. An important difference from the (original) CMU Proof Tutor, however, is that *Syrus* will focus on quantified formulas over mathematical theories of central importance in CS education. From examining the available materials describing the AProS project [26], it is not clear how a proposed new Proof Tutor, now under development, will deal with quantifiers or with underlying mathematical theories such as those that will provide the context for practice problems in *Syrus*. A web-accessible demo of the new tool's intended use in the "Logic and Proof" course of CMU's Open Learning Initiative [3] suggests that the emphasis remains on formal symbolic proofs in propositional logic using natural deduction. *Syrus* also will automatically generate practice problems; it is not clear whether the new Proof Tutor will do that.

The EPGY Theorem Proving Environment is used by gifted students in Stanford's Education Program for Gifted Youth. Its published evaluation [28] involved 170 students doing geometry proofs. This is a sophisticated and rather heavyweight proof environment that can be used across a variety of mathematical domains, with a pedagogical focus on proving interesting results in a variety of underlying mathematical theories (including geometry, linear algebra, multivariate calculus, and differential equations). The tool's purpose is to fill in gaps in student proofs—anything that can be completed by the proof assistant Otter [18] within 4-5 seconds. It attempts to supply automatically the below-the-radar details that normally would not appear in a proof in the mathematical literature, and hence to allow students to prove theorems as would be done in "standard mathematical practice". *Syrus* will differ in several important respects. It will provide students with a platform for repeated practice with logic and proof arguments on problems that involve underlying mathematics in theories of interest to CS students, so it can focus specifically on student understanding of quantified sentences and proof arguments involving them. *Syrus* sample problems will be generated automatically and will admit rather straightforward proofs arguments, rather than being selected from among classical and especially interesting theorems in the underlying theories with emphasis on proofs that professional mathematicians (or extremely talented students of mathematics) might produce. A demo of the EPGY tool [8] as used in the geometry course discussed in [28] quickly reveals an emphasis on formal symbolic proof, rather than on proof argument as in *Syrus*. Finally, *Syrus* will be freely available to anyone and will be usable from any computer, while the EPGY Theorem Proving Environment apparently is available only to students who pay for an EPGY course and runs only on Windows computers.

A well-known program with a somewhat related objective is Tarski's World [1]. It uses a blocks-world context to help students learn the semantic connections between symbolic first-order logic and "real world" situations. A tool with similar objectives is LOGTUTOR [23]. Other tangentially related tools that are intended to accompany specific textbooks used in symbolic logic courses are Logic Tutor [11], an on-line tutorial offering multiple choice questions; and LogicCoach9 [20], a more interactive tool that offers hundreds of sample problems of various kinds. There are dozens of other software tools supporting various aspects of instruction in logic, primarily intended to highlight formal symbolic logic and proof as taught in philosophy courses [30] and hence complementary to *Syrus*, which will provide underlying mathematical context important for CS students.

Discussion Topics for the Resolve Workshop

Our plan for the Resolve Workshop presentation is to demonstrate the *Syrus* prototype and then to explain briefly

how it works. Discussion will be focused on technical details of how we plan to elaborate a taxonomy of logical sentence types and associated proof argument techniques that we believe students need to learn—and that they can learn with sufficient practice mediated by *Syrus*.

REFERENCES

1. Barwise, J., and Etchemendy, J. Computers, visualization, and the nature of reasoning. In *The Digital Phoenix: How Computers are Changing Philosophy* (T.W. Bynum and J.H. Moor, eds.), 1992, 93–116.
2. University of Cambridge and Technische Universität München. *Isabelle*. http://www.cl.cam.ac.uk/research/hvg/Isabelle, viewed 5 May 2007.
3. Carnegie Mellon University. *Open Learning Initiative: Course Features*. http://www.cmu.edu/oli/features, viewed 5 May 2007.
4. Collins, A., Brown, J.S., and Newman, S.E. Cognitive apprenticeship: teaching the crafts of reading, writing, and arithmetic. In *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser* (L.B. Resnick, ed.), Erlbaum, Hillsdale, NJ, 1989, 453-494.
5. Cooper, D. C. Theorem-proving in arithmetic without multiplication. *Machine Intelligence 7*, 1972, 91-99.
6. Edwards, S.H. Using test-driven development in the classroom: providing students with concrete feedback on performance. In *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications (EISTA '03),* August 2003.
7. Edwards, S.H. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings 35th SIGCSE Technical Symposium on Computer Science Education*, ACM, 2004, 26-30.
8. EPGY (Education Program for Gifted Youth). *Sample Lecture: Proving Universal Statements*. http://epgy.stanford.edu/courses/math/M015/lecture.html, viewed 5 May 2007.
9. Ericsson, K.A., Krampe, R.T., and Tesch-Römer, C. The role of deliberate practice in the acquisition of expert performance. *Psychological Review 100,* (1993), 363-406.
10. Ganesh, V., Berezin, S., and Dill, D.L. Deciding Presburger arithmetic by model checking and comparisons with other methods. In *Formal Methods in Computer-Aided Design (FMCAD '02),* Springer-Verlag LNCS 2517, November 2002, 171-186.
11. Green, M.K. *Logic Tutor*. http://www2.wwnorton.com/college/phil/logic3, viewed 5 May 2007.
12. Gries, D., and Schneider, F.B. *A Logical Approach to Discrete Math*. Springer-Verlag, New York, 1993.
13. Harrison, J. *Theorem Proving Examples*. http://www.cl.cam.ac.uk/~jrh13/atp, viewed 5 May 2007.
14. Harrison, J. *The HOL Light Theorem Prover*. http://www.cl.cam.ac.uk/~jrh13/hol-light, viewed 5 May 2007.
15. Hemmendinger, D. The ACM and IEEE-CS guidelines for undergraduate CS education. *Communications of the ACM 50*, 5 (May 2007), 46-53.
16. Kuncak, V., Nguyen, H.H., and Rinard, M. *An Algorithm for Deciding BAPA: Boolean algebra with Presburger Arithmetic*. MIT-LCS-TR-958, July 2004, http://www.lcs.mit.edu/publications/specpub.php?id=1736.
17. Long, T.J., Weide, B.W., Bucci, P., Gibson, D.S., Hollingsworth, J.E., Sitaraman, M., and Edwards, S.H. Providing intellectual focus to CS1/CS2. In *Proceedings 29th SIGCSE Technical Symposium on Computer Science Education*, ACM, 1998, 252-256.
18. McCune, W. *Otter and Mace2*. http://www.cs.unm.edu/~mccune/otter, viewed 5 May 2007.
19. Microsoft Research. *Spec#*. http://research.microsoft.com/specsharp, viewed 5 May 2007.
20. Pole, N. *The Unofficial LogicCoach9 Page*. http://academic.csuohio.edu/polen, viewed 5 May 2007.
21. Praxis High Integrity Systems, Ltd. http://www.praxis-his.com, viewed 5 May 2007.
22. Ross, P.E. The exterminators. *IEEE Spectrum 42*, 9 (September 2005), 36-41.
23. Rowe, N.C. A computer tutor for logic semantics. In *Proceedings 29th Annual Frontiers in Education Conference,* Volume 2, IEEE, 1999, 12C3/1-12C3/6.
24. Scheines, R., and Sieg, W. Computer environments for proof construction. *Interactive Learning Environments 4*, 2 (1994), 159-169.
25. Sieg, W., and Scheines, R. Search for proofs (in sentential logic). In *Philosophy and the Computer* (L. Burkholder, ed.), 1992, 137-159.
26. Sieg, W. *AProS: Automated Proof Search*. http://www.phil.cmu.edu/projects/apros, viewed 5 May 2007.
27. Sitaraman, M., Long, T.J., Weide, B.W., Harner, J., and Wang, C. A formal approach to component-based software engineering: education and evaluation. In *Proceedings 2001 International Conference on Software Engineering,* IEEE, 2001, 601-609.
28. Sommer, R., and Nuckols, G. A proof environment for teaching mathematics. *Journal of Automated Reasoning 32*, 3 (February 2004), 227-258.
29. Talcott, C. *Database of Existing Mechanized Reasoning Systems*. http://www-formal.stanford.edu/clt/ARS/systems.html, viewed 5 May 2007.
30. van Ditmarsch, H. *Logic software and logic education*. http://www.cs.otago.ac.nz/staffpriv/hans/logiccoursesware.html, viewed 5 May 2007.

Combining Theory and Implementation: A Proposed Course on Distributed Computing

Sally K. Wahba and Andrew R. Dalton
School of Computing, Clemson University
Clemson, South Carolina
{sallyw, adalton}@cs.clemson.edu

***Abstract*— Universities typically offer graduate-level courses in computer science that focus either on implementing software systems or on the theoretical foundations of these systems. In such an environment students who enjoy programming select elective courses that focus on software implementation. Likewise, students who enjoy theoretical topics select elective courses that focus on foundational topics. The result is a set of graduating students who are either good programmers, but lack the theoretical foundations supporting the programming concepts, or good theoreticians who lack the ability to implement software systems well. We believe computer science graduates should be proficient at *both* theory and implementation. One of the graduate-level theory courses offered at Clemson University is *The Foundations of Distributed Computing* (CpSc 873). We outline a new syllabus based on this course where not only are the theoretical foundations taught, but also a concrete implementation of the algorithms developed.**

I. Introduction

Computer science is "the study of the theoretical foundations of information and computation and their implementation and application in computer systems" [5]. Accordingly, a good computer scientist needs to be proficient at both theory and implementation.

The *theory of computation* is defined as "the branch of computer science that deals with whether and how efficiently problems can be solved on a computer" [6]. In fact, Dean shows that theory is the branch of computer science that differentiates computer scientists from programmers as the former can better survive technology changes [3]. Unlike programmers, computer scientists have the sufficient tools to provide both a provably correct and efficient solution for a problem domain – computer scientists are problem solvers.

An *implementation* is defined as "a process following a well defined model that is understood and can be expressed in an algorithm, protocol, network topology, etc" [4]. Without such expression, theory has limited practical use in real life. For example, in a simple sorting algorithm, theory provides the algorithms and the bounds on the time required to perform the operations, while implementation performs the actual sorting. This concept applies not only to sorting, but also to all other theoretical problems. Accordingly, implementation complements theory to form computer science.

Combining both theory and implementation is a crucial requirement in any computer science graduate degree. In their joint report, ACM and IEEE-CS attest that there should be a link between these two areas to achieve a balance between them [8]. On the one hand, some students intend to pursue a career in industry. They need to not only excel in the implementation area, but also know whether certain problems are actually solvable, and if possible how to efficiently solve them. On the other hand, other students want to pursue a career where theory is their main focus. Those students need implementation to show how their findings can solve certain problems. Accordingly, it is important to combine both theory and implementation in university courses.

One of the problems in computer science graduate degrees is that the curricula at different universities such as Clemson University [13], The Ohio State University [10], and Virginia Tech [12] allow students to favor one area and neglect the other while meeting the requirements of their program of study. This is mainly because few, if any, of the courses offered in different areas combine both software implementation and the theoretical foundations of those implementations.

In an attempt to solve this problem, we outline a course integrating both theory and implementation. This outline is based on *The Foundations of Distributed Computing,* a course offered at Clemson University [7]. Our course is similar in the topics covered, as well as the sequence in which they are covered. Unlike the foundation course, which is purely theoretical, we integrate implementation projects. Our course description serves as an example of how to integrate both areas in a single course.

Paper Organization. The remainder of this paper is organized as follows. Section II outlines our position on the integration of theory and implementation. Section III presents our vision of a distributed computing course that combines both theoretical topics, and algorithm implementations. Section IV includes a discussion of the benefits and challenges associated with our proposed course. Section V concludes with a summary of our position.

II. Position

Our position is that whenever possible, theory and implementation should be taught within individual cohesive courses. We believe that doing so will be beneficial to students in a number of ways. First, students will not be able to isolate themselves from material that does not match their specific talents, be it theory or implementation. Second, students will not only learn the theoretical foundations, but will also learn

how to map elements of the theoretical model to actual implementation details. This will better achieve the goal of using theory to aid in the construction of correct implementations. Third, courses that include theory and implementation will help students master both areas in their field of concentration. Finally, it will be beneficial for students to see their theoretical studies in action. This will especially be advantageous when studying proposed theoretical solutions that fall short of the desired outcome. We understand that such a course will limit the number of topics that can be covered in a semester. We consider this drawback in detail later in our discussion section. In the next section, we detail the description of our proposed course.

III. Course Description

In this section, we describe our proposed course on distributed computing. We outline the major topics that will be covered, in order, and discuss the way we envision presenting the associated material.

Predicate Calculus. The course begins with a presentation of predicate calculus. The classes teach students how to reason about programs using predicate calculus and a UNITY-style [2] computational model. This portion of the course is purely theoretical as it will serve as a primer for the theoretical study throughout the course.

Gossip. The course then moves to a set of case studies of specific problems encountered during the development of distributed systems. The first case study is diffusing computation where "gossip" algorithms are presented. First, the problem of communicating information from a single source to an entire network is described. Next, an algorithm is developed in class to solve the problem. The instructor will guide the students toward an algorithm that can be proved using the established predicate calculus and UNITY-style computation model. The class will then work through the proof in order to become familiar with the process of proving properties in the selected model. Finally, the instructor will provide an implementation of a gossip algorithm in a suitable programming language or network simulator. The instructor will then lead the class in understanding how the elements of the theoretical model map to the implementation provided.

Termination Detection. The second case study is termination detection. The process of teaching this case study will model the previous; however, students will be more involved in the development of both the proof and the implementation. The instructor will aid the students in developing the proof, guiding the development in the direction of a proof that has already been developed. Once the proof has been completed, the instructor will provide the students with a stubbed-out implementation of a program that is used for distributed termination detection. The students will form small groups to complete the implementation of the program as a homework assignment. The instructor will spend a small amount of time in subsequent classes discussing specific problems the students should avoid while completing the implementation. For example, some obvious solutions to certain problems have pitfalls that are only discovered after careful analysis of the program and the possible cases of the problem. Once the implementations are complete, each group will be asked to give an in-class demonstration of their solution, and to describe how the elements of their solution map to the theoretical proof discussed in class.

Mutual Exclusion. The last case study is on mutual exclusion with an emphasis on the dining philosophers problem. The presentation of this case study will model the previous two. In this case, however, rather than the instructor providing a proof, students will be asked to form groups. Each group will be tasked with developing an algorithm and associated proof. The instructor will answer specific questions from groups, and try to guide the groups toward a correct algorithm and proof. The goal from this exercise is to show the students that the development of algorithms to solve distributed computation problems is non-trivial. Once each group has developed an algorithm and an associated proof, the instructor will assign *different* groups to implement the algorithms. The goal of this exercise will be to practice mapping theoretical concepts to an associated implementation, and to identify possible errors in the theoretical treatment. The assumption is that there will be errors in most groups' proofs. Any error identified will be discussed in class and shown how to be avoided in the future. The class will develop a final proof, algorithm, and its implementation.

IV. Discussion

In this section, we discuss the benefits and challenges associated with our proposed course. We consider potential pitfalls that may arise during the development of such a course. We also attempt to address some of the objections our colleagues might have to the ideas we have presented.

Analysis. The idea of combining both theory and implementation is not novel. For example, this idea has been carried out in other universities and has proved to be successful [9], [1]. Anderson also shows how one area of computer science can help students understand more about the other [1].

Benefits. This type of course provides several benefits. First, this course shows students how to apply theory to the real world and how it can map to implementation. Second, the course actively engages students in the learning process. Rather than simply tracing a proof, the students are involved in developing proofs and implementations. Third, the course helps students learn new proof techniques and also teaches students how to implement distributed computing applications. Third, this course achieves the aim of ACM and IEEE-CS in de-compartmentalizing the topics studied [8]. Finally, the assignments enhance team work and the presentations foster oral communication.

Challenges. Even though the benefits are encouraging, there are still challenges in applying this type of course. First, it may be challenging to find a balance between theory and implementation and not spending more time in one area on the account of the other. We believe instructor(s) can find a balance between the two after offering such a course a number

of times. It may also be necessary to tune the treatment of each topic depending on the experience of the students in the course. Second, it may be difficult to get students excited about the area in which they are weak. It is our hope that in a class with a diverse set of talents from both theory and implementation, students will find encouragement from not only the instructor, but also other students. Our vision is for strong theory students to share their passion for theory with the strong implementation students, and vice-versa. Furthermore, in group projects and assignments the instructor(s) can form groups consisting of students competent in each area. Third, students may become overwhelmed by the details associated with creating distributed systems, hindering the progress of the course. One way to solve this challenge is to use a distributed computing framework such as *DCEZ* [11]. Such a framework hides the implementation details irrelevant to the course and allows students to focus on the algorithms themselves. Finally, there is the potential for "down time" in a course of this nature, where the students are working on a project and it would be inappropriate for the instructor to continue to the next case study. During such times, the instructor could answer any questions the students have about the task at hand. If there are no questions, additional theoretical topics can be covered. These topics might include event ordering, logical and physical time, time synchronization and distributed snapshots.

Potential Pitfalls. The major pitfall is not being able to cover all of the topics that can be covered if the course is purely theoretical. However, it is more beneficial to teach students *how to learn* different topics and not just present a broad range of topics. Additionally, an advanced course can be offered to cover the rest of the material in the same way. This way in each course the students will have the chance to know more about the two areas.

Objections. There may be some objections from instructors who favor one area over the other. For example, instructors who have been teaching implementation courses will not favor adding theory to their courses, and vice-versa. However, this point can be addressed by having team taught courses. For example, there can be an instructor for the theory part and another for the implementation part. In fact, this way the students will benefit the most by gaining from the experience of both instructors in their field of specialization. Additionally, different lab assistants can be available to help students with their implementation or proof assignments.

V. Conclusion

The proposed course outline is just an example of how universities can integrate both theory and implementation in their graduate-level computer science courses, enabling graduate students to master both areas. Although our proposed outline may not fit every theory course, and may limit the number of topics covered in some courses, we believe the advantages discussed earlier outweigh such drawbacks. We have proposed this course outline in our belief that computer scientists should know both areas in their field of concentration. This way we can enhance the quality of education at the graduate level. Hence the students will be able to deal with different problems in a scientific and professional manner by finding not only possible solutions that produce the expected results but also those that are provably efficient and correct.

Acknowledgments

The authors would like to thank Dr. Jason O. Hallstrom for his insightful feedback. The authors would also like to thank the anonymous reviewers for their detailed comments and suggestions.

References

[1] Scott D. Anderson. A course on simulation, probability and statistics. In *SIGCSE '07: Proceedinds of the 38th SIGCSE technical symposium on Computer science education*, pages 110–114, New York, NY, USA, 2007. ACM Press.

[2] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[3] B. C. Dean. Algorithms explained. Unpublished.

[4] Wikipedia The Free Encyclopedia. Computation. http://en.wikipedia.org/wiki/Computation.

[5] Wikipedia The Free Encyclopedia. Computer science. http://en.wikipedia.org/wiki/Computer_science.

[6] Wikipedia The Free Encyclopedia. Theory of computation. http://en.wikipedia.org/wiki/Theory_of_computation.

[7] Jason O. Hallstrom. Cpsc 873: Foundations of distributed computing. http://www.cs.clemson.edu/∼jasonoh/courses/CpSc873_Spring_07/.

[8] IEEE/ACM Joint Task Force on Computing Curricula. Computing curricula 2001, final report. http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/cc2001/cc2001.pdf, December 2001.

[9] Kirby McMaster, Nicole Anderson, and Brian Rague. Discrete math with programming: better together. In *SIGCSE '07: Proceedinds of the 38th SIGCSE technical symposium on Computer science education*, pages 100–104, New York, NY, USA, 2007. ACM Press.

[10] Ohio State University Department of Computer Science and Engineering. Graduate programs. http://www.cse.ohio-state.edu/grad/index.shtml.

[11] Chuck Pheatt. An easy to use distributed computing framework. In Ingrid Russell, Susan M. Haller, J. D. Dougherty, and Susan H. Rodger, editors, *SIGCSE*, pages 571–575. ACM, 2007.

[12] VT Computer Science. Computer science virginia tech. http://www.cs.vt.edu.

[13] Clemson University. Computer science. http://www.registrar.clemson.edu/publicat/catalog/2005GC/gc_coi/gc_cpsc.html.

Using the "New" Internet to Advance the Resolve Group's Goals

Joseph E. Hollingsworth
Indiana University Southeast
4201 Grant Line RD
New Albany, IN 47150
jholly@ius.edu

I. Issue

How the Resolve Group currently goes about the dissemination of Resolve principles to the computing world, and how it goes about fostering the adoption of these principles by non-Resolvers.

II. Position

We have to continue to make use of the traditional methods for dissemination of the Resolve principles, but we need to take advantage of specific tools in today's Internet: to find additional ways to disseminate and draw attention to Resolve principles; to recruit new members to the Resolve Group; and to get work done on the project through wider collaboration.

III. Justification

One of the Resolve Group's long-running battles has been to garner a wider exposure and adoption of fundamental Resolve ideas. Some of the methods used to date include (in no particular order and incomplete): journal publications, conference proceedings, conference presentations, infiltrating existing workshops (e.g., WISR), applying Resolve principles to commercial programming languages, engaging non-Resolve researchers one-on-one, creating and teaching introductory and advanced classes based on Resolve principles, competing for external funding, etc. These approaches have produced positive results, but I believe many from the Resolve Group would agree, that they have not produced as much exposure and adoption as we would have liked.

Recently the Internet has transformed from mainly static sites where users would visit read and possibly download some information, to sites where visitors can do all of the above in addition to creating new content by sharing and collaborating with other site visitors. This new Internet is more about building communities, encouraging participation, and mass collaboration. Some of the tools that facilitate this collaborative creation include wikis, blogs, chat rooms, instant messaging, and websites like SourceForge.net [1].

The Resolve Group as a whole was quick to adopt the tools of the old Internet building mainly static web sites, but the Group as a whole now needs to devote some of its energy to adopting and using the tools of the new Internet in order to better disseminate our ideas, and to attract new talent to the Resolve Group.

IV. Specific Discussion Goals

Enumerate each of the traditional methods of dissemination and reevaluate their impact with respect to "getting the word out" about Resolve, and attracting new talent to the Group.

Enumerate the tools of the new Internet (wikis, blogs, etc.) and determine how we as a group can take increased advantage of these tools for improved dissemination and for attracting new talent.

Examine how we might make our existing software projects (and ones we would like to start) into open source projects in an attempt to harvest work from those outside the group.

References

[1] Tapscott, D., Williams, A.D., *Wikinomics: How Mass Collaboration Changes Everything*, New York, NY, Penguin Group (USA) Inc., 2007, ISBN: 978-1-59184-138-8