Proceedings of the RESOLVE Workshop 2006

Blacksburg, VA

March 22-23, 2006

Edited by Stephen H. Edwards

Virginia Tech TR #06-10
April 12, 2006

Department of Computer Science
Virginia Tech
660 McBryde Hall (0106)
Blacksburg, VA 24061

Position Papers

Preface

The aim of the RESOLVE Workshop 2006 was to bring together researchers and educators interested in:

- Refining **formal approaches to software engineering**, especially **component-based systems**, and

- Introducing them **into the classroom**.

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

- 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 formalisms with model-based behavioral specification approaches
- Formal characterization of user interfaces
- Formal modeling 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
- Using Resolve in undergraduate and graduate CS curricula
- Pedagogical techniques to help teach the above topics

The RESOLVE Workshop 2006 was chaired by Stephen Edwards (Virginia Tech) and sponsored by the Department of Computer Science at Virginia Tech. The remainder of the program committee consisted of Joseph Hollingsworth (Indiana University Southeast), Murali Sitaraman (Clemson University), Bruce Weide (The Ohio State University), and Bill Ogden (The Ohio State University, emeritus).

An Infrastructure to Study and Address Students' Difficulties with Pointers

Paolo Bucci[1], Wayne D. Heym[1], Joseph E. Hollingsworth[2], Timothy Long[1], and Bruce W. Weide[1]

[1] Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210 USA
+1-614-292-5813
{bucci,long,weide}@cse.ohio-state.edu
w.heym@ieee.org

[2] Department of Computer Science
Indiana University Southeast
New Albany, IN 47150 USA
+1-812-941-2425
jholly@ius.edu

ABSTRACT
A model and a taxonomy to characterize pointer manipulations are introduced, along with an instrumentation technology that leverages them to provide students with immediate reports of their pointer errors in C++ programs. A key innovation is that not only does the student get feedback about pointer errors; so does the instructor. The method used to provide students with feedback also permits logging of student errors for analysis by the instructor, thereby facilitating both empirical research into students' understanding of pointers and new possibilities for improved pedagogy. Preliminary data collected using this infrastructure, and other possible uses of it, are discussed.

Categories and Subject Descriptors
K.3.2. [**Computer and Information Science Education**]: Computer science education, Curriculum. E.1. [**Data Structures**]: Lists, Stacks, and Queues. D.3.3. [**Language Constructs and Features**]: Data types and structures, Dynamic storage management.

General Terms
Languages, Experimentation.

Keywords
C++, CS2, data structures, linked list, pointers, references.

1. INTRODUCTION
Ask anyone who has taught a course such as CS2, in which students are learning to use pointers to build linked data structures: students make plenty of errors with pointers! Yet there is no body of research that expressly asks (or answers) why this is so. Of course, pointers involve indirection—and this is a hard concept. But what are the details of students' misconceptions about writing programs that involve pointers? Which pointer manipulations do students most readily understand, which do they find most challenging, which errors do they most often make with pointers in their programming assignments, do they even realize they are making mistakes, how do they debug their programs? Instructors now must appeal to personal experience and anecdotal evidence to try to answer such questions, and for guidance about what to emphasize in class and where to focus student attention. Is it possible for instructors to better understand student understanding (and misunderstanding) of pointers? As with any question of this kind, the answer is "perhaps—if only we had some data!"

Our chief objective is to show how to collect that data; a secondary objective is to suggest how to use it. Specifically:

- We introduce a model for explaining to students the execution-time dynamics of C++ pointers, and a taxonomy for organizing discussion of the correctness issues that arise from their use.
- We explain how we have used this model and taxonomy to give students immediate feedback on pointer errors in their programming assignments.
- We describe how we have adapted the instrumentation used to provide feedback to students so it logs student errors for off-line analyses by the instructor and/or by researchers.
- We discuss preliminary data collected with this infrastructure over the past year in a CS2 course, and suggest future research studies and pedagogical innovations that it makes possible.

The contributions of the paper lie in all four of the above areas. The model itself is novel. It does not purport to explain what pointers are or exactly how pointers work—either directly, as in "a pointer is a memory address", or metaphorically, as in "a pointer is like an apartment key". Rather, it covers an orthogonal issue: the need for students to realize that every manipulation of pointers is either "always safe", "dangerous", or "never safe". It does this by defining a reasonably simple but language-specific finite-state machine (which we discuss in detail for C++) and by classifying its transitions into the above three categories. The instrumentation technique used to give students feedback on their programming errors is based on *checked pointers* [2], but it incorporates a few new twists to adapt that technique to the model and taxonomy of this paper. The method for logging pointer errors at first seems rather straightforward—yet it raises some interesting technical and non-technical issues. Our advice on things to watch for

should help others who might wish to develop a similar infrastructure at other institutions or for other programming languages. Finally, our preliminary data is the first to be reported about the nature of pointer errors in student C++ programs.

The focus of the paper on C++ requires a brief explanation. It is, first and foremost, a product of the fact that C++ is the delivery vehicle in the introductory courses at the authors' institutions. Recent reports indicate that C++ is no longer as popular as Java in introductory CS courses, yet it is used in perhaps 25% of CS1 courses [5]; and it probably accounts for a similar or even larger share of CS2 courses, where most students meet linked data structure implementations. Moreover, a significant fraction of industrial software projects use C/C++. So, there are still good reasons to teach students how to build software in which automatic garbage collection is not assumed. Whether this material arises in CS2 or in a later course, our infrastructure can be used. In fact, preliminary data (see Section 5) suggest that failure to reclaim storage properly is a common error made by students learning to use pointers—far more popular than, say, dereferencing a null pointer.

The paper is organized as follows. Section 2 introduces the model and taxonomy, which serves as the basis for providing students with immediate feedback on pointer errors in their programming assignments. Section 3 briefly describes how checked pointers work in C++ and outlines what we changed from the approach proposed in [2]. Section 4 discusses a few unanticipated technical problems that arose during our first year of data collection and reports on some non-technical issues raised by the very act of logging student errors. Section 5 summarizes some of the data collected so far. Finally, Section 6 concludes with some suggestions for future empirical research and instructional innovations made possible by the infrastructure described. Because of the page limit, Sections 4 and 5 discuss only the version of the software used, and the preliminary data collected, at OSU. Similar conclusions apply to the software used, and the data collected, at IUS.

2. MODEL AND TAXONOMY

This section outlines our model for explaining certain aspects of the execution-time dynamics of C++ pointers, and for classifying problems that may arise from their use.

2.1 Replacing Built-In C++ Pointers

In C++, it is possible to ensure that only "acceptable" (to the instructor) uses of pointers will compile. This prevents certain classes of errors students might otherwise make. For example, we outlaw pointer arithmetic, e.g., using an expression such as `p++` or `p+2`, though it is legal for built-in C++ pointers. Ruling out such constructs can be achieved by requiring students to use a `Pointer` class template in which only certain operators are defined [3]. To declare two pointers to `int` called `p` and `q`, for example, a student simply writes:

```
Pointer<int> p, q;
```

rather than:

```
int *p, *q;
```

The `Pointer` template makes public only operators that the instructor wants students to be able to use, say `*`, `->`, `=`, `==`, and `!=`; nothing else will compile. Figure 1 shows the operators that are public in the `Pointer` template that is used throughout this paper and available for download from our web site at http://www.cse.ohio-state.edu/sce/SIGCSE2006. Here, `p` and `q` are variables of type `Pointer<T>` for the same parameter type `T`; or `q` may be the constant `NULL`. It is technically possible to override **`new`** and **`delete`** to retain all the syntax of built-in C++ pointers except declaration; or, to provide slightly different syntax for some operators (such as the Pascal-like syntax of `New` and `Delete` shown in Figure 1). We have tried both approaches to syntax at our institutions and have seen no apparent impact on student behavior. On the other hand, we have kept the semantics of built-in C++ pointers, warts and all, on the grounds that students should be aware of the problems that can arise from using language-supplied pointers.

Unary	Binary

`New(p);`
`Delete(p);`
`*p`
`p->...`
	`p = q` (assignment and copy constructor)
	`p == q`
	`p != q`

Figure 1: Allowable `Pointer<T>` operations

So, as with any version of pointers, plenty of code that will *compile* with this approach can still be wrong. Using some of the operators under certain conditions is an error that a compiler, in general, cannot detect, and that typical C++ compilers do not report even when technically they might do so with a sophisticated static analysis. Such errors include dereferencing a null pointer, dereferencing a pointer that has been deleted, creating a memory leak by allowing the last pointer to a block of memory to leave scope, etc. Fortunately, as explained in [2] and discussed further in Section 3, the `Pointer` template can be implemented in a way that (almost) every such error can be detected and reported immediately at the point during program execution where it occurs. By contrast, most errors with built-in C++ pointers—indeed, *all* such errors except dereferencing a null pointer—might not be manifested through observably anomalous behavior during program testing. Even if they are manifested eventually, this might not happen until far beyond the program point where the pointer error actually occurred. And even then, there might be only a cryptic system-level error message such as "bus error" or "segmentation fault" that offers no help for debugging.

2.2 Abstract Pointer States and Transitions

The model in this paper has a slightly different purpose than the traditional explanatory devices for pointers. It abstracts the actual value of a pointer (i.e., now `Pointer`) variable—which could be any one of millions or billions of memory addresses—into one of a small set of states. This simplification implies that the model cannot be used to reason *in full detail* about what happens during execution of a program that uses pointers. So, regardless of whether an instructor uses our model, completeness demands that he/she still adopt a traditional explanation of pointer details, e.g., the direct version that a pointer variable's value is a memory address. The four states of a pointer variable in the simplified model are:

- **Alive** — the variable refers to memory that the storage management system has given to the program, i.e., the program "owns" that memory;
- **Dead** — the variable refers to memory that the storage management system has never given to the program or has reclaimed from it, i.e., the program does not "own" that memory;

- **Null** — the variable refers to no memory location at all.

The need for the last state arises because of the possibility of memory leaks:

- **Out of scope** — the variable is not in scope.

Figure 2 and an accompanying discussion in class help students understand the model. The diagram shows the states that a single pointer variable, say `p`, can be in; and the various transitions it might undergo via the allowable pointer operators:

- "declare" means the variable `p` is being declared;
- "}" means the variable `p` is leaving the scope in which it was declared;
- "*" means the variable `p` is being dereferenced, using either `*` or `->`;
- "NULL" means the variable `p` is being assigned the constant `NULL`;
- "New" and "Delete" are obvious.

An exhaustive analysis reveals that there are exactly two other ways that a pointer variable `p` can change state via the operators in Figure 1: one obvious and the other subtle. Both arise from copying pointers with `=`. First, a variable `p` that is in scope can be assigned a pointer `q` that is in a different state, in which case `p` changes to the state of `q`. For example, if `p` is in state **Null** and `q` is in state **Alive**, then after the statement `p = q;`, `p` is also in state **Alive**. None of these six obvious transitions is shown in Figure 2. Second, if `p` is **Alive** and is aliased to another pointer `q` that transitions from **Alive** to **Dead**, then `p` also becomes **Dead**. This is the only *spontaneous* transition, i.e., the only possible change to the state of `p` that can occur as the result of a statement that does not syntactically mention `p`; it captures an interesting effect of aliasing. This transition is shown as an unlabelled double-line arrow to notify students that it is possible, and that they must beware of it.

Of course the other binary operators, which test equality and inequality of pointers, do not cause any state changes.

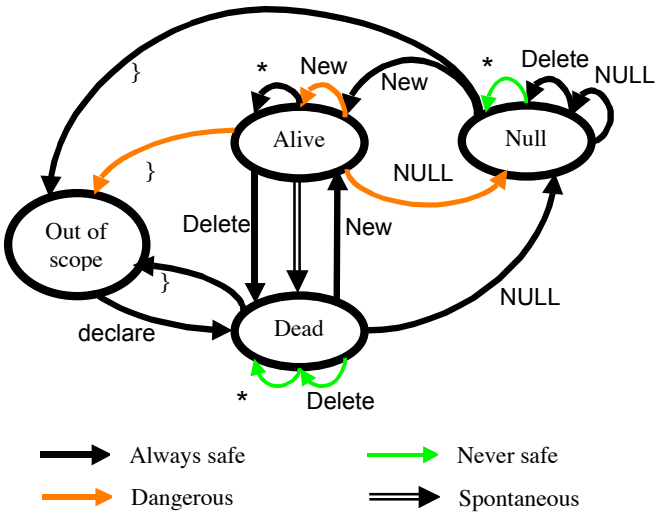

Figure 2: C++ State Machine and Taxonomy of Transitions

2.3 Taxonomy of Transitions

This model for explaining pointer states and transitions allows the classification of every manipulation of a pointer as "always safe" (i.e., "good"), "dangerous", or "never safe" (i.e., "bad"). Fortunately, as summarized in Figure 2, many transitions are always safe. A few are never safe, though of course they will compile: dereferencing a pointer that is **Null**, and dereferencing or deleting a **Dead** pointer.

The remaining transitions are "dangerous", in the sense that they may or may not result in memory leaks. If a pointer variable `p` is **Alive** and it leaves scope, then whether there is a memory leak depends on whether the memory referenced by p is also reachable via another **Alive** pointer that is aliased to `p`; similarly for the statements `New(p);` and `p = NULL;` in the same situation. That such a highly abstracted model of pointer behavior fails to predict, for sure, whether a memory leak will result from these transitions is a consequence of the fact that it abstracts away actual memory addresses. So, there is a trade-off between the model's predictive power and ease of reasoning. The model's simplicity helps a student (or, potentially, a static analysis tool) identify possible trouble spots in a program without demanding detailed reasoning about actual pointer values. Yet, as Section 3 explains, it provides the ability to immediately detect, report, and log at run-time transitions that are never safe, as well as dangerous ones that turn out actually to cause memory leaks.

Note that the corresponding finite-state machine for Java in Figure 3 is substantially simpler than the one for C++, having but one transition that is never safe and none that are either dangerous or spontaneous. Because there is no explicit storage reclamation in Java (i.e., no "Delete" transition) and no **Dead** state, life is much simpler for the student and for the professional programmer. This is precisely why students who have learned how to use Java references to implement linked data structures must *not* be expected to do so competently in a non-garbage-collected language like C++, without further education and practice.

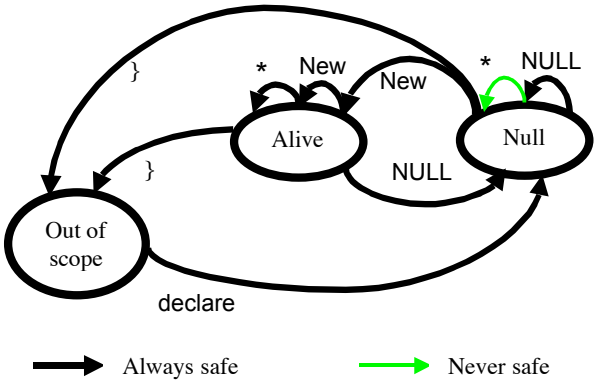

Figure 3: State Machine for Java References

3. CHECKED POINTERS

The `Pointer` template we use is based directly on checked pointers as described in [2]. It supports the operations in Figure 1, and reports at run-time immediately, upon occurrence, every "never safe" transition and (almost) every "dangerous" transition that actually leaks memory. In addition, the template checks and reports whenever a **Dead** pointer variable is compared using operator `==` or operator `!=`. With respect to

implementation of the `Pointer` template, interested readers can consult [2] for details and/or download our code.

One issue needs to be explained here. The qualifier "almost" above is the result of using reference counting to identify memory leaks—an extension to the approach used in [2]. If a reference count for some memory block reaches zero, then there really is no way to access it. However, it is possible to construct circularly linked structures in which blocks of memory are inaccessible yet where all those blocks have positive reference counts: they point to each other. Like [1] and [2], we do not try to detect memory leaks of this sort upon occurrence, but instead wait until the end of the program and report all memory that has been allocated but not yet deallocated. In other words, any memory leak reported by our checked pointers is truly a leak, but not all leaks are immediately reported. Whether this has practical importance depends on how likely students are to write code that creates defective circularly linked data structures, either on purpose or by accident; and this, in turn, on the assignments made by the instructor.

4. LOGGING AND ASSOCIATED ISSUES

This section outlines several issues that anyone trying to collect similar data from their own students should consider.

4.1 Technical Issues

When a student program generates a pointer error at run-time, the code of the `Pointer` template first reports to the student the nature of the error (e.g., "Deleting dead pointer") and produces a call trace to help in debugging. The code then writes to a central log file all this information, plus the time of the error (accurate to the second), the student's encrypted login name, the name of the program the student was executing, the command the student used to run the program, and the total number of calls executed by the program. Finally, the code uses a C++ `exit` call to quit the program.

For GCC version 3.4.1 running under Solaris, we encountered sticky technical issues in obtaining nearly every piece of system information for the log: the call trace, the name of the executing program, etc. Anyone adapting our code to different circumstances might anticipate a few technical hurdles here.

At OSU, students do their programming assignments in the introductory courses using department computers, which are Sun "login servers" that they can access from dedicated labs or via the internet. The centralized nature of this facility helped simplify logging, but it is not essential if it can be assumed that students have internet connectivity over which error reports can be sent from the student's computer to a central site. Error-logging functionality can thereby be provided at the cost of some additional code that might vary according to the institution's local circumstances.

We also faced some technical questions when interpreting preliminary log data. Specifically, our log files contained a few situations where the same student was reported to have made two errors at the same time—despite the fact that processing the first error ends with an `exit` call. We found that this *could* happen under some conditions. Specifically, a student's code might include a global variable (i.e., static storage in file scope). A pointer error that generates the first log entry might result in that variable's data representation invariant being violated. When there are global variables, a call to `exit` does not immediately quit the program [1,3] because the C++ compiler registers destructors for global variables so they are executed before the program actually quits as a result of `exit`. Thus, a second pointer error might occur—even in correct code for a global variable's destructor. In this situation, using `abort` is a better way to quit.

We also discovered a situation in which one student experienced the same error some thirty times in a row, at approximately one-second intervals. This could happen if the student ran a script as part of the testing process.

Therefore, we decided to count—not while logging but while processing log data off-line—only the first of a series of closely spaced errors by the same student, and to ignore any error by that student within 10 seconds of the previous one. Based on our observations of students in closed lab situations and on our own attempts to make a simple change in a program and then re-compile and re-run it as quickly as possible, at least this much time must elapse between successive legitimate error records. In both the case of the two-error situation involving destructors of global variables, and in the case of repeated execution via a script, it makes sense to consider the first error as a "true" error and the remainder as spurious.

4.2 Non-Technical Issues

The most important non-technical issue we faced was a consequence of our mere intent to use logged data in research to be reported to the CS education community. This meant that an institutionally-approved human subjects protocol was required before data collection could even begin [4]. If we had decided to use the error logs only for local instructional purposes (e.g., to identify for special attention students having the most trouble on their programming assignments), then no such obstacle would have been imposed.

Our approved protocol includes a plan for collecting baseline data without the students' knowledge: a "deception" of the students, in the parlance of such protocols. The reason is that we do not want to preclude being able to study later whether the fact that students know that their errors are being logged might affect their behavior. However, even without a deception, an approved protocol is required for such research.

It also is essential to maintain confidentiality so individual students are not identifiable from any data that could be published. We do not want to preclude longitudinal studies of individual student behavior. Hence, our logging software records encrypted student login names, so if a log file were inadvertently compromised there would be no way for anyone to connect an error record with a particular student. Moreover, only one of the investigators even has "read" permission for the data log files, in order to further limit the likelihood of such a security breach. These provisions have proved acceptable to our Institutional Review Boards.

5. PRELIMINARY DATA

We logged all student pointer errors at OSU in Au04-Sp05, then filtered the data as explained in Section 4.1 and also limited the focus to students doing assignments for CS2. Three programming assignments were involved: a closed lab (done in pairs) to implement a stack class, given a queue class as a model; and two open labs (done individually) to implement a singly linked list class and a doubly linked list class. We logged 2765 pointer errors made by 139 students.

Figure 4 lists each possible error message seen by these students, along with two pieces of information for each: the percentage of students making at least one error overall (i.e., 139) who made that particular error at least once, and the percentage of all errors (i.e., 2765) accounted for by that particular error.

Error	Students Making Error	Percentage of All Errors
Creating memory leak by pointer leaving scope	74%	21%
Creating memory leak by using = (i.e., assignment)	61%	18%
Creating memory leak by using = NULL (i.e., assignment)	4%	1%
Creating memory leak by using New	1%	0%
Deleting dead pointer	19%	2%
Dereferencing dead pointer by using * or ->	70%	33%
Dereferencing null pointer by using * or ->	57%	16%
Using dead pointer with != (i.e., inequality checking)	30%	5%
Using dead pointer with != NULL (i.e., inequality checking)	10%	1%
Using dead pointer with == (i.e., equality checking)	13%	2%
Using dead pointer with == NULL (i.e., equality checking)	9%	1%

Figure 4: Preliminary Data

Our logs contain a wealth of other information and will require more detailed analyses during the initial data exploration phase, in which we seek to identify features in the data that should suggest carefully designed future experiments. A few interesting observations already evident are:

- About 5/6 of all errors (i.e., all except dereferencing a null pointer) might not have been detected at all without checked pointers; and if they appeared, they would have resulted in later mysterious behavior of the program rather than straightforward error messages.
- Most students who made any errors created a memory leak.
- Most students who made any errors dereferenced a dead pointer (more than dereferenced a null pointer); indeed, a third of all errors were of this kind.
- Some errors apparently were much easier for students to correct than other errors. Many students made the following kinds of errors at least once, yet they account for a relatively small fraction of all errors: creating memory leaks in various ways, and using the comparison operators with dead pointers.

6. CONCLUSION

The infrastructure described in this paper has been designed to support empirical research studies that we expect to conduct in the future. Some are self-evident, e.g., investigations of conceptual errors vs. technical ones, examination of the value of pictures such as Figure 2 to explain the model, etc. Other studies are suggested by the preliminary data. For example, consider the first bullet point above. As instructors, we have little doubt that immediate detection and reporting of pointer errors facilitates student debugging as compared to built-in C++ pointers. This claim could be tested by building multiple versions of checked pointers that provide different error diagnostics to students, while still logging all the data of the current version. One version could allow the program to continue after every error to perform exactly like built-in C++ pointers, providing a control group for the study. Another version could report each error upon detection and stop the program, but always with the same general message such as "pointer error". Another version could provide detailed error messages as shown in Figure 4. Some questions to be answered include: How does student debugging behavior differ under such circumstances? Do students end up making significantly fewer total errors when presented with immediate error detection, and/or when given detailed error messages?

Pedagogical innovations are also facilitated. The infrastructure could be adapted to alert the instructor to students who are making many pointer errors, to those who are making a single error many times in a short period, etc. Such students could be singled out for special attention. Or, an instructor could bring to class a chart showing how many students made each type of error while doing the previous assignment, and call on students to explain the circumstances that led to errors of certain kinds. There are many creative ways in which error data—now, of course, with logging not concealed from students—could be used to focus classroom and/or individual attention on problems students are actually having, rather than on problems the instructor might have expected them to have.

7. ACKNOWLEDGMENTS

We appreciate the many contributions of our students, who have provided helpful feedback on the comprehensibility of the model and taxonomy described in Section 2, and who have (so far unwittingly) provided us with a wealth of log data.

8. REFERENCES

[1] Anderson, P., and Anderson, G. *Navigating C++ and Object-Oriented Design*. Prentice Hall, Upper Saddle River, N.J. ,1998.

[2] Pike, S.M., Weide, B.W., and Hollingsworth, J.E. Checkmate: Cornering C++ Dynamic Memory Errors With Checked Pointers. *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 2000, 352-356.

[3] Stroustrup, B. *The C++ Programming Language (3rd edition)*. Addison Wesley Longman, Reading, MA, 1997.

[4] U.S. Department of Health and Human Services. Office for Human Research Protections (OHRP). Viewed 4 Sept 2005 at http://www.hhs.gov/ohrp.

[5] Van Scoy, F.L. The Reid List of the First Course Language for Computer Science Majors. Viewed 4 Sept 2005 at http://www.csee.wvu.edu/~vanscoy/reid.htm.

Using Industrial Tools to Test and Grade Resolve/C++ Programs

Stephen Edwards
Dept. of Computer Science
Virginia Tech
660 McBryde Hall (0106)
Blacksburg, VA 24061 USA

edwards@cs.vt.edu
Phone: +1 540 231 5723
Fax: +1 540 231 6075
URL: http://people.cs.vt.edu/~edwards/

Abstract

We can adapt industrial-quality tools for developing, testing, and grading Resolve/C++ programs, and use them to bring modern software testing practices into the classroom. This paper demonstrates how this can be done by taking a sample Resolve/C++ assignment based on software testing ideas, building a simple Eclipse project that handles build and execution actions for the assignment, writing all of the tests using CxxTest, and processing a solution through Web-CAT, and flexible automated grading system. Using tool support to bring realistic testing practices into the classroom has demonstrable learning benefits, and adapting existing tools for use with Resolve/C++ will allow these same techniques to be used in courses where Resolve/C++ is used.

Keywords

Software testing, CxxTest, Eclipse, JUnit, unit testing framework, test-driven development, test-first coding, IDE, interactive development environment, Web-CAT, automated grading

1. Introduction

Software testing is a topic that does not receive full coverage in most undergraduate curricula [Shepard01, Edwards03a]. If we want to teach testing practices more effectively, it may be appropriate to integrate software testing across many--or even most--courses in an undergraduate program [Jones00, Jones01, Edwards03a]. We have had some success with this approach in our core curriculum at Virginia Tech, after integrating software testing throughout our freshman and sophomore courses.

There are a number of potential benefits to learning when software testing is included in the curriculum, since formulating software tests requires a student to formulate and write down their own understanding of how the software they are writing is intended to behave [Edwards03b]. Further, running tests requires students to experimentally verify (or refute) their understanding of what their code does. Experimental results suggest that student code quality improves as a result. One of our experiments showed an average 28% reduction in bugs per thousand lines of non-commented source code (bugs/KNCSLOC), with the top 20% of students writing their own tests achieving 4 bugs/KNCSLOC or better--comparable to commercial quality in the U.S. Students who did only informal testing on their own never achieved this level of quality, with the best students achieving approximately 32 bugs/KNCSLOC [Edwards03b].

2. The Problem

To make software testing practices a regular part of the classroom experience, two things are critical: we must make it easy for students to write and execute tests with minimum overhead, and we must provide concrete and directed feedback on how students can improve their performance. Both of these goals are solvable with appropriate tool support.

First, using an appropriate unit testing framework can simplify test writing and execution. For Java, the JUnit framework [JUnit06] provides excellent support that is easy for students to grasp and use. Similar frameworks, which go by the name *XUnit* frameworks, exist for other languages as well [XProgramming06]. The problem is that **no such unit testing framework exists for Resolve, or Resolve-based languages like Resolve/C++**.

Second, automated grading tools can be used to provide clear and concrete feedback to students on performance. Web-CAT is one such automated grading system [Edwards03a, Edwards04]. It supports assignments where students are required to write tests for their own code. For students programming in Java or C++, it also instruments student code and collects test coverage data as student tests are executed. Students receive feedback in the form of a color-highlighted HTML source code view that highlights portions of the code that have not been executed or that have been undertested. Still, however, **no such grading tools exist for Resolve or Resolve-based languages**.

3. The Position

We can adapt industrial-quality tools for developing, testing, and grading Resolve/C++ programs, and use them to bring modern software testing practices into the classroom.

More specifically, we can adapt an appropriate unit testing framework so that it works with Resolve/C++. We can also adapt a professional-level IDE that is still suitable for classroom use. Finally, we can adapt a flexible automated grading system to work with Resolve/C++ and provide concrete feedback on correctness and testing.

While no XUnit framework exists for Resolve, why not adapt one from another language? At Virginia Tech, we have had success using CxxTest [CxxTest06] with students learning to program in C++. It is possible to use CxxTest to write unit tests for Resolve/C++ components in order to bring unit testing practices into the classroom. Further, the Eclipse-based IDE support we use for C++ development will also work for Resolve/C++ development, including full GUI support for unit test execution and viewing of results. While Eclipse is a professional IDE, it is seeing increasing use in educational settings as well [Storey03, Reis04].

Together, CxxTest plus Eclipse will provide a modern, high-impact IDE environment for developing Resolve/C++ code that will provide greater ease of use for students. Further, it will ease some of the transition out of Resolve/C++ to other languages and tools. But most importantly, it will allow industry practices regarding unit-level software testing to be included in a Resolve/C++ classroom, along with the learning benefits this approach supports.

Note that the CxxTest framework described here is completely independent of Eclipse. It can also be used via the command line or a makefile without any IDE support if desired. Both command-line and IDE approaches will be demonstrated at the workshop as part of the paper presentation.

Finally, Web-CAT provides a great deal of flexibility for automated grading tasks by providing a plug-in architecture so that instructors can extend its grading capabilities for different assignments. Plug-ins for grading C++ assignments that include student-written CxxTest-style test cases already exist, and provide support for using a commercial code coverage tool called Bullseye Coverage to give students feedback on where they can improve their testing. Web-CAT can be extended to support Resolve/C++ grading by adapting the existing C++/CxxTest plug-in to work with Resolve/C++ too.

4. Justification

Justification for this position comes in the form of a "proof by example". We have taken a sample Resolve/C++ assignment based on software testing ideas, built a simple Eclipse project that handles build and execution actions for the assignment, written all of the tests using CxxTest, and processed a sample solution through Web-CAT using an adapted Resolve/C++ plug-in. This section will summarize the example, show how CxxTest test cases as written, and illustrate how the Eclipse interface presents test results.

For our example, we chose CSE 221's closed lab 5, a Resolve/C++ assignment used at Ohio State. In this lab, students must write a test suite to demonstrate a number of bugs in a `Swap_Substring` operation. Students in CSE 221 currently write test driver programs that read commands from stdin and write output to stdout, and allow one to exercise all of the methods under test with user-specified parameters. Students write test cases, or entire suites of test cases, as plain text files that can be fed to such a test driver using I/O redirection on the command line.

Unfortunately, test inputs in such a format do not include any corresponding expected output. Instead, textual output from the test driver is typically captured in a separate output file. Regression testing can be performed by comparing output from a new test run against stored output from an earlier test run using tools like `diff`. However, it is cumbersome for students to write and maintain their own expected output, and without this step, automated checking for correct test results is challenging.

In the closed lab 5 assignment currently being used, students simply write a single test suite (a test input file). As part of the lab setup, students have access to eight separate test driver programs that are provided for them, where each test driver encapsulates a different buggy implementation of the `Swap_Substring` operation. Students also have access to a test driver that correctly implements this operation. Students are also given a helper script that will run a student's test input file against one buggy test driver, also run the same test input against the correct test driver, and then provide the student with the `diff` results on the two output files. This is a form of *back-to-back* testing where a known correct implementation is used as the test oracle for a (possibly) buggy alternative implementation.

There are several disadvantages of this approach. First, students only write test inputs--they are never forced to articulate their own understanding of what the code should do, but only need write down how it should be invoked. Second, students cannot use back-to-back testing easily on new code that they write, since it requires a reference implementation that is known to be bug-free to compare against. Third, using this approach requires that one construct a test driver for each unit to be tested. This involves additional input, parsing, and output code that is not directly relevant to the task itself and that may also contain its own bugs. The more sophisticated the component to be tested, the more work must go into the test driver. Also, if one wishes to extend the testing scenario, say by allowing multiple objects to interact, or by adding a new method to the class under test, the test driver code must be extended and kept in-sync with the code being developed. Fourth, this approach does not keep all of the test information in one place. The test input is in one text file, the expected output (if the student writes it at all) is in another, and the test driver and actual calls to the class under test are in a third location inside the test driver program. Keeping these all in sync becomes more difficult as component complexity increases.

XUnit-style frameworks fix this problem by (a) making all test cases directly executable, written directly in the programming language; (b) allowing the expected output or behavior change to be expressed as part of the test case

itself; and (c) eliminating the need for test drivers by providing a framework the provides all the features of a completely reusable test driver that can work with any set of test cases, so no input/parsing/output code need be written in order to run tests. To see how this works, examine Figure 1, which shows a single test case enclosed in a `CxxTest::TestSuite` class. This test case is for the `Swap_Substring` operation from closed lab 5.

```
1  #ifndef SWAP_SUBSTRING_TESTS_H_
2  #define SWAP_SUBSTRING_TESTS_H_
3
4  #include <cxxtest/TestSuite.h>
5  #include "RESOLVE_Foundation.h"
6  #include "../CI/Text/Text_Swap_Substring_1_Body.h"
7
8  class Swap_Substring_Tests : public CxxTest::TestSuite
9  {
10 public:
11
12     void testSwapSubstring()
13     {
14         // Swapping all of non-empty t1 and non-empty t2
15         Text_Swap_Substring_1 t1;
16         Text_Swap_Substring_1 t2;
17         Integer pos = 1;
18         Integer len = 3;
19
20         t1 = "hello";
21         t2 = "world";
22
23         t1.Swap_Substring( pos, len, t2 );
24
25         TS_ASSERT_EQUALS( t1, "hworldo" );
26         TS_ASSERT_EQUALS( t2, "ell" );
27         TS_ASSERT_EQUALS( pos, 1 );
28         TS_ASSERT_EQUALS( len, 3 );
29     }
30 };
31
32 #endif /*SWAP_SUBSTRING_TESTS_H_*/
```

Figure 1. A CxxTest test case.

In Figure 1, the `testSwapSubstring()` method is a single test case written as executable code. In this example, it creates an object, calls the `Swap_Substring` method, and makes assertions about the results. In other words, it encapsulates one test case, including the setup, the test actions to be carried out, and the behavior that should be observed if the test "passes". A `TestSuite` class can contain as many of these test cases as desired, each framed as a separate method (that is, a separate public void method, taking no parameters, and having a name that begins with "test"). A `TestSuite` class can also contain helper methods that are reused in different test cases. Finally, a `TestSuite` can even contain common "set up" actions that are performed before each test case in the suite, as well as common "tear down" actions performed after each test case, in order to extract recurring pieces of infrastructure when needed.

As part of the build process, the CxxTest build support automatically identifies the classes that are subclasses of `CxxTest::TestSuite`, automatically identifies all of the test case methods in each such class, and automatically builds the necessary test driver to execute all of the tests the student has written. If there is no `main()` procedure in the project, then the test driver itself will provide one. Otherwise, test execution happens as global objects are initialized, just before the student's `main()` method is called.

Using CxxTest reduces the process of writing test cases to a fairly simple coding exercise, which is something students have already practiced. The CxxTest framework takes care of all of the other details regarding test execution and result reporting. When run from the command line, this single test would produce output like that shown in Figure 2. If a buggy version of `Swap_Substring` that failed this test case were used instead, the output would be similar to Figure 3. This output is a little odd because the default CxxTest machinery does not know how to write Resolve/C++-style values to an output stream, but that can be remedied easily.

```
Running 1 test
.
Failed 0 of 1 tests
Success rate: 100%
```

Figure 2. Output from a successful test run.

```
Running 1 test
In Swap_Substring_Tests::testSwapSubstring:
../test-cases/Swap_Substring_Tests.h:22: Error: Expected (t1 == "hworldo"), found
 ({ E4 09 51 00 CE 29 82 00 ...  } != hworldo)
Failed 1 of 1 tests
Success rate: 0%
```

Figure 3. Output from a failed test case.

In addition to using CxxTest to write test cases, students could also use an IDE, like Eclipse, to compile and test their code. As part of our Web-CAT SourceForge project, we have a CxxTest plug-in for Eclipse that provides a graphical view of CxxTest results. Figure 4 shows a partial screen shot of the CxxTest view within Eclipse on this example.

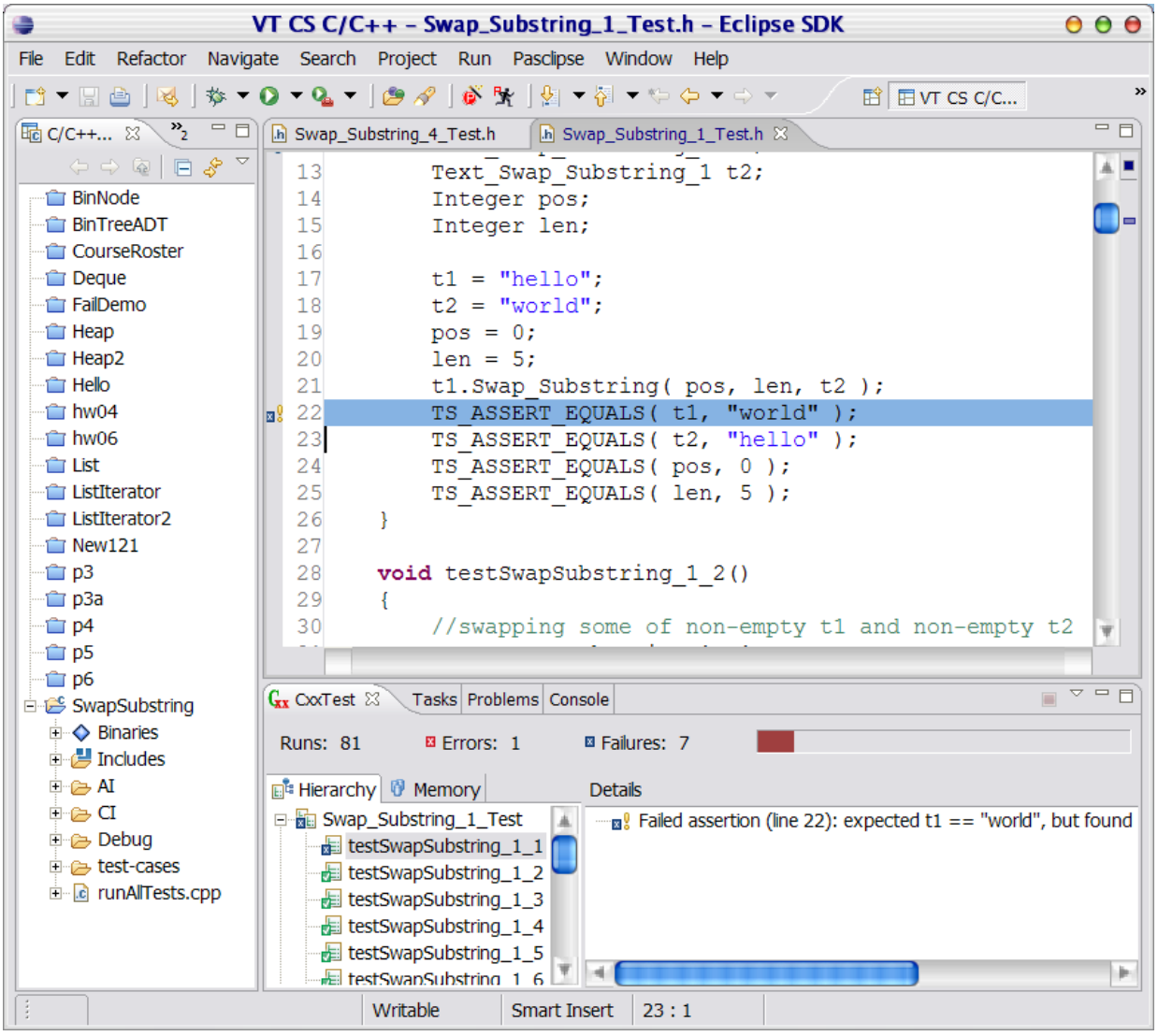

Figure 4. A screen shot of the CxxTest graphical view within Eclipse.

Finally, we customized the CxxTest-based grading plug-in for Web-CAT so that it also supports Resolve/C++ assignments. We submitted this example. Web-CAT produces a variety of feedback to students, most of which is captured in a unified, color-highlighted HTML "print out" of the student's submission. Figure 5 provides a brief example of what this output looks like for Resolve/C++ code using the modified plug-in. You can hover your mouse over the highlighted code lines to see why specific portions have not been tested as well as necessary. Resolve/C++-specific keywords are also highlighted, thanks to the customized plug-in. More information on Web-CAT is available elsewhere [Edwards03a, Edwards03b, Edwards04].

```
1  //  /*-------------------------------------------------------------------*\
2  //  |   Concrete Instance Body : Text_Swap_Substring_1
3  //  \*-------------------------------------------------------------------*/
4
5  #ifndef CI_TEXT_SWAP_SUBSTRING_1_BODY
6  #define CI_TEXT_SWAP_SUBSTRING_1_BODY 1
7
8  ///-------------------------------------------------------------------
9  /// Global Context ----------------------------------------------------
10 ///-------------------------------------------------------------------
```

```
11
12  #include "Text_Swap_Substring_1.h"
13  /*!#include "CI/Text/Text_Swap_Substring_1.h"!*/
14
15  ///------------------------------------------------------------------------
16  /// Public Operations ----------------------------------------------------
17  ///------------------------------------------------------------------------
18
19  procedure_body Text_Swap_Substring_1 ::
20      Swap_Substring (
21          preserves Integer pos,
22          preserves Integer len,
23          alters Text_Swap_Substring_1& t2
24      )
25  {
26      object Integer index = pos + len - 1;
27      object Text_Swap_Substring_1 tmp;
28
29
30      // Fails when swapping all of non-empty t1 and non-empty t2
31      if ((self.Length () > 0) and
32      (t2.Length () > 0) and
33      (self.Length () == len))
34      {
35      // should be while (index >= pos)
36      while (index > pos)
37      {
38          object Character c;
39
40          self.Remove (index, c);
41          tmp.Add (0, c);
42          index--;
43      }
44
45      index = t2.Length () - 1;
46      while (index >= 0)
47      {
48          object Character c;
49
50          t2.Remove (index, c);
51          self.Add (pos, c);
52          index--;
53      }
54
55      t2 &= tmp;
56      }
57      else
58      {
59      while (index >= pos)
60      {
61          object Character c;
62
63          self.Remove (index, c);
64          tmp.Add (0, c);
65          index--;
66      }
67
68      index = t2.Length () - 1;
69      while (index >= 0)
70      {
71          object Character c;
72
73          t2.Remove (index, c);
74          self.Add (pos, c);
75          index--;
76      }
77
78      t2 &= tmp;
79      }
80  }
81
82
83  void Text_Swap_Substring_1::operator =(const Text& rhs)
84  {
85      Text::operator=(rhs);
86  }
87
88
```

```
89  void Text_Swap_Substring_1::operator=(const Text_Swap_Substring_1& rhs)
90  {
91      Text::operator=(rhs);
92  }
93
94
95  #endif // CI_TEXT_SWAP_SUBSTRING_1_BODY
```

Figure 5. Example code view produced by Web-CAT.

5. Related Work

A number of other educators have advocated including software testing across the curriculum [Shepard01, Jones00, Jones01]. An overview of related work appears elsewhere [Edwards03a, Edwards03b]. The Eclipse and CxxTest support described here have been reported in the more general context of supporting Java and C++ development as well [Allowatt05].

6. Conclusion

XUnit-style testing frameworks provide many benefits for students. They make it easier to write and execute operational tests on individual classes and methods. Once written, XUnit-style tests are completely automated. As a result, they completely automate regression testing, so that students can re-run all their tests each time they add some new code or modify a feature. When students are encouraged to write their tests as they go--"write a little test, write a little code"--tests give students greater confidence that the code they have written so far works as intended. It also gives students a better feel for how much they have completed vs. how much remains to be done, and gives students greater confidence when they repair or modify code that is already working. Finally, when students write tests this way, it siginificantly reduces or prevents **big bang integration** problems, since each method has been tested in isolation before classes are assembled into larger structures. In perception surveys, students report that they see these benefits themselves, and prefer to use such techniques even when they are not required in class (once they have been exposed, that is) [Edwards03b].

CxxTest provides a useful vehicle for obtaining these benefits in class when students are programming in C++. The same tools can also be used on Resolve/C++ code with no real modification needed. Further, tool support--like Eclipse's CDT and Virginia Tech's CxxTest support for students--can also be used on Resolve/C++ programs with no modification. While this leaves some cosmetic issues unaddressed, it points in a promising direction away from simple text-based test driver programs as a way to teach students about software testing, as well as introducing software testing practices into more and more Resolve/C++ class activities.

References

[Allowatt05]
Allowatt, A., and Edwards, S. IDE Support for Test-driven Development and Automated Grading in Both Java and C++. In *Proc. 2005 OOPSLA Eclipse Technology eXchange Workshop*, ACM, 2005, pp. 100-104.

[CxxTest06]
CxxTest home page. http://cxxtest.sourceforge.net/.

[Edwards03a]
Edwards, S.H. Rethinking computer science education from a test-first perspective. In *Addendum to the 2003 Proc. Conf. Object-oriented Programming, Systems, Languages, and Applications*, ACM, 2003, pp. 148-155.

[Edwards03b]
Edwards, S.H. Improving student performance by evaluating how well students test their own programs. *J. Educational Resources in Computing*, 3(3):1-24, Sept. 2003.

[Edwards04]
Edwards, S.H. Using software testing to move students from trial-and-error to reflection-in-action. In *Proc. 35th SIGCSE Tech. Symp. Computer Science Education*, ACM, 2004, pp. 26-30.

[Jones00]
Jones, E.L. Software testing in the computer science curriculum--a holistic approach. In *Proc. Australasian Computing Education Conf.*, ACM, 2000, pp. 153-157.

[Jones01]
Jones, E.L. Integrating testing into the curriculum--arsenic in small doses. In *Proc. 32nd SIGCSE Technical Symp. Computer Science Education*, ACM, 2001, pp. 337-341.

[JUnit06]
JUnit home page. http://www.junit.org/.

[Reis04]
Reis, C. and Cartwright, R. Taming a professional IDE for the classroom. In *Proc. 35th SIGCSE Tech. Symp. Computer Science Education*, ACM, 2004, pp. 156-160.

[Shepard01]
Shepard, T., Lamb, M., and Kelly, D. More testing should be taught. *Communications of the ACM*, 44(6): 103-108, June 2001.

[Storey03]
Storey, M.-A., Damian, D., Michaud, J., Myers, D., Mindel, M., German , D., Sanseverino, M., and Hargreaves, E. Improving the usability of Eclipse for novice programmers. In *Proc. 2003 OOPSLA Eclipse Technology eXchange*

Workshop, ACM, 2003, pp. 35-39.

[XProgramming06]
XProgramming.com Software Downloads (see the "Unit Testing" section). http://www.xprogramming.com/software.htm.

Some Preliminary Rules of Engagement for Java

Joseph E. Hollingsworth
Computer Science Department
Indiana University Southeast
4201 Grant Line Road
New Albany, IN 47150-2158 USA

jholly@ius.edu
Phone: +1 812 941 2425
Fax: +1 812 941 2637
URL: http://homepages.ius.edu/JHOLLY

Bruce W. Weide
Department of Computer Science and Engineering
The Ohio State University
2015 Neil Avenue
Columbus, OH 43210-1277

weide.1@osu.edu
Phone: +1 614 292 1517
Fax: +1 614 292 2911
URL: http://www.cse.ohio-state.edu/~weide/

Abstract

We propose some "rules of engagement" for developing software in Java in order to achieve the following goals: simplified specification and reasoning when compared to alternatives such as JML, efficiency comparable to typical Java software, and the ability to make use of existing Java components (e.g., Swing). It has been said that the best place to start is at the beginning, and [Weide02, Harms91] tell us that the beginning has to be with the movement of data. If we do not get that aspect of software development right to begin with, then we are already fighting a loosing battle with respect to achieving the stated goals. Next comes the development of what we call "clean and safety net" foundational components which aid in building of larger scale applications [Hollingsworth00] that meet our goals. Finally, we need some additional rules that must be followed in order to achieve the stated goals.

Keywords

data movement, generics, move, transfer, program reasoning, clean and safety net components

Paper Category: technical paper
Emphasis: research

1. Introduction

Since the early 1990's the Reusable Software Research Group (RSRG) has advocated swapping [Harms91] as the primary means by which data should be moved within a sequential, imperative program. Only one research language (Resolve [Ogden94] created by RSRG) and no commercial languages (that we know of) support swapping as the built-in, primary method for moving data within a program. RSRG has proposed disciplines for Resolve/Ada [Hollingsworth92-2] and Resolve/C++ [Hollingsworth94] that permit building software in those languages using the swapping paradigm.

The contribution of this paper is that it proposes a data movement paradigm for Java, using existing built-in Java constructs, along with a preliminary set of "safety net" foundational components, and some preliminary "rules for engagement" for developing Java programs, all intended to reduce the cognitive load with respect to reasoning about Java programs and object references inherent in them, and still be efficient.

Section 4.1 discusses why retrofitting swapping into Java (as was done in Resolve/Ada, and Resolve/C++) cannot be done easily, and our choice as the next best alternative appears in Section 4.2. Section 4.3 mentions (without detail) some of the clean and safety net components, while Section 4.4 introduces some of the preliminary rules for engagement for using Java in a disciplined manner.

2. The Problem

How to design and implement software written in Java to improve ease of reasoning and specification, efficiency, and the ability to make use of existing Java components if so desired. Note: "ease" is almost always directly dependent on the difficulty associated with computing the solution to the problem, therefore "ease" is a relative term. That is, we are not claiming everything is going to be "easy".

3. The Position

By building on previous RSRG work, we have developed a prototype approach that addresses the above problem with a system of rules that even hard core Java hackers can embrace.

4. Justification

4.1 In Java, "Move" or "Transfer" Data, but Do Not "Swap" it

[Weide02] explores the various ways in which data can be moved within a program, i.e., to address the *data movement problem*. That paper also proposes some *data movement evaluation criteria* for evaluating these different methods for moving data. We are not going to rehash that work here, rather, we plan to use its results to pick the best data movement paradigm for Java programs. In this section, we first discuss why we are not picking the swapping data movement paradigm for Java, which by the data movement evaluation criteria turns out to be an excellent choice for many commercially available, sequential, imperative languages. Next we propose that data be "moved" or "transferred" in Java. This *move* approach is not new - it is described in [Weide02], and by the evaluation criteria it is rated the next best choice for moving data within a program.

4.2 In Java, We Are *Not* Going to Swap

Anyone who has been even peripherally acquainted with RSRG's research knows that swapping of data has been (and continues to be) at the core of the Resolve software discipline and approach. That we are *not* choosing swapping for Java must come as a great surprise. But do not be fooled. The heart of the problem within sequential, imperative programs is how best to move data. The answer is swapping in the Resolve research language and for many commercial languages (except Java). We are not married to swapping. We are dedicated to solving the data movement problem, and if that means choosing a method other than swapping, then so be it.

Because of the way in which Java is defined, swapping does not turn out to be the most effective method for moving data. The roots of this data movement problem in Java lie with the fact that all non-scalar type objects are implemented by using a reference variable to store the address to the object data (as was done in Modula), but no direct access is given to where this reference is stored. By direct access, we mean, a Java software developer cannot gain access to the reference variable in order to change the existing reference stored there so that the variable can reference a different object. If one *could* access where the references are stored in Java reference variables, then one should be easily able to implement the swapping paradigm.

A folk theorem of computing is that every problem can be solved by adding a level of indirection [**Weide01**]. So why not stick with swapping and implement all objects with a second level of indirection as is mentioned in [**Sridhar02**]? That is, when creating a new object class C, have the only data member inside C be a reference to the actual data that needs to be stored by object instances of C. Then to swap the data between two object instances of C, say A and B, one would just call an operation exported from C which would have access to A's and B's data members (both of which are references to the real data) and and just swap them. The original Java variable references for A and B would remain the same. (As a side note, this double level of indirection is often referred to as a *handle*, and is used by some operating systems to permit the operating system to do memory compaction on the fly while the program remains in an execution state.) Why not choose this approach? Every dereference of an object now requires two dereferences, and as was seen in [**Hollingsworth00**] actual commercial systems can be highly layered. Thus when a dereference is made at a high level this dereference might cause a cascading of dereferences down through the layers to the bottom layer. Performing two dereferences for each access of an object at each level of a highly layered system most certainly will cause a performance hit. We have not measured this effect because there is a viable alternative to swapping; we leave this for future work.

Ruling out two levels of indirection leaves us with trying to implement swapping in Java by using the built in language constructs, e.g., assignment, parameter passing, returning of values, etc.

- Call-by-reference - *If* Java supported call-by-reference, then every class C could export an operation which would take two objects from that class and do a straightforward three line swap using a locally declared temporary variable. The call to such an operation might look like: `x.swap(y);` After the call, x would reference y's data, and vice versa.
- Return multiple values - *If* Java supported the returning of more than one value, then one might construct an operation which would take two objects and return those two objects in "reverse" order. The call to such an operation might look like: `y, x = swap(x, y);`
- Preprocessor magic - *If* Java supported a preprocessor then one might be able to use *preprocessor magic* (as it is known in the C/C++ world) to implement swapping.

Our proposal is to implement data movement by using Java's assignment statement in combination with the recently introduced generic class and interface capability. We found in [**Hollingsworth00**] that dealing with a piece of data gives rise to one of two situations: 1) we either momentarily need to hand off an object to some other operation (which implements some algorithm that uses the data, and possibly changes it); or 2) we need to put it in some container for safe keeping for later use. For both situations, Java is well suited with its call-by-value for small and large objects. Why? Because at most, all that is required for passing a scalar is the (small) value itself, and all that is required for passing large objects is the (small) variable reference.

But what about ease of specification and reasoning? The problem is visible aliasing, i.e., aliased references to mutable objects, so that changes to an object through one reference are visible through another reference that is not mentioned in the statement that changes the object. We want to drive the number of references to each object to the bare minimum, i.e., one. Think of this as a program wide invariant, similar to a loop invariant, where there might be short times when we need to have multiple visible aliases, but after that need is satisfied, we go back to our invariant of just one reference (just like reestablishing the loop invariant). That is, if in a particular part of a Java program, we have a Java variable reference to an object (and its value), and we know (because of our program-wide invariant) that no other part of the program aliases this object, then we can reason with certainty and confidence about the before and after values of the object when working with it. We can do this without introducing the complications of modeling or reasoning about references [**Weide01**].

When handing off an object - In the situation where our sequential program needs to hand off the object to another operation, there will be two references in existence momentarily, one held by the caller and one held by the callee. For now, we are not going to allow the callee to make and save a copy of the reference, so that when the callee finishes, its reference automatically gets eliminated. Later we will discuss what has to be done if the callee needs to save a copy of the reference.

When saving an object for safe keeping - When inserting an object into a container object for safe keeping and for later use, the container gets and stores the reference, and simultaneously we eliminate the reference stored in the caller by replacing it with 'null'. Below (in Figure 1) is some sample code using a Queue for a container:

Client of Queue	IQueue.java

```
import queue.*;

void Op1 ()
{
   SomeMutableOrImmutableObject x = new SomeMutableOrImmutableObject()
   IQueue<SomeMutableOrImmutableObject> q1 = new
Queue1<SomeMutableOrImmutableObject>();

   x.modify(...);
   x = q1.enqueue(x); // returns null
   ...
}
```

```
package queue;

public interface IQueue<E> extends
Cloneable {
   public void clear();
   public IQueue<E> clone();
   public E dequeue();
   public E enqueue(E value);
   public boolean equals (Object
other);
   public E peek();
   public int size();
   public String toString();
}
```

Figure 1.

All container components are implemented so that:

- when the reference goes into the container, 'null' gets returned and the calling operation assigns this result to the reference variable that was passed to the container's insert operation;
- when the reference comes out of the container, no reference to it is maintained within the container, and the calling operation assigns the returned reference to its own local reference variable

4.3 A Set of Clean, and Safety Net Components

This part of the effort involves encapsulating, into a family of well designed, understandable, and carefully implemented software components, support for clean components that encapsulate the most important uses of indirection with simple contract specifications; and safety-net components to handle all residual uses of indirection not otherwise covered. There is not enough room to describe them here. A demonstration will be provided at the workshop, however. Details basically follow [**Hollingsworth92-1**, **Hollingsworth92-2**].

4.4 Rules for Engagement

In the past it has been called a "discipline". It's time for a new name, and that name is "rules for engagement". Think of it as how we are going to engage the language (Java in this case) in a way that helps us achieve our goals. The incomplete list of rules that follow are not in any particular order of importance. One must remember that it is allowable to break any of these rules for engagement; however, one needs a very good reason to do so, and also needs to be prepared to pay the consequences.

- Use value-based modeling and specification.
- Make 'null' a member of the value space for all object models. That way when an object is assigned 'null' it has a legal value for its type.
- Use the "generic" construct so that the compiler can aid in helping identify type mismatches in client code at compile time.
- Use the "interface" construct to capture the abstract idea of a component, use one or more "class" constructs to implement the interface.
- Build all container components by layering on the clean and safety-net components mentioned in Section 4.3.
- Implement 'clone' so that it makes a true deep copy.
- Implement 'toString' so that it produces a string representing the abstract value of the object.
- Implement a 'clear' operation to reset the object's abstract state to be exactly as if the object had just been created.
- Implement a 'terminate' operation to be called when the object is no longer of any use. This is not the same as 'clear', because after 'clear' is called the object can continue to be used.
- At declaration time, assign a value to an object reference. One might use 'new' and call the object's constructor, or one might 'clone' a value, or one might set the reference to 'null'. There are many options, but one way or the other, the object reference must get assigned a value at declaration time.
- Move data around as is described in Section 4.2.
- At all times drive the number of references to bare minimum by moving/transferring references rather than copying them.
- If you make use of components that do not follow these rules of engagement, be aware that operations such as 'clone', 'clear', etc., will probably not work as advertised.

5. Related Work

We have made reference to other work throughout the paper.

6. Conclusion

As was mentioned in Section 3 (The Position), we have developed a prototype approach that simplifies reasoning and specification about Java programs without sacrificing efficiency or existing libraries. We can think of no better way in which to make it stronger (or discredit it) than to hold it up for scrutiny by the attendees of the workshop. We will come prepared to show the clean and safety-net components, containers layered upon these components, and even a small GUI application developed using these components.

Acknowledgments

This work is supported in part by Indiana University by providing sabbatical release time for this research.

References

[Harms91]

Harms, D.E., and Weide, B.W., **"Copying and Swapping: Influences on the Design of Reusable Software Components"**, *IEEE Transactions on Software Engineering 17*, 5 (1991), 424-435.

[Hollingsworth92-1]
Hollingsworth, J.E., and Weide, B.W., "Engineering 'Unbounded' Reusable Ada Generics," *Proceedings 10th Annual National Conference on Ada Technology,* Arlington, VA, February 1992, 82-97. **[PDF]**

[Hollingsworth92-2]
Hollingsworth, J.E., *Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada*. Ph.D. thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1992. (If you obtain this document electronically, please read the file **README.txt** in the document's directory, then take all the files.) **[PDF; 12 files]**

[Hollingsworth94]
Hollingsworth, J.E., Sreerama, S., Weide, B.W., and Zhupanov, S., "RESOLVE Components in Ada and C++," *Software Engineering Notes 19,* 4 (October 1994), 53-63. **[PDF]**

[Hollingsworth00]
Hollingsworth, J.E., Blankenship, L., and Weide, B.W., **"Experience Report: Using RESOLVE/C++ for Commercial Software"**, *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*, 2000, ACM Press, 11-19.

[Ogden94]
Ogden, W.F., Sitaraman, M., Weide, B.W., and Zweben, S.H., "The RESOLVE Framework and Discipline- A Research Synopsis," *Software Engineering Notes 19,* 4 (October 1994), 23-28. **[PDF]**

[Sridhar02]
Sridhar, N., Weide, B.W., and Bucci, P., "Service Facilities: Extending Abstract Factories to Decouple Advanced Dependencies", in C. Gacek, ed., Software Reuse: Methods, Techniques, and Tools (Proceedings Seventh International Conference on Software Reuse), Springer-Verlag LNCS 2319, 2002, pp. 309-326. **[PDF]**

[Weide01]
Weide, B.W., and Heym, W.D., "Specification and Verification with References", Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems, October 2001, **http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001**.

[Weide02]
Weide, B.W., Pike, S.M, Heym, W.D. "Why Swapping?", *Proceedings of the RESOLVE Workshop 2002*, Columbus, OH, June 2002, **Technical Report TR-02-11**, Dept. of Computer Science, Virginia Tech, Blacksburg, VA, June 2002, 72-78.

Automation of Verification Condition Generation for a Verifying Compiler

Heather Keown
Dept. of Computer Science
Clemson University
220 McAdams Hall
Clemson SC 29634 USA

hkeown@cs.clemson.edu
Phone: (864)-656-2840
Fax: (864)-656-0145
URL: http://people.clemson.edu/~hkeown/

Abstract

One component of a verifying compiler is a verification condition generator. The generator will take a Resolve component (with suitable specifications and implementations) as inputs and output one or more assertions that if proved by a theorem prover, will verify that the component is correct. The objective of this paper is to examine two ways (goal-oriented and tabular reasoning) of creating the assertions.

Keywords

Verification Automation, goal-oriented Method, tabular reasoning

Paper Category: position paper
Emphasis: research

1. Introduction

The Resolve verifier will create assertions that when proved by a theorem prover, will prove the code is valid. Two methods for creating the assertions have been discussed in the Resolve literature: a goal-oriented approach [Odgen00] and a tabular reasoning method [Sitaraman00]. This paper is intended to initiate a discussion of which way is simpler and more appropriate to automate for the Resolve verifier.

2. The Problem

Will the goal-oriented approach be simplest to automate? Are there any major obstacles to automating the goal-oriented approach? Would the tabular reasoning approach be just as simple to automate? Are there any major obstacles in automating the tabular reasoning approach? Should the output be created only for machines or also for a human to read?

3. The Position

The Resolve verifier is to be based on a template similar to the one for the current Resolve to Java translator. So the verifier, for example, will use a visitor pattern to walk the abstract syntax trees to generate assertions. This implementation decision, fortunately, might have no special impact on how the conditions should be generated.

It is also possible that the verification condition generation is equally easy for both approaches, but the differences arise when the conditions are proved mechanically later by a prover or when the conditions are used by programmers as debugging aids.

In the beginning stages of the implementation, it appears that the goal-oriented approach is more easily automated than the tabular reasoning approach.

4. Justification

The example in figure 1 will be used to demonstrate the differences:

```
Assume true;

        If J > I then
                I:=:J
        end If;

        If K > I then
                I:=:K
        end If;

Confirm I >= J ^ I >= K;
```

Figure 1. An Example Assertive Code

4.1 Tabular Reasoning Approach

The tabular reasoning approach forms as its name implies a table. The table is split into states and shows how the variables change by numbering them in each state. As seen in Figure 2, the table is formed to include the assumptions, requirements, and path conditions. To implement this approach, the verifier will walk the ADT in the same direction and pattern as does the Resolve compiler. The assumption clauses are easily retrieved from the specification of each procedure. It appears that the most difficult part of the implementation will involve numbering the variables. Also, eliminating unnecessary statements may prove to be challenging, but is not an absolutely necessary step (except that then it's left to the prover).

State	Path Conditions	Assumes	Requires
0		True	
	`If J > I then`		
1	J1 > I1	J1=J0 and I1 = I0 and K1=K0	
	`I:=:J`		
2	J1 > I1	I2=J1 and J2=I1 and K2=K1	
	`End;`		
3.1	J1 > I1	I3=K2 and K3=I2 and J3=J2	
3.2	~(J1 > I1)	I3=I0 and K3=K0and J3=J0	

	If K > I **then**		
4	K4 > I4	J4=J3 and I4 = I3 and K4=K3	
	I:=:K		
5	K4 > I4	J5=J4 and I5 = K4 and K5=I4	
	End;		
6.1	K4 > I4	J6=J5 and I6 = I5 and K6=K5	I6 >= J6 ^ I6 >= K6
6.2	~(K4 > I4)	J6=J3 and I6 = I3 and K6=K3	I6 >= J6 ^ I6 >= K6

Figure 2. Reasoning Table

The final assertions to be proved by the tabular method are as follows:

1.(((J1 > I1) => (J1=J0 and I1 = I0 and K1=K0)) and ((J1 > I1) => (I2=J1 and J2=I1 and K2=K1)) ^ ((J1 > I1) => (I3=K2 and K3=I2 and J3=J2)) and (~(J1 > I1) => (I3=I0 and K3=K0and J3=J0)) and ((K4 > I4) => (J4=J3 and I4 = I3 and K4=K3)) and ((K4 > I4) => (J5=J4 and I5 = K4 and K5=I4)) and ((K4 > I4) => (J6=J5 and I6 = I5 and K6=K5))) => (I6 >= J6 and I6)

2.(((J1 > I1) => (J1=J0 and I1 = I0 and K1=K0)) and ((J1 > I1) => (I2=J1 and J2=I1 and K2=K1)) and ((J1 > I1) => (I3=K2 and K3=I2 and J3=J2)) and (~(J1 > I1) => (I3=I0 and K3=K0and J3=J0)) and ((K4 > I4) => (J4=J3 and I4 = I3 and K4=K3)) and ((K4 > I4) => (J5=J4 and I5 = K4 and K5=I4)) and (~(K4 > I4) => (J6=J3 and I6 = I3 and K6=K3))) => (I6 >= J6 ^ I6)

When using the tabular approach, there is no need to generate new names for variables that are not affected by a certain statement. If this aspect is mechanized, then several unnecessary names would disappear. After this simplification, there might not be much difference between the two approaches in terms of the assumptions generated to complete the proofs.

4.2 Goal-oriented Reasoning Approach

The goal-oriented approach forms an assertion working "backwards." By applying the appropriate rule in each step, the assertion changes under the directions of the rule. The goal-oriented approach seems fairly simple to implement. For each step a rule will be applied to modify the assertion. The most difficult part of this implementation appears to be based on the difficulty of the rules. As seen in Figure 3, this approach may seem more complicated, but seemingly more machine processable.

Step 1: Apply If_Then Rule	

A. Context/Assume true;
 	
If J > I **then**
I:=:J
end If;
 	
Assume K > I
I:=:K
 	
Confirm I >= J ^ I >= K;
	B. Context/Assume true;
	
	If J > I **then**
	I:=:J
	end If;
	
	Assume ~(K > I)
	
	Confirm I >= J ^ I >= K;

Apply Swap Rule:
2A. Context/Assume true;
 	
If J > I **then**
I:=:J
end If;
 	
Assume K > I
Confirm K >= J ^ K >= I;
	Apply Assume Rule:
	2B. Context/Assume true;
	
	If J > I **then**
	I:=:J
	end If;
	
	Confirm ~(K > I) =>
	(I >= J ^ I >= K);

Apply Swap Rule:
3A. Context/Assume true;
 	
If J > I **then**
I:=:J
end If;
 	
Confirm (K > I) =>
(K >= J ^ K >= I);	

Step 2: Apply If_Then Statement			
Context/Assume true; **Assume** J > I I:=:J **Confirm** (K > I) => (K >= J ^ K >= I);	**Context/Assume** true; **Assume** ~(J > I) I:=:J **Confirm** (K > I) => (K >= J ^ K >= I);	**Context/Assume** true; **Assume** J > I I:=:J **Confirm** ~(K > I) => (I >= J ^ I >= K);	**Context/Assume** true; **Assume** ~(J > I) I:=:J **Confirm** ~(K > I) => (I >= J ^ I >= K);
Apply Swap Rule: **Context/Assume** true; **Assume** J > I **Confirm** (K > J) => (K >= I ^ K >= J);	Apply Assume Rule: **Context/Assume** true; **Confirm** ~(J > I) => ((K > I) => (K >= J ^ K >= I));	Apply Swap Rule: **Context/Assume** true; **Assume** J > I **Confirm** ~(K > J) => (J >= I ^ J >= K);	Apply Assume Rule: **Context/Assume** true; **Confirm** ~(J > I) => (~(K > I) => (I >= J ^ I >= K));
Apply Assume Rule: **Context/Assume** true; **Confirm** (J > I) => ((K > J) => (K >= I ^ K >= J));	Apply Assume Rule: **Context/** **Confirm true => (~(J > I) => ((K > I) => (K >= J ^ K >= I)));	Apply Assume Rule: **Context/Assume** true; **Confirm** (J > I) => (~(K > J) => (J >= I ^ J >= K));	Apply Assume Rule: **Context/** **Confirm true => (~(J > I) => (~(K > I) => (I >= J ^ I >= K)));

Apply Assume Rule: **Context/** **Confirm** true => ((J > I) => ((K > J) => (K >= I ^ K >= J)));	Apply Confirm Rule: **Context/** true => (~(J > I) => ((K > I) => (K >= J ^ K >= I)));	Apply Assume Rule: **Context/** **Confirm** true => ((J > I) => (~(K > J) => (J >= I ^ J >= K)));	Apply ConfirmRule: **Context/** true => (~(J > I) => (~(K > I) => (I >= J ^ I >= K)));
Apply Confirm Rule: **Context/** true => ((J > I) => ((K > J) => (K >= I ^ K >= J)));		Apply Confirm Rule: **Context/** true => ((J > I) => (~(K > J) => (J >= I ^ J >= K)));	

Figure 3. An application of the goal-oriented approach

The final assertions from the goal-oriented Approach to be proved are as follows:

1.Context/ true => ((J > I) => ((K > J) => (K >= I ^ K >= J)));

2.Context/ true => (~(J > I) => ((K > I) => (K >= J ^ K >= I)));

3.Context/ true => ((J > I) => (~(K > J) => (J >= I ^ J >= K)));

4.Context/ true => (~(J > I) => (~(K > I) => (I >= J ^ I >= K)));

An advantage of the goal-oriented approach would be only in the cases where the goals that need to be proved are weaker than what can be proved

5. Related Work

There is no comparable verification condition generator in the literature that is similar to the one outlined in this paper.

6. Conclusion

The objective of this paper is to initiate a discussion on automation of verification condition generation and proofs at the workshop. At the time of the workshop, we hope to have examples of automatically-generated assertions for discussion.

Acknowledgments

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

I thank Dr. Murali Sitaraman for his help in completing this paper.

References

[Odgen00]
W. F. Ogden, The Proper Conceptualization of Data Structures, Computer and Info. Science, The Ohio State University, 2000.

[Sitaraman00]
Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W. D., Pike, S. M., and Hollingsworth, J. E. 2000. Reasoning about Software-Component Behavior. In Proceedings of the 6th international Conerence on Software Reuse: Advances in Software Reusability (June 27 - 29, 2000). W. B. Frakes, Ed. Lecture Notes In Computer Science, vol. 1844. Springer-Verlag, London, 266-283.

Software Verification Is Not Dead, But It Needs a New Way to Express Mathematics

Joan Krone
Denison University
krone@denison.edu

William F. Ogden
Ohio State University
ogden@cse.ohio-state.edu

Abstract

The RESOLVE vision of building software includes a library of components that satisfy two important criteria:?each component must have mathematical specifications, and each implementation for a given component must be certified to be correct.? Both of these criteria require systemic support for writing mathematics.? Our position is that the mechanisms for expressing mathematical specifications in typical verification systems are insufficiently powerful to support general program verification.? In addition, we propose that the tool to achieve software verification should be designed as a relatively powerful proof check, rather than as a general theorem prover.

Keywords

Verification, mathematics, syntax, semantics

1. Introduction and Related Work

The RESOLVE philosophy includes the thesis that software components should be certified to be correct before they become part of a software system.? Of course, some may think that carrying out an elaborate scheme for testing the software is adequate for certification, while others would like to insist on applying proof rules from a formal proof system in order to put a stamp of approval on any given component.

Because attempts at formal verification have been around for decades without making a significant impact on how software is written or used, many members of the community have lost hope that formal verification can be a reality.? Indeed, we know that the question of whether two given programs are equivalent is an undecidable problem, so if we consider a specification and an implementation as two programs we need to compare, hope for automated verification vanishes.

So the question of finding a way to address automated verification in the RESOLVE sense (a realization satisfies the concept for which it is written) presents many challenges, not the least of which is the need to articulate exactly what we hope to be able to do.

2. The Problem

2.1? The Context for Articulating the Verification Challenge

Over a period of nearly two decades a RESOLVE vision of software has evolved from the idea of developing a language that supports formal specifications with multiple implementations, to a vision of designing software components for a library to be used and added to by programmers.? One mainstay among many guidelines for these software systems is that no component should be permitted into the library unless that component has been certified as correct.

The plan is that certification can be to a large extent an automatable task, based on a system of proof rules which, when applied to the code, generate mathematical clauses equivalent to the correctness of the program.? Of course, once those clauses have been generated, the question of who or what will deal with proving those clauses has remained unanswered.? A few people have looked at theorem provers hoping to find one capable of handling the multi-theory programs in the RESOLVE style, only to find that existing theorem provers, usually based on induction, are aimed primarily at single level programs written about integers.? Such provers fail to accommodate abstraction and modularity, both of which are essential to RESOLVE.

2.2 The Current State of Verification

At present there are automatable proof rules ready to apply to the simple constructs in the RESOLVE language, and there are drafts of proof rules for the more complex constructs, such as realizations and facilities.? Relational semantics have been developed to permit the establishment of soundness and relative completeness for the proof rules.? The most up to date proof rules deal not only with functional correctness, but with performance as well.

It has been proven that a mechanism in the language for supporting auxiliary constituents is a necessity for completeness, and equally importantly that it is always possible to find an auxiliary constituent for writing a correspondence from a correct implementation to its concept.

2.3 The Challenge

With proof rules to generate the clauses necessary for proving correctness, but the absence of a theorem prover to process those clauses, we face the problem of either requiring a human to step in and prove the clauses or finding a mechanical way

to process those clauses.? We know from the experience of applying proof rules to small programs that the clauses generated can quickly grow to complicated forms which, although a human may be able to simplify the clauses and eliminate large portions of the forms, present a daunting challenge for mechanization.

One of those challenges is that of figuring out how to simplify clauses as they are generated.? Such questions as when simplification should take place and what simplification forms should be chosen do not have obvious answers.? Nevertheless, those questions need to be addressed with the aim of having our clause generators produce forms that are reasonable to process.? The literature has many accounts of ways various theorem provers have addressed the issue of simplification, but we have found none that seem to fit well with the multi-theory nature of RESOLVE.

A more interesting challenge, calling for a new way to look at verification is that of looking at the problem of proving correctness, not as a theorem proving exercise, but as a proof checking exercise.? This means that we consider automating the process of checking proofs, rather than creating proofs, or possibly some combination of the two.

3. Our Position and Justification

3.1 Math Units

Our position is that in addition to concepts, profiles, realizations, and facilities, RESOLVE needs a mechanism for describing mathematical theories.? The syntax must support the process of finding particular properties of any given mathematical type and checking to see that those properties have been used correctly in proofs.? We propose that the language will have a collection of theories built in, including set theory, which underlies all other theories, and other basic mathematical theories such as natural numbers, integers, and reals.

Not surprisingly, the mere presence of such a collection does not solve the problem of how those theories can be used effectively by the verification system.? Many new questions arise with regard to organization and the differences between mathematical material and programming material.

For example, one point of agreement among RESOLVE proponents is that the language should support strong type checking.? There may be some controversy as to when one kind of number can be used in a mixed computation with another (such as adding an integer to a real number), but for the most part, there is agreement that type checking should be done by name.? In any case, rules for numerical mixing can be worked out so that syntactic checks sufficiently sort out what may and may not be done.

Mathematical type checking presents some additional challenges.? Certainly, we want to allow mathematicians and others to write mathematics as closely to the way they are used to as possible.? This means that most arithmetic symbols must be overloaded, and rules for when it is all right to do such overloading must be worked out both for convenience of expression and correctness of semantic interpretation.

For example, if we have an assertion that x = a + b, and the *a* is real, while the *b* is an integer, then it should be assumed that the + sign represents real addition, rather than integer addition.? In the world of mathematics it is taken for granted that such mixing of types is acceptable, and so the rules we develop for overloading will need to be more liberal than those for programming.? We need to build into our mathematical typing the assumption that every natural number is an integer, every integer is rational, every rational is real, and every real is also complex without any need to explicitly cast.

Moreover, abstraction complicates the efforts at mathematical type inferencing.? To decide whether a particular operation can be applied to a given object, it must be possible to determine what the domain of the operation is, as well as the type of the object.? This means we need some notion of type compatibility different from that used in programming.

To get an idea of what a math unit might look like, we include a short example here showing part of what would be necessary for supporting binary relations.? It is not our intent to explain the details of the example, but rather to illustrate our position that we need syntactic constructs for putting mathematics into the RESOLVE library and to show one example of what such syntax might look like.? The pr?cis includes only properties and definitions, but no proofs, while a complete math unit includes the proofs as well.? The thinking is that for proof checking, once a math unit (with proofs) has been entered into the library, a proof checker?then needs only the properties that have been proven in order to apply them to its proof checking task.

```
Precis Basic_Binary_Relation_Theory;

    Def. Is_Reflexive( ρ: (D: Set)×D→𝔹 ): 𝔹 = ( ∀ x: D, x ρ x );

    Def. Is_Transitive( ρ: (D: Set)×D→𝔹 ): 𝔹 = ( ∀ x, y, z: D, if x ρ y and y ρ z, then
                                                                        x ρ z );

    Def. Is_Symmetric( ρ: (D: Set)×D→𝔹 ): 𝔹 = ( ∀ x, y: D, if x ρ y, then y ρ x );

    Def. Is_Antisymmetric( ρ: (D: Set)×D→𝔹 ): 𝔹 = ( ∀ x, y: D, if x ρ y and y ρ x, then
                                                                        x = y );

    Def. Is_Asymmetric( ρ: (D: Set)×D→𝔹 ): 𝔹 = ( ∀ x, y: D, if x ρ y, then ¬ y ρ x );

    Def. Is_Irreflexive( ρ: (D: Set)×D→𝔹 ): 𝔹 = ( ∀ x: D, ¬ x ρ x );

    Theorem Rln1: ∀ D: Set, ∀ ρ: D×D→𝔹, if Is_Transitive( ρ ), then
                                                Is_Irreflexive( ρ ) iff Is_Asymmetric( ρ );

    Def. Is_Total( ρ: (D: Set)×D→𝔹 ): 𝔹 = ( ∀ x, y: D, x ρ y or y ρ x );

    Theorem Rln2: ∀ D: Set, ∀ ρ: D×D→ 𝔹, if Is_Total( ρ ) then Is_Reflexive( ρ );
            ⋮
end Basic_Binary_Relation_Theory;
```

3.2 Proof Checking as a Replacement for Theorem Proving

Our position is that the math units described can be used for checking proofs submitted for new theories identified by designers of programs.? For example, in writing a prioritizer template as the specifications for sorting we might put in a definition for the total entry count in an entry keeper, something that is not included in the built in basic mathematical theories, but that makes writing the requires and ensures clauses simple and easy to read.

Similarly, we may want to add not only a definition, but possibly a lemma or theorem related to the particular component we are writing, also not in the built in mathematics, but useful to a proof checker.? For example, when doing an enumerated set template in which the type family, enumeration, is a record with one component being the size as a natural number and the other component a mapping from positive natural numbers to the elements in the sets under question, it would be convenient to include a definition, Set_of, for designating the set of elements for this type and a lemma stating that the cardinality of that set is the size in the record constituent.

We propose that when a new lemma is written in a component, the programmer should supply a proof, which can then be used in the verification process.? Of course, before such a component can be entered into the RESOLVE library, among other things, the proof supplied by the programmer must be checked.? This is quite different from asking that the proof be automatically generated by the verification system.

4. Conclusion

Once some built-in math units and precis have been placed in the library, their proofs completed, additional math units may be needed as new concepts and realizations are written.? In that case, since the new math units will be based on existing library units, their proofs will refer to the properties of particular library pr?cis, and so new proofs can be checked against properties of existing precis.? Designers of new math units will put on their uses lists the names of the existing math units on which their new ones depend, so the proof checker can match claims in the new units against what has already been proven.

This is a whole new approach to the challenge of verification.? In this approach we do not depend on automated proof creation, but rather we automate the process of checking proofs.? Of course, when proof rules are applied to programs and clauses are generated, the problem of when and how to simplify those clauses must be addressed.?Additionally, there is the challenge of how much an automated system can do with regard to supplying proof parts and then letting users know when human hints are helpful or necessary to keep the proof progressing.?

The idea is to have humans and the proof system work together with the math units to address the verification challenge.? Programmers supply new math units as needed.? The proof checker checks to see that the programmer supplied proofs are correct in the context of the library math units.

Of course, this approach depends on a well-defined syntax for expressing mathematical theories, no small matter.? Moreover,

for efficiency in applying these theories during the verification process, we need both rules of logic for our system and rules for type inferencing in our mathematical notation.

Issues in the Creation of an Automated Prover

Kimberly Roche
Clemson University
Clemson, SC 29634 USA

kroche@cs.clemson.edu

Abstract

This paper presents several possibilities of approaching the automation of proof generation. Using basic heuristics, a theorem prover could be written capable of utilizing a library of tactics and sources and the strategy of resolution theorem proving to compose a set of step by step instructions for ultimate verification by the Coq Proof Assistant. This paper lays out design issues in the implementation of such a proof generator.

Keywords

Coq, proof, theorem proving, proof generator, heuristics, reasoning

Paper Category: position paper
Emphasis: education

1. Introduction

An integral part of the verifying compiler as presented in the paper by Kulczycki, Duckworth, Sitaraman, & Weide will be the Automated Prover [Kulczycki]. While a proof assistant, like Coq, may provide a powerful tool for evaluating and verifying proofs, it will lack the functionality to generate its own proofs [Coq]. The proof generation process would have to be automated within the compiler if Coq were to be used as a proof engine in conjuction with Resolve.

2. The Position

Some parts of the proof could be generated mechanically (such as the initial generation of assumptions) but for the rest of the proof, heuristics would have to be developed to guide the proof generation.

3. Justification

One of the goals of the Resolve research project is to have a utility for the verification of Resolve programs - a verifying compiler [Kulczycki]. This utility will have to perform two independent and very complex jobs: generation of a theorem to be proved from Resolve specifications and code by a "Verification Condition Generator", and creation of a proof of that theorem for soundness [Kulczycki]. While there is still debate about how to best implement the verification condition generator, once these conditions, or obligations, were defined for proving, the proof generator could go to work building a step by step proof and using Coq to evaluate it.

One of the first issues in the automation of this system is straightforward: how do we know which imports will be needed for a particular proof? Will there be a way that we can narrow the searchable references down in order to reduce running time? Producing too large a search space would create an unnecessary burden on the system, but failing to import all the necessary sources could mean the proof would fail altogether.

For many user-generated obligations, we can speculate that the proof will likely need to reference a recently verified theorem within the same package (as is the case in Figure 1). In this case, the modularity of the source and the grouping of similar theorems may become important, especially if the system were designed to intuitively apply the rules. In the attempt to resolve a proof, preference would most likely be given to utilizing local theorems and definitions first. Only when one scope has been exhausted would the search space be widened to include other assumptions and eventually whole math units.

Figure 1 presents the splice proof first introduced in the paper by Kulczycki, Duckworth, Sitaraman, & Weide [Kulczycki]. We will consider this a prototypal proof in our discussion.

```
Theorem neq_0_ge_gt : forall n m : nat,
    n <> 0 -> m >= n -> m > 0.

forall Q0 P0 P1 : string Entry * string Entry,
    |P0 Rem| >= |Q0 Rem| -> |Q0 Rem| <> 0 -> P1 = P0 -> |P1 Rem| > 0.
Proof.
intros.
replace P1 with P0.
apply (neq_0_ge_gt (|Q0 Rem|)(|P0 Rem|)).
assumption.
assumption.
Qed.
```

Figure 1. Proof of Splice Procedure on Lists for Coq

Of Coq's many proving "tactics", the initial method applied when the obligation takes the form of an inference is usually "intros". It applies deduction to assume the first half of an implied relationship, adding that statement to the "local context"

and forms a new obligation to be proven from the implied statement. This can be considered a standard step in the proof. Similarly, once the obligation has been reduced to the point that the only steps left to complete the proof are simple substitutions of already existing assumptions, the "auto" or "assumption" tactics can be called upon to do just that. But what is the point at which the "auto" feature of Coq can be called upon? Would the system attempt to use the "auto" tactic after every step? Or would there be a more sophisticated reasoning in place that could recognize a proof near completion?

The initial and final steps of the proof could in this way be mechanized. The real work of automation will occur after the local context has been formed but before the point at which the "auto" or "assumption" tactics can be invoked. Here, the system designer must find some mechanical method for "guiding" the Coq proof assistant.

For very basic obligations containing equalities, it may be possible for a simple substitution routine to be applied within the first stages of the proof. This will involve the matching and rewriting of variables. Certain other relationships (e.g. inequalities) may reference a set of behavioral rules entirely their own. Such rules would specify how a series of inequalities might be correctly rewritten. The automated prover would use these rules to search the local context for applicable assumptions.

The most complex part of the scenario, however, is the part that Coq cannot independently execute. User input is required to decide which theorems to apply and to guide the direction of the proof. While the opening and concluding steps of the proofs can be painlessly mechanized, implementing a higher order of intelligence in a verification system would require several important design choices to be made. When the program does not immediately see a local hypothesis it can apply to the obligation at hand, might it resort to applying each theorem from the imported list iteratively or would a complex set of heuristics come into play?

The automated prover will need to exhibit a level of artificial intelligence at this stage; the generation of proof tactics will not be strictly mechanical. The system will more than likely need to discern what the most appropriate next step is. For this, it might be necessary to hold a list of commonly applied tactics. This list could be specific to the semantic format of the given local context and obligation. The system would use heuristics to score each step in the list for probability based on the information at hand in the proof. The best candidate for the next step would be selected and, failing an ultimate solution using that path, the program would backtrack, apply the next best step, and so on. The bulk of the logic in this scenario would go into the development of the heuristics. This may involve matching the format of the obligation to the resolution offered by a theorem. For example, an obligation in the form "x > 0" will be matched to a list of theorems which conclude with an assumption in the form "x > 0".

One step may be common to all proofs. Before the system looks to apply any theorems, it will need to check the local context to see if a substitution or resolution is possible from the existing assumptions. This check will likely need to take place after every step in the proof.

To illustrate some of the complications in implementing an automated prover, we can examine a specific proof.

```
Confirm (|T| > 0) and (∀ T': Stack, F': Entry, (T = <F'> • T') => (|T'| < Max_Depth
and ... ) )

Proof |T'| < Max_Depth ;
   Goal |T| = | <F'> • T' | ;                 /* from the assumption */
   Goal |T| = | <F'> • ((T')Rev)Rev | ;       /* Thm. S10 from String Theory */
   Goal |T| = | (ext((T')Rev, F')Rev | ;      /* Defn. of Rev */
   Goal |T| = | (ext((T')Rev, F') | ;         /* Thm. S11 from String Theory */
   Goal |T| = suc( | (T')Rev | ) ;            /* Defn. of | | */
   Goal |T| = suc( | T' | ) ;                 /* Thm. S11 from String Theory */
   Goal |T| = | T' | + 1 ;                    /* Defn. of suc() */
   Goal |T| - 1 = | < Max_Depth | ;           /* Substitution for T' */
   Goal |T| <= | Max_Depth | ;                /* True by constraint on the Stack
Template */
```

Figure 2. Proof of |T'|

The first step follows from the assumption but obligates us to find a way to deduce |T'| from its relationship with T. Finding nothing helpful in the assumptions, the prover would ultimately call upon String Theory. Here, we find ourselve limited in the theorems we can apply to a string in the form <x> • a. The ()Rev operation allows us to reverse this string and apply the ext() operation as per the definition of ()Rev. The proof follows neatly from there.

This example raises an interesting question. Are there likely to be more theorems which can be proven in a single series of steps or which can be proven in multiple ways? It seems that in the case of **Figure 2**, after an initial creative leap, we are relatively limited in the number of theorems we can apply and in this way part of the proof guides itself. A strict and concise library of theorems might of primary importance in this way.

4. Related Work

The Resolve research group at Clemson University is currently looking into ways to incorporate the Coq proof assistant into the Resolve system and automate proof generation for Coq.

5. Conclusion

The design of a automated prover will be an essential part of the future development of the Resolve software package. Using an existing proof checker would not only make this task easier but would provide a basis for discussion about the design intricacies of a Resolve proof generator. How would an intelligent, efficient proof generator be constructed?

References

[Kulczycki]
Gregory Kulczycki, Scott Duckworth, Murali Sitaraman, and Bruce W. Weide. Abstracting Pointers for a Verifying Compiler.

[Coq]
The Coq Proof Assistant Reference Manual. Web page available at: **http://coq.inria.fr/doc/main.html**.

Mechanical Verification of Parallel Programs

Murali Sitaraman
Computer Science
Clemson University
Clemson, SC, 29634-0974 USA

murali@cs.clemson.edu
Phone: +1 864 656 3444
Fax: +1 864 656 0145
URL: http://www.cs.clemson.edu/~resolve

Abstract

Verification of correctness of parallel programs is an open research problem. In proving correctness of such programs, it is necessary to show that the processes are not simultaneously modifying the same variable. Even if a shared variable is not modified, it is often necessary to show that the variable is preserved, i.e., its value never changes. This position paper examines a few simple exercises to understand the kinds of problems that must be addressed in the verification of any parallel program, in general, and point-based programs, in particular. It postulates possible proof rules for mechanical verification.

Keywords

Parallel constructs, proof of correctness, pointers, preserves parameter mode

Paper Category: technical paper
Emphasis: research

1. Introduction

Verification of correctness of parallel programs is an important problem. Resolve includes a **preserves** parameter mode especially for use in parallel programs. A procedure that claims to preserve a parameter must guarantee that the parameter's value never changes–not even temporarily— during execution of the procedure. This paper considers proving correctness of programs that use preserves parameter mode and uses this idea as a starting point towards development and verification of parallel programs in Resolve. The paper presents examples and example proof rules, hoping to stimulate discussions on these and related topics at the workshop.

2. The Problem

The problems addressed in this paper include verification of programs that use preserves modes in Resolve that, by design, facilitate parallelization, and verification of correctness of parallel programs with and without pointer behavior.

3. The Position

Research in construction and verification of parallel programs, where the goal is to speed up processing, can be a useful first step towards verification of more general concurrent programs. Moreover, understanding the preserves parameter mode (as opposed to the restores mode) may serve as a starting point.

4. Justification

This section considers several Resolve examples to justify the position.

Preserves Mode Parameters

Kulczycki, et al., have discussed formal specification and reasoning of correctness of an operation to splice or interleave two given lists [Kulczycki 05, Kulczycki 06]. They consider two versions of the Spice code. One version is based on a data abstraction for lists and another is based on pointers. The pointer-based version is reproduced below. For the purpose of this first section, the *preserves* parameter mode of Position p is the focus. The preserves mode is different from the restores mode in that the restores mode simply demands that the code ensure p = #p, whereas the preserves mode demands that p never changes. To show that p is preserved, it is not sufficient to show that p remains is the same after every step in the code. This is because if p is passed as a parameter to an operation that merely restores p, then there's no guarantee that p is preserved.

Definition Var Is_Reachable_in(hops: **N**; p, q: Location): **B** =
  $\text{Target}^{\text{hops}}(p) = q$ **and** "k: **N**, **if** $\text{Target}^{k}(p) = q$ **then** k ³ hops;

Definition Var Is_Reachable(p, q: Location): **B** = $k: **N** ' Is_Reachable_in(k, p, q);

Definition Var Distance(p, q: Location): **N** = ;

Definition Var Is_Info_Str(p, q: Location; a: Str(Info)): **B** =
  $n: N ' Is_Reachable_in(n, p, q) **and** a =;

```
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 ( "t: Location, if not Is_Reachable(#p, t) and not Is_Reachable(#q, t)
                                  then Target(t) = #Target(t) ) and
      ( "a, b, g: Str(Info), if Is_Info_Str(p, Void, a) and
                                  Is_Info_Str(#p, Void, b) and
                                  Is_Info_Str(#q, Void, g) then a ≤!≥ (b, g) );

Recursive procedure
   decreasing Distance(q, Void);

   Var r, s: Position;

   If (not At_Void(q)) then

      Relocate(r, p);

      Follow_Link(r);

      Relocate(s, q);

      Follow_Link(s);

      Redirect_Link(p, q);

      Redirect_Link(q, r);

      Splice(r, s);

      Clear(q);

   end;

end Splice;
```

Figure 1. Pointer-Based Splice Operation Specification and Code from [Kulczycki 05]

Proof Rules for Handling Preserves Mode Parameters

An obvious solution to the problem is to specify the Splice operation so that p is restored and confirm that p = #p at the end of the code. The general reason to use the preserves mode over restores mode is to facilitate parallelization. Before considering a parallel version of Splice, first we consider verification of code with preserves mode parameters. Suppose that **Preserve** is a special verification assertion. Then we can formulate straightforward rules to prove that variables are preserved.

Procedure declaration rule:

Context È {**Operation** P (**preserves** x: T);} / **Remember**; **Assume** pre; body; **Preserve** x; **Confirm** post;

Context È {**Operation** P (**preserves** x: T1; **updates** y: T2); **requires** pre; **ensures** post} / **Procedure** P (**preserves** x: T); body; **end** P;

Modified Procedure Call rule:

Context È {**Operation** P (**preserves** x: T);} / assertive_code; **Preserve** u; **Confirm** …

Context È {**Operation** P (**preserves** x: T1; **updates** y: T2); **requires** pre; **ensures** post;}/ assertive_code; P(u, v); **Preserve** u; **Confirm** Q;

Modified Swap Statement Rule:

Context/ assertive_code; **Preserve** x; **Confirm** Q [y -> z; z -> y];

Context/ assertive_code; y :=: z; **Preserve** x; **Confirm** Q;

A Parallel Version of Splice

Though it's hard to envision much to be gained by parallelizing Splice procedure, it offers a useful example. Shown below is an example. The specification of Splice is the same as in the sequential version. In the code, for correct working of the parallel code, it's necessary to establish that the parallel process do not affect or update the same variable. For this reason, we specify a set of positions affected by each process in an **affects** clause. Just as in the specification of an operation, all

"external" variables not mentioned in the ensures clause are assumed to be (and implicitly established to be) restored, all variables not listed in the affects clause must be "preserved" by each process.
In general, a formal verification system needs to and can establish this invariance using the **Preserve** assertion discussed in the last subsection. However, it may not be necessary to show that all variables are preserved, only that the ones referred to (but not necessarily updated) by the processes remain unchanged. For this purpose, it's useful to add a **refers to** clause for each process. In the Splice example, the position r is referred to by both processes, but is modified by neither.

```
Operation Splice (preserves p: Position; clears q: Position);
        requires …
        ensures …
Recursive Procedure
        decreasing …
        Var r, s: Position;
        If (not At_Void(q)) then
                r = p;
                s = q;
                Follow_Link (r);
                Follow_Link(s);
                Cobegin
                        Process P1: affects q, Target(p), Target(q); refers to r;
                                Retarget_Link(p, q);
                                Retarget_Link(q, r);
                                Clear(q);
                        Process P2: affects {t: Position  | Is_Reachable_in (1, r, t) or Is_Reachable_in(0, s, t)}; refers to
r;
                                Spice(r, s);
                end;
        end if;
end Splice;
```

Figure 2. A Parallel Version of Splice Code

A proof rule for verification using the affects and refers to clauses is given below. Note that the above parallelization of Splice will not work if Splice is specified to restore its parameter p, instead of preserving it.

A proof rule for verification of parallel code

Context/assertive_code; **Confirm** W1 Ç W2 = Æ;
Context/assertive_code; code_1; code_2; **Preserve** R1, R2; **Confirm** Q;

```
Context/assertive_code;
        Cobegin
                Process P1: affects W1; refers to R1; code_1;
                Process P2: affects W2; refers to R2; code_2;
        end;
        Confirm Q;
```

Additional Examples

Unlike the pointer-based version of Splice, the List-based version given in [Kulczycki 06] is not readily parallelizable because the participating lists are modified by the recursive List Splice procedure. So we consider below a variation. This parallel program is unlikely to make it any faster than the regular Splice, but it is an example of that shows how we can make the processes independent by trading off more space (extra List variable R here).

```
Operation Splice(updates P: List; clears Q: List);

    requires |Q.Prec| = 0 and |P.Rem| ≥ |Q.Rem|;

    ensures $a: Str(Entry) ' a ≤!≥ (#P.Rem, #Q.Rem) and

            P.Prec = #P.Prec  a and |P.Rem| = 0;

Recursive procedure Splice(updates P: List; clears Q: List);

    decreasing |Q.Rem|;

    Var E: Entry;

    Var L: List;


    If (Rem_Length(Q) /= 0) then

        Advance(P);
```

```
        Remove(E, Q);
        Swap_Remainders(P, R);
        Cobegin
            Process 1: affects E, P;
                Insert(E, P); Advance(P);
            Process 2: affects R, Q;
                Splice(R, Q);
        end;
        Swap_Remainders(P, R);
        Advance_to_End(P);
    end;
end;
```

Figure 3. A Parallel Version of List_Based Splice Code

An example with a loop is given below. In general, if operations called in parallel, such as Push and Pop in the following example, were non-trivial operations, then the parallelism would be useful. Notice that the code will work properly even if Pop(F, S) is done before Push(E, S_Flipped). But there's no need to prove all kinds of inter leavings, if the processes are independent.

```
Operation Flip (updates S: Stack);
        ensures S = #S^Rev;
Procedure
        Var E, F: Entry;
        If (Depth_of(S) ¹ 0) then
                Pop(E, S);
                While (Depth_of(S) ¹ 0)
                        maintaining #S = S_Flipped^Rev o <E> o S;
                        decreasing |S|;
                    do
                        Cobegin
                                Process P1: affects E, S_Flipped;
                                        Push(E, S_Flipped);
                                Process P2: affects F, S;
                                        Pop(F, S);
                        end;
                        E :=: F;
                end;
                Push(E, S_Flipped);
        end if;
        S :=: S_Flipped;
end Flip;
```

Figure 4. A Parallel Version of Stack Flip Code

We conclude the paper with a more typical array parallelization example. Other examples include situations where an array is referred to, but not affected by the processes.

```
        Facility SF is Stack_Template …
        Var Array (1..Max) of SF.Stack;

        Cobegin for I := 1 to Max
                    maintaining "J: N, if 1 £ J £ I, then A[J] = #A[J]^Rev;
          do Process (I): affects A[I];
                    Flip(A[I]);
        end;
```

Figure 5. Parallel Processing with an Array of Objects

5. Related Work

The most closely related work for this paper is previous efforts by Pike et al. on automatic parallelization of sequential Resolve programs [Pike 02]. Using static analysis, it is indeed possible to extract the parallelization inherent in some of the examples mechanically, but the general problem requires a linguistic solution. There is significant work in the literature in verification of parallel programs. However, much of that work is not directly related to the focus of this that is concerned with enhancing Resolve with parallel constructs and establishing Resolve programs to be correct. This paper is based on the

well-known idea of establishing processes that are mostly independent. Considerable research remains to be done to show correctness in the presence of shared variables.

6. Conclusion

Most of the ideas in this paper are exploratory in nature and the corresponding findings are preliminary. The objective of this paper is to stimulate a discussion on verification of parallel programs at the workshop.

Acknowledgments

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

References

[Kulczycki 05]

Kulczycki, G., Sitaraman, M., Weide, B. W., Rountev, A.: A Specification-Based Approach to Reasoning about Pointers. In: **SAVCBS Workshop at FSE 2005**. ACM Software Engineering Notes, Vol. 31, No. 2 (2005).

[Kulczycki 06]

Kulczycki, G., Duckworth, S., Sitaraman, M., and Weide, B. W.: **Abstracting Pointers for a Verifying Compiler**. Technical Report RSRG-06-01, Clemson University. (2006).

[Pike 02]

Scott M. Pike and Nigamanth Sridhar, "Early-Reply Components: Concurrent Execution with Sequential Reasoning" in *Proceedings of **ICSR7**. Software Reuse: Methods, Techniques, and Tools.* Springer-Verlag, **LNCS 2319**, pp. 46-61 (2002).

An overview of the Sulu programming language

Roy Patrick Tan
Dept. of Computer Science
Virginia Tech
660 McBryde Hall
Blacksburg, VA 24061-0106 USA
rtan@vt.edu

Abstract

Sulu is a object-oriented programming language that adopts many of the features of RESOLVE, providing an alternative mechanism for the embodiment of RESOLVE components. In this paper, I provide a short tutorial for programming in Sulu, and a discussion on some of the issues surrounding the design of the programming language.

1 Introduction

In the RESOLVE 2002 workshop, Weide's paper *Good News and Bad News About Software Engineering Practice* presented a pessimistic view of the impact of RESOLVE research. At one point, he wrote: "None of the RESOLVE innovations has had the slightest influence on CSTs [commercial software technologies], even via their inevitability." [**Weide02a**] Later, at the workshop itself, Dr. Weide presented a much more positive view of RESOLVE research; nonetheless the tone of the paper perhaps represents the frustration of the community over the relatively low impact of this research in the larger software engineering community.

I have often thought that this state of affairs is brought about partly by the lack of a programming language implementation that embodies the ideas espoused by RESOLVE, and the lack of a novel application for the ideas espoused by RESOLVE research. The Sulu programming language is partly an attempt at working towards alleviating this perceived lack.

To date, the most common vehicle for presenting RESOLVE components is to implement them in RESOLVE/C++. It is a testament to the flexibility of the C++ language, and the ingenuity of the community that people can program in the RESOLVE discipline using that language. But the idioms of RESOLVE/C++ is quite *alien* to regular C++ idioms. Showing RESOLVE-like components in a different programming language like Sulu may lessen the cognitive dissonance of looking at programs written with idioms different from the host programming language.

Sulu is not RESOLVE, but it is strongly influenced by it. In the same RESOLVE 2002 paper [**Weide02a**], Weide lists some of the ideas brought forth by the RESOLVE community. The following table shows how Sulu tries to incorporate those ideas into the language.

Idea	**How it is implemented in Sulu**
separating specifications from implementations	Like RESOLVE, Sulu has both Concepts that hold specifications, and Realizations that holds implementation details. It has syntactic slots for executable design-by-contract specifications. The specification language is influenced by JML, the Java Modeling Language. [**Leavens99**]
allowing for multiple interchangeable implementations of a single specification	The separation of concepts and realizations in Sulu makes this trivial. All realizations must have corresponding concepts.
having a standard set of "horizontal", general-purpose, domain-independent components such as lists, trees, maps, etc., in a component library	This is currently work in progress, with example components for stacks, lists, and binary trees already implemented.
templates are a useful composition mechanism	Sulu concepts and realizations both allow generic template parameters.
having value semantics is useful even for user-defined types	Sulu uses swapping as the main data movement operator, and thus has value semantics, but minimizes the need for deep copying.
reasoning about programs that use pointers/references is complicated and error-prone	Having value semantics, reasoning about pointers/references is limited only to when the programmer uses a pointer component.
problems related to storage management, such as memory leaks, are serious	The Sulu interpreter, being implemented in Java, uses the Java garbage collection mechanism to implement memory management. But in theory, a Sulu compiler can be built with no need for garbage collection at all.

Sulu is not RESOLVE, its syntax is divergent from the one presented in the Software Engineering Notes paper [**Sitaraman94**]. Most notably, Sulu is object-oriented. It uses what has become the standard obejct-oriented dot notation, and supports various forms of inheritance.

Additionally, the Sulu interpreter introduces an infrastructure that uses the formal specification to aid in automatically generating and executing unit tests. It is my hope that this infrastructure will prove to be a useful vehicle for research in automatic test-case generation.

2 An example: implementing a list

Perhaps the best way to give a feel for how you can program in Sulu is through an example. In this section we will implement a List component using two stacks.

Figure 1 illustrates our conceptual view of a list component. We imagine a List as two sequences (let's call them *leftSequence* and *rightSequence*), both starting out empty. We have two insert methods insertLeft and insertRight. The insertLeft method will insert an item to the end of leftSequence, and insertRight will insert an item at the beginning of the rightSequence. We also have a method removeLeft that removes the end of the leftSequence (provided that it is not empty), and similarly, a removeRight method. Finally, we also have an advance and retreat method which shifts elements from the leftSequence to rightSequence and vice versa.

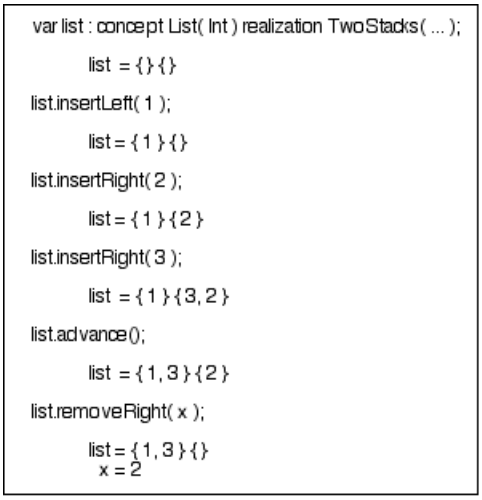

Figure 1. A conceptual view of a list component as two sequences.

Figure 2 is the code for the List concept. For brevity, not all the specification is included. You might notice that the concept is a template, it takes a parameter Item, which must *match* the concept Abstractable. We will discuss this idea of matching in the next section.

```
/******
 * A List is conceptually two sequences, a left one and a right one.
 ******/
concept List( Item: concept Abstractable() )
{

    model method leftSequence(): concept Sequence( concept MathObject() );

    model method rightSequence(): concept Sequence( concept MathObject() );

    method leftLength(): Int
        ensures leftLength == leftSequence().size();

    method insertLeft( elem: Item )
        ensures leftSequence() ==
                        old( leftSequence() ).append( elem.getMathObject() ) );

    method removeLeft( elem: Item );
        requires leftSequence().size() > 0
        ensures old( leftSequence() ) ==
                        leftSequence().append( elem.getMathObject() );

    method advance()
        requires rightSequence().size() > 0
        ensures {
            leftSequence() == old( leftSequence() ).append(
                                                    rightSequence().at(0) );
            rightSequence() == old( rightSequence() ).removeFirst();
        };

    method retreat();

    method rightLength(): Int;

    method insertRight( elem: Item );

    method removeRight( elem: Item );

    method length(): Int;
}
```

Figure 2. The List concept

Sulu specifications are *executable*. That is, there is an actual concept in Sulu called Sequence, and one or more realizations for it. We take care to separate the components that are in the specification world (components that inherit from MathObject) from components that are used in the implementation world (components that inherit from Object).

Methods that are prefixed by *model* are specification-only methods. Often, these methods are abstraction functions, mapping the internal state of the object to its representation in the specification world.

Figure 3 is a realization that implements a List concept using two stacks. Note that the type of Stack the realization uses is also left as a generic parameter. This provides great flexibility over what kinds of stack implementation to be used.

```
realization TwoStacks( ItemStack: concept Stack( Item ) )
    implements List( Item )
{
    /* instance variables */
    var left: ItemStack;
    var right: ItemStack;

    /* abstraction methods */
    model method leftSequence(): concept Sequence(Item) {
        leftSequence := left.getSequence().reverse();
    }

    model method rightSequence(): concept Sequence(Item) {
        rightSequence := right.getSequence();
    }

    /* methods */
    method insertLeft( elem: Item ) {
        left.push( elem );
    }

    method removeLeft( elem: Item ) {
        left.pop( elem );
    }

    method leftLength(): Int {
```

```
            leftLength := left.size();
        }
    }
```

Figure 3. This realization implements a List using two stacks

If you are familiar with Java, you may want to think of concepts as the equivalent of Java interfaces, and realizations as classes that implement the interface. Since concepts and realizations are generics, there is a need to "instantiate" the templates. This is done with the *class* construct in Sulu. Figure 4 illustrates how a List of String objects may be created and used.

```
// create a StringList class of String objects, using the TwoStack realization
// which in turn uses a linked-list based stack.
class StringList extends concept List(String)
    realization TwoStacks( concept Stack( String)  realization LinkedList );

//Create a StringList object;
var list: StringList;

// create a console object to print out the strings
var c: Console;

// create a String object to hold the various strings
var str: String;

list.insertLeft("Hello");
list.insertRight("World");

list.retreat();

list.removeRight(str);
c.println(str); //prints "Hello"

list.removeRight(str);
c.println(str); //prints "World"
```

Figure 4. How a List component may be used

The class construct instantiates the generic concept and realization with fully determined types. In figure 4, we are saying that the StringList class is a List of Strings; it is implemented by the TwoStack realization, which in turn uses Stack objects implemented using a linked list. Hopefully, with the exception of the class construct, variable declaration, method calls, etc. will be familiar to developers used to modern object-oriented languages.

3 Language and Runtime Design

In this section, I discuss some of the more notable aspects of Sulu's language and runtime design. Specifically we talk about various object-oriented features of Sulu, its support for automated testing research, and a note on swapping as implemented in Sulu.

3.1 Support for Automated Testing

Sulu's architecture is designed such that it can support automated test-cases generation, execution, and evaluation. Many test-cases generation srategies, such as in [**Edwards00**] follow these steps:

- Generate candidate test cases
- Filter out invalid candidates
- Run the test-cases using an oracle to determine whether the test-cases pass or fail
- Determine the efficacy of the test-cases using metrics such as code coverage

As of this writing, the Sulu runtime is being developed to support a plug-in architecture that will allow it to run different test-cases generation algorithms.

The executable nature of Sulu's specifications contribute to two aspects of the evaluation of test-cases. The first is that execution of preconditions can act as a test-case *filter* test-cases that result in precondition failures are considered invalid.

Valid test cases (those that do not result in precondition failures) can then be run against method postcondition and invariat checks. Valid test-cases that result in postcondition and invariant failures means that that test-case found a bug. That is, runtime checking of specs can also be used as an *oracle* determining whether the component behaved correctly or not.

Finally, after running the test-cases, we may wish to determine how well the test-cases exercised the component under test. One increasingly popular way to do this is to use code coverage metrics. The Sulu runtime system already compiles statement coverage and decision coverage information, and work to support the collection modified condition-decision coverage is underway.

3.2 Object Orientation

As can be seen from the example in Section 2, Sulu adopts the modern object-oriented dot notation. It also embraces the typical notion of encapsulation of data and functionality within an object's implementation -- a Sulu realization.

One of the core defining aspects of object-oriented languages is inheritance. Sulu also has inheritance. Concepts may inherit from other concepts, realizations "inherit" a concept's behavioral specification by implementing them, and realizations may inherit from other realizations as well, allowing it to reuse the parent realization's implementation.

For the sake of simplicity and ease of implementation, however, Sulu uses a single inheritance heirarchy. Multiple inheritance is not allowed; perhaps most notably, a realization is only allowed to implement one concept (and all that concept's ancestors). Moreover, a realization that inherit from another realization may only implement the parent realization's concept. That is, the inheritance hierarchy discussed in [**Tan02**] and shown in figure 5, is currently still impossible with Sulu.

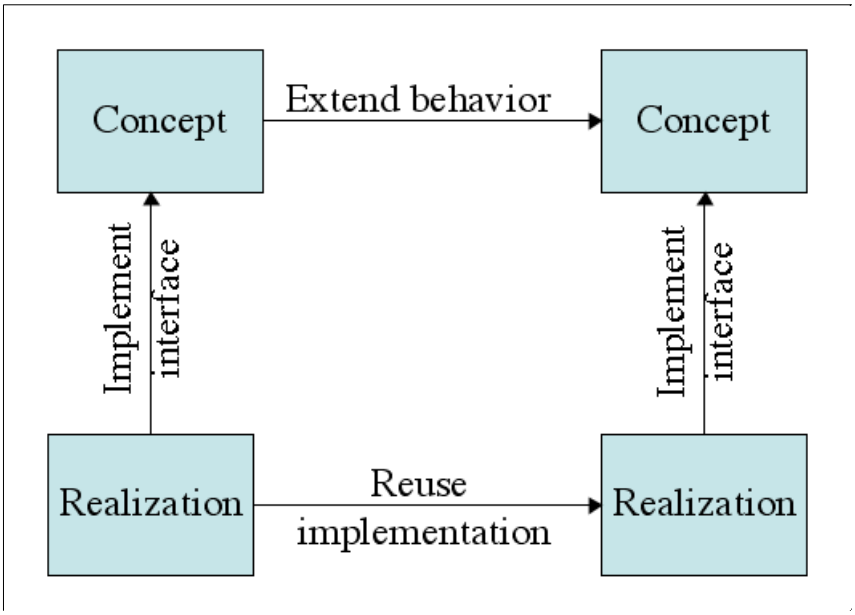

Figure 5. This kind of inheritance hierarchy is not (yet) allowed in Sulu

3.4 Matching and the binary operation problem

One feature of the Sulu programming language is that the inheritance relationship also defines a *matching* relationship, instead of the traditional "is a" relationship. Sulu matching relationship essentially follows the work found in [**Bruce02**].

Sulu's use of the matching relationship stems from the need for binary operations. It is often the case where an object needs to operate on other objects of the same type. One class of operators, for example, are comparison operators.

Imagine we want to build a sorting machine. Properly designed, this sorting machine should be able to sort all kinds of objects that can be compared with each other. So perhaps we create a concept called Comparable, that all objects that can be compared to each other can inherit from. Perhaps it looks like this:

```
concept Comparable() {
    method greaterThan( other: concept Comparable() ): Bool;
    method lessThan( other: concept Comparable() ): Bool;
    method equals( other: concept Comparable() ): Bool;
}
```

Figure 6. A Comparable concept using traditional OO design.

Using conventional OO design, we might want to build a subtyping hierarchy that looks like this:

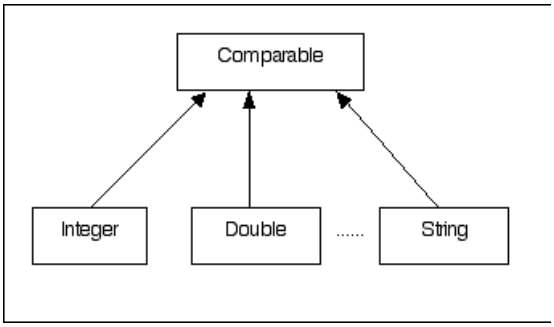

Figure 7. A conventional OO hierarchy for Comparable.

What happens when we compare Strings with Integers? Should we be able to?

Preserve subtyping via invariant parameters

Conventional wisdom says that interface inheritance should define a *subtyping* [**Liskov94**] relationship. For the subtyping relationship to hold, however, input parameters must be contravariant, and output parameters must be covariant. Sulu parameters are always in-out, so they have to be invariant.

Practically, this means that Integer objects must be able to accept other types of Comparable objects (say, String objects) as the parameter to the lessThan method. In Java, and many other OO languages, the solution is to have use the instanceof oparator (or equivalent) , and throw an exception if the parameter is of the wrong type. Sulu does not have exceptions, but a similar solution may be to have an extra out parameter that tells you the status of the operation -- whether the comparison succeeded or not. While this solution preserves subtyping, I believe it is quite awkward.

```
method greaterThan( other: concept Comparable(), compared: Bool ): Bool {

    if( other instanceof Integer() ) {
        //do comparison here
    } else {
        compared := false;
    }
}
```

Figure 8. A greaterThan method that preserves subtyping.

Allowing covariance

A second possiblity is to junk the subtyping relationship and allow covariance.

```
concept Integer() extends Comparable() {

    method greaterThan( other: concept Integer() ): Bool;

    method lessThan( other: concept Integer() ): Bool;

    method equals( other: concept Integer() ): Bool;

}
```

Figure 9. An Integer concept when covariance is allowed

This is essentially the path that Meyer takes with Eiffel. The main problem with allowing covariant solutions is that it becomes impossible to build a sound static type checker for the language. That is, some type errors would only be detectable at runtime.

Using generic parameters

A third possibility is to use generic parameters to break up the monolithic hierarchy into several subtyping hierarchies. That is, we can add a parameter to the Comparable concept that determines what types of objects can be compared. Unfortunately, the resulting solution in Sulu necessitates self-referential parameters.

```
concept Comparable( SelfType: concept Comparable( SelfType ) )  {

    method greaterThan( other: SelfType ): Bool;

    method lessThan( other: SelfType ): Bool;
```

```
        method equals( other: SelfType ): Bool;
    }
```

Figure 10. A Comparable concept using generic parameters to generate different subtyping hierarchies.

Here is how one might create and use a fully instantiated integer class:

```
    class Int: Integer( Int ) realization Builtin();
    var a: Int;
    a := 1; //etc.
```

Figure 11. Creating and instantiating an integer class

Figure 12 illustrates how using self-referential generic parameters can generate several subtyping hierarchies for the Comparable concept.

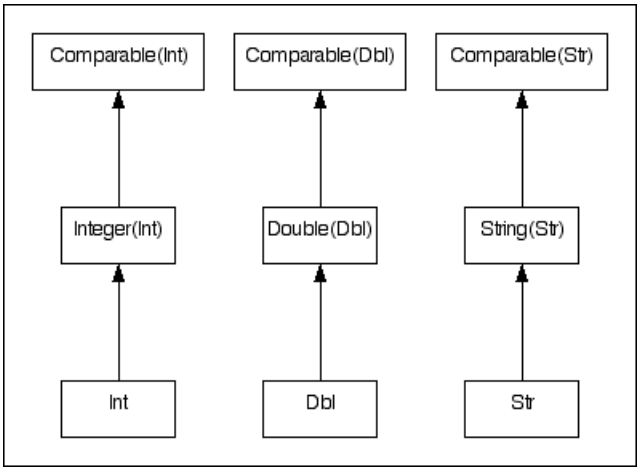

Figure 12. Using generic parameters breaks up the subtyping hierarchy.

Finally, using this scheme, here's how a header for a SortingMachine concept would look like:

```
    concept SortingMachine( Item: concept Comparable( Item ) ) { //...
```

Figure 13. Sorting machine concept header using the generic parameter scheme

While this works, you can imagine my frustration at the confusing, circular syntax!

Allowing covariance for self types only

Finally, the solution arrived by Bruce in [**Bruce02**] is also the approach used by Sulu. It essentially replaces subtyping with a new relationship called *matching* that allows covariance but only for self types. In Sulu, we've adopted a keyword called selftype:

```
    concept Comparable()  {
        method greaterThan( selftype ): Bool;
        method lessThan( selftype ): Bool;
        method equals( selftype ): Bool;
    }
```

Figure 14. Comparable concept using the selftype keyword.

Thus, an Integer realization may use the same signature for greaterThan, lessThan, and equals, but it is taken to mean that the types taken in by the methods are of the Integer realization, not any implementation of Comparable.

Using selftype has these advantages:

- Unlike the invariant parameter approach, using selftype limits binary operations only to objects of the same type.
- Unlike the covariant parameters approach, using selftype makes it possible to build a sound type system.
- Unlike the generic parameters approach, the syntax is straightforward and more easily understood.

3.5 Swapping, Assignment, and Parameter passing

Sulu follows RESOLVE in that the main data movement operator is swapping, not assignment. In Sulu, variables map names to objects. Objects may be swapped between variables, or passed into and out of methods also via swapping, following the strategy used in [**Harms91**]. In a departure from RESOLVE, however, all parameter passing in Sulu is in-out.

[Weide02b] presents the argument for using swapping as the data movement operator in a programming language. Because objects can only be swapped between variables, variables hold unique objects. This solves the problem of aliasing introduced by by-reference assignment. But because swapping can be implemented under the hood as a constant time operation, it solves many of the efficiency problems of by-value assignment.

Sulu, however, does allow assignment in one case. That is the case where a variable is assigned the return value of a method. This is permissible because methods always return new objects, and never aliases.

It is sometimes necessary, especially when dealing with low-level implementation of data structures, that a programmer needs some way of sharing references to objects. For example, if one needs to implement tail-sharing lists. Sulu is envisioned to provide various pointer components for this. Currently, a *ChainPointer* component is provided that an be used for implementing various acyclic data structures.

4 Future work and concluding remarks

In this paper, I have presented a brief overview of the Sulu programming language, and a couple of language design issues that I found challenging to tackle. However, there are still many outstanding issues that need to be resolved (as it were) for the Sulu programming language; there is much that still needs to be done. Here are some:

Automated testing evaluation. This is the core focus of my research; as of this writing, the plug-in system for automated test-cases generation is still under development. There have been various strategies that have been implemented, with varying levels of automation [**Cheon02**, **Edwards00**, **Mungara03**] that can be adapted for Sulu.

More components. There is a need to write more components for the programming language. Traditional data-structure type of components to be sure, but also I/O components and GUI components as well. Determining how well the swapping paradigm works for GUI programming may be an interesting topic to explore. There is also a need for a larger number of fully specified components.

Bug fixes and documentation. Because of its relative newness, Sulu is not as stable and bug-free as I would like it to be; the syntax and semantics of the language is also not as completely well documented as I'd like (I hope this paper has helped in alleviating that problem, however slightly).

IDE integration. Integration with Eclipse, syntax-coloring files for vim and emacs, or even giving Sulu its own editor would be nice to have.

I hope that I have piqued the reader's interest in the Sulu programming language. Sulu is still being actively developed; if the reader is interested in contributing to this work, he or she is encouraged to check out the sulu-lang project at Sourceforge [**Sulu-sfg**]. Any help for this project is certainly welcome.

References

[Bruce02]
Kim B. Bruce. Foundations of Object-Oriented Languages: Types and Semantics. MIT Press, 2002.

[Cheon02]
Yoonsik Cheon and Gary T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In Boris Magnusson (ed.), ECOOP 2002 -- Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 2002, Proceedings. Volume 2374 of Lecture Notes in Computer Science, Springer Verlag, 2002, pages 231-255.

[Edwards00]
Stephen H. Edwards. Black-box testing using flowgraphs: An experimental assessment of effectiveness and automation potential. Software Testing, Verification and Reliability, 10(4):249--262, December, 2000.

[Harms91]
Douglas E. Harms and Bruce W. Weide. Copying and Swapping: Influences on the Design of Reusable Software Components. IEEE Transactions on Software Engineering, 17, 5, May 1991.

[Leavens99]
Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), Behavioral Specifications of Businesses and Systems, chapter 12, Kluwer, 1999.

[Liskov94]
Barbara Liskov and Jeanette M. Wing. A Behavioral Notion of Subtyping. ACM Trans. Programming Languages and Systems, 16(6):1811-1841, November 1994.

[Mungara03]
Mahesh Babu Mungara. A method for systematically generating tests from object-oriented class interfaces. Master's thesis, Virginia Tech, 2003. (**etd**).

[Sitaraman94]
Murali Sitaraman and Bruce W. Weide, editors. Special feature: Component- based software using RESOLVE. Software

Engineering Notes, 19(4):21-67, Oct 1994.
[Sulu-sfg]
Sulu sourceforge site. **http://sourceforge.net/projects/sulu-lang**
[Tan02]
Roy Patrick Tan. Design issues Toward an Object-Oriented RESOLVE, in Proceedings of the 2002 RESOLVE workshop.
[Weide02a]
Bruce W. Weide, Good News and Bad News About Software Engineering Practice. in Proceedings of the 2002 RESOLVE workshop.
[Weide02b]
Bruce W. Weide, Scott M. Pike, Wayne D. Heym. Why Swapping? in Proceedings of the 2002 RESOLVE workshop.

Behavioral Specification of Multiparadigm Programs

Matthew Thornton
Dept. of Computer Science
Virginia Tech
660 McBryde Hall
Blacksburg, VA 24061-0106 USA
thorntom@vt.edu

Abstract

The idea of a multiparadigm programming language, languages that incorporate two or more programming styles into one language, has been around for over 20 years. Multiparadigm languages, especially those that integrate the functional, imperative, logical, and object-oriented paradigms, are very flexible and free the programmer from following one dogmatic approach to programming or another. Practitioners who use these languages, however, do not have a rigorous approach to specify the behavior of their programs. Research is now underway to attempt to answer some of the questions that are raised in developing a specification language for a multiparadigm programming language.

Keywords

Formal Methods, Formal Semantics, Programming Languages

Paper Category: Position Paper
Emphasis: Research

1. Introduction

Multiparadigm programming languages are languages that incorporate more than one programming paradigm into them. Most programming languages today will incorporate at most 2 different paradigms. However, there is a set of less-known programming languages that integrate many paradigms, including the Functional, Imperative, Logical, and Object-Oriented paradigms, examples of which include Leda [Budd 1995], Oz [Muller 1995], and JavaMP [Naik 2003]. These types of programming languages are interesting in that they free the programmer from thinking in one mind set and from a software engineering standpoint allow for the best approach to be used to solve particular problems or integrate paradigms to come up with novel solutions to problems.

One shortcoming of programming in this type of language is the complete absence of formalisms in specifying the behavior of programs written in this style. There has been no work done in developing a specification language for programs written using a multiparadigm language. I contest that writing such a language is possible and that it can be done in a way that will not eliminate any of the advantages of using a multiparadigm programming language.

The rest of this paper will discuss a general outline of a solution to developing this multiparadigm specification language. Section 2 discusses the solution in a broad stroke. Section 3 will address some of the open problems and possible solutions that could provide the start of a discussion. Section 4 will discuss some of the work that has been done in semantics, specification, or programming that could be useful in this research. Section 5 will address "where we are" and "where we are going."

2. The Outline of a Solution

The very first question that needed to be addressed was the issue of what type of language to develop-a generic specification language that would work for all multiparadigm programming languages or a specification language specific to a particular programming language. Considering the fact that the idea of what constitutes a multiparadigm programming language is a moving target and the complexities involved in specifying behavior for all such languages when it isn't necessarily certain it can be done for one language, it was decided to pick a specific language. This, obviously, begs the question "Which language?"

Deciding which language was the next issue that needed addressed. For the scope of this project, developing our own multiparadigm languages was not a viable solution, especially considering the availability of other languages. While many languages filled the necessary requirements, it was decided that JavaMP would be the language of choice in this research. Other languages had x-factors that would make them more difficult to use (Leda was too old and Oz includes syntax that would add a level of complexity to potential user studies, for examples.) JavaMP is a translator that takes JavaMP code and translates it into pure Java where it can be compiled using the Java compiler. The language is essentially Java with additional grammar to include multiparadigm features, such as lambda functions, logical relations, and a global scope. JavaMP has many advantages over other multiparadigm languages, including familiar syntax, a popular language foundation, and the availability of the code.

Making use of JavaMP had an additional advantage-it is based off of Java. Researchers have devised a specification language for Java, called the Java Modeling Language (JML) [Leavens 2005]. The Java Modeling Language allows for specifying the properties of classes and class functions using pre and post conditions, as well as class invariants. JML is open source, so it is possible to extend the syntax of JML to include constructs that will allow us to specify programs written in a language such as JavaMP.

So, as a place to begin, to demonstrate that writing a specification language for a multiparadigm programming language is

possible, we will develop extensions to the specification language JML that will allow us to write specifications for programs written in JavaMP. To demonstrate that the extensions for JML are capable of specifying the behavior of JavaMP programs, it will be necessary to develop the formal semantics of JavaMP, extend the semantics of JML to include the extensions that are written, and show that a specification written in JML has a solution in JavaMP.

3. Open Problems and Possible Solutions

In order to define this specification language, the syntax for JML needs extended, which is the current focus of this research. The goal is to extend JML in such a way that the "look and feel" of the extensions is comparable with JML and that the changes to JML be minimized.

3.1. Representation of Functions in a Specification Language

In JavaMP, functions are a first-class data type, which means that they can be returned as results of a function, passed as a parameter, and redefined "on-the-fly". These types of behavior require that their behavior be addressed in any extensions to JML. Several proposed syntaxes are discussed below. The questions are which of these solutions should be used, should they be used in combination, or is there a better solution?

In the case where a function is passed as a parameter or returned as a result, it will be necessary to specify the behavior of the function. It may not, however, be desirable to write the complete behavior for the function, but only discuss some of the properties of the function, like acceptable parameter values, etc. As a result, it may be desirable to discuss functions as a collection of properties: the return value, arity, parameter types, for example. The advantage of this type of syntax is that it is very flexible and can be used in a number of circumstances in the same manner. The disadvantage is that while it is flexible when specifying general properties of a given function, it can be more cumbersome when a full specification of a function is desirable.

```
/*@
  @ requires
  @     f.result>0 && myList.length() > 0
  @ ensures
  @      \exists(IntList newList; newList.length()==myList.length();
  @               \forall(int x;
  @                       0<x<myList.length();
  @                        newList.get(x)==f(myList.get(x)));
  @      && \result.equals(newList);
  @ */
public IntList map(IntList myList, [int(int)] f)
```

Figure 1.An example property-based function specification. Notice that we have constrained the set of functions to those where the result is greater than 0.

The other possibility is to develop a way of embedding typical JML specifications into the extensions to specify functions. In this case, a traditional JML "requires-ensures" clause would be included in the specification of the function. The advantage here is this is true to the ideas of JML and would require few changes to the syntax. However, in cases where one is specifying return values or parameters being passed, complete information about the function may not be available and you also will have the case where there will be nested requires-ensures clauses within the requires-ensures clause of the member function.

```
/*@
  @ requires
  @     function f [
  @       ensures \result >0
  @       ] && myList.length() > 0
  @ ensures
  @      \exists(IntList newList; newList.length()==myList.length();
  @               \forall(int x;
  @                       0<x<myList.length();
  @                        newList.get(x)==f(myList.get(x)));
  @      && \result.equals(newList);
  @ */
public IntList map(IntList myList, [int(int)] f)
```

Figure 2. An example property-based function specification. Notice in this case, we have specified the behavior of the function in an embedded requires-ensures clause.

Another possibility is to treat functions as a set mapping. JML includes set comprehension syntax and it may be possible to extend the syntax to describe the solution set of the function. The advantage here is that, as we see in section 3.2, the best solution to JavaMP relations is to describe a set of possible solutions. Because we are already talking about a set of solutions in relations, it is a simple matter of translating idea to functions, which minimizes the changes to JML. The disadvantage is, again, the problem of having to fully specify the function rather than just talk about properties of the function.

```
/*@
  @ requires
```

```
@     function f [
@       ensures
@           \result_set={\solution(int x, int \result)|
@             if(x<0) \result=-x
@             else \result=x+5}
@       ] && myList.length() > 0
@ ensures
@     \exists(IntList newList; newList.length()==myList.length();
@              \forall(int x;
@                     0<x<myList.length();
@                      newList.get(x)==f(myList.get(x)));
@     && \result.equals(newList);
@ */
public IntList map(IntList myList, [int(int)] f)
```

Figure 3. An example set-comprehension specification. In this case, we are defining the mapping of the solution set and because it is a physical mapping from A->B, we need to talk about both the incoming values and the result of the function. This is encapsulated as a tuple, \solution.

The best solution that we have at the moment is to make use of the set-comprehension syntax as well as including the ability to look at properties of functions individually, such as the function result or arity.

3.2. JavaMP Relations

JavaMP provides the idea of a relation to allow for logical predicates to be defined. As in other languages, such as Prolog, these logical predicates can result in more than one possible solution based on the arguments that are passed. JavaMP's if statements can recognize the presence of a solution. While loops in JavaMP allow a programmer to iterate over the entire set of solutions. As a result, we must account for all of the possible solutions in our specification language. Because we have already developed the concept of a solution set for the solutions of functions, it is a natural extension to use it as the solution set for relations.

The mechanism used in JavaMP to convey the solutions of a predicate is through parameter passing. The language includes pass-by-name as a method for moving data back and forth across the called method. Like in other logical programming languages, JavaMP allows the variables to be "bound" or "unbound" (unbound in this case means, effectively, passing a variable that is null.) This introduces the interesting issue of what using a named variable actually means in the specification-how can you specify a function in terms of a variable that may be unbound? If pass-by-name did not allow assignments to the parameter (x:=y+1, where x is a pass-by-name parameter), we could make the implication that a variable that is passed by name is potentially unifiable and have a specification similar to figure 4. Unfortunately, JavaMP allows for assignments to a parameter when the actual parameter is a variable. As a result, we need to distinguish between the cases where pass-by-name is simply the preferred passing mode for data and the case where the variable is potentially unifiable in a logical relation.

Possible solutions to this problem vary. The variance is on how much information remains hidden from the programmer. The best case being that the binding of the variables is implicit to the specification, resulting in comparatively simple specifications. The worst case is defining each possible binding pattern (using tags, such as \bound() and \unbound() in the specification) of the variable and the specification for each pattern in a way similar to the pattern matching idea used in languages like ML [**Milner 1997**].

The middle case scenario is to develop tags indicating that a given parameter is potentially unifiable. This solution, however, pushes down the complexity of the specification into the postcondition where one is specifying the behavior of the function. In each case that a parameter is potentially unifiable, the specification has to include an existentially-quantified variable that satisfies some aspect of the postcondition. That variable is then unified with the parameter. This results in a postcondition that is long and full of nested existential quantifiers, making the specification difficult to read, at best. Work is now underway to find a way around this within the behavior of pass-by-name and relations in JavaMP or to develop a convincing cover story that would eliminate the need to write all of the existential quantifiers.

```
/*@
  @     ensures
  @     \result_set=({\solution(head, tail) | edgeSet.hasEdge(
  @              head, tail) });
  @*/
public relation edge(Integer @ head, Integer @ tail)
```

Figure 4. Ideal specification of a logical relation. The best case is to make the binding behavior of the language implicit in the specification so that it works for any binding pattern. However, additional information about the variables may be necessary.

3.3. Developing the Semantics of JavaMP and JML Extensions

The most daunting part of this research will be the development of the semantics of JavaMP and the JML extensions. There has been work done in the past on the formal semantics of both JML and Java [**Berg 1999**, **Jacobs 2001**, **Jacobs 2002**]. However, the semantics that were devised were written with the goal of making it easy to implement a theorem prover, rather than make them readable or understandable to humans. The best approach in our mind is to use this as a basis from which to write our own specifications that are more readable for humans.

In addition to this more procedural issue is the very real issue of how to specify in the semantics behavior of constructs in a programming language that is not often specified. These include issues such as pass-by-name, logical relations, and functions as return values and parameters. Each of these concepts has a relatively common way of describing the necessary behavior. (Pass-by-name creates a parameterless function to pass an argument, for example.) The question to answer now is whether or not these standard ways of describing concepts in JavaMP relate to useful formal semantics for the language.

4. Related Work

JML is a behavioral specification language that is being actively researched by a number of groups. A reference manual to JML, including the syntax of the language to be extended can be seen here [**Leavens 2005**]. A survey of the tools that may be available to extend for use in user studies can be found here [**Burdy 2005**].

There are many examples of programs written in a multiparadigm fashion. Some of the best come from Dr. Budd's works on programming in Leda [**Budd 1992**, **Budd 1995**]. These will be used as the basis of any case studies or user studies used in our research and provide very clean examples of programs to specify with our specification language. This language forms the basis of the work done in JavaMP.

The best solutions available to the open problems discussed in section 3 make use of set comprehension syntax, including using it to specify the behavior of logical relations. An interesting example of using similar syntax in a database query language (which is where the benefits of a logical programming language could be leveraged) is found in [**Buneman 1994**].

5. Conclusions

Programming in a multiparadigm programming language has many benefits. Providing a specification language to a multiparadigm language will provide practitioners specifications with the type of rigor necessary in larger scale projects. Currently the syntax of the extensions needed for JML are being written. Following this, the semantics for these extensions will have to be integrated into some existing specification of JML. After this, it will be shown that the specification language will allow one to write specifications for JavaMP by creating a mapping between JML and JavaMP. (This will, of course, require a definition of the semantics of JavaMP, as well.) The preceding has been a survey of the work that is already been done and is currently being done. It is obviously an incomplete accounting and more problems will arise and (hopefully) better solutions will be found.

References

[Berg 1999]
J. v. d. Berg, M. Huisman, B. Jacobs and E. Poll, *A Type-Theoretic Memory Model for Verification of Sequential Java Programs*, Recent Trends in Algebraic Development Techniques (WADT '99), Springer-Verlag, 1999, 1-21.

[Budd 1992]
T. Budd, *Multiparadigm Data Structures in Leda*, Proceedings of the 1992 International Conference on Computer Languages, 1992, 165-173.

[Budd 1995]
T. Budd, *Multiparadigm Programming in Leda*, Addison-Wesley, 1995.

[Buneman 1994]
P. Buneman, L. Libkin, D. Suciu, V. Tannen and L. Wong, *Comprehension Syntax*, SIGMOD Record, 23 (1994), pp. 87-96.

[Burdy 2005]
L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Rustan, M. Leino and E. Poll, *An Overview of JML Tools and Applications*, International Journal on Software Tools for Technology Transfer, 7 (2005), pp. 212-232. Available at **ftp://ftp.cs.iastate.edu/pub/leavens/JML/sttt04.pdf**

[Jacobs 2001]
B. Jacobs and E. Poll, *A Logic for the Java Modeling Language (JML)*, Lecture Notes in Computer Science, 2029 (2001), pp. 284-291.

[Jacobs 2002]
B. Jacobs and E. Poll, *Coalgebras and Monads in the Semantics of Java*, Theoretical Computer Science, 291 (2002), pp. 329-349.

[Leavens 2005]
G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Muller and J. Kiniry, *JML Reference Manual*, 2005. Available at **ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmlrefman.pdf**

[Milner 1997]
R. Milner, *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1997.

[Muller 1995]
M. Muller, T. Muller and P. V. Roy, *Multi-Paradigm Programming in Oz*, Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog (Workshop at ILPS 95), Portland, OR, 1995.

[Naik 2003]
R. Naik, *Multiparadigm Programming with JavaMP*, Electrical Engineering and Computer Science, Oregon State University, 2003. Available at **http://eecs.oregonstate.edu/library/files/2003-2/Project%20Report.pdf**

Design Issues in Developing Data Abstractions for Evolving Graphs

Nighat Yasmin
Computer and Information Science
University of Mississippi
University, MS 29634 USA

yasmin61@juno.com
Phone: +1 864 656 3327

Murali Sitaraman
Computer Science
Clemson University
Clemson, SC, 29634-0974 USA

murali@cs.clemson.edu
Phone: +1 864 656 3444
Fax: +1 864 656 0145
URL: http://www.cs.clemson.edu/~resolve

Abstract

The benefits of recasting graph algorithms as data abstractions include ease of use, implementation independence, and the flexibility of incremental computation. This paper examines the issues in designing specifications of data abstractions for finding cheapest paths in graphs, when the graphs are allowed to change after some of the paths have been computed. These more complicated abstractions are necessary to capture the more typical problem of communication and transportation network topologies that evolve because of additions (and deletions) of network links. The paper also considers issues in building implementations that must be optimized to take advantage of shortest paths computed earlier in answering queries for new shortest paths. Developing these non-trivial components so that they are amenable to formal verification will help establish the scalability of Resolve verifying compiler technology. Specification and verification of performance profiles for optimizing implementations is another important topic for discussion.

Keywords

Shortest path computation, dynamic graphs, specification, communication networks, optimization

Paper Category: technical paper
Emphasis: research

1. Introduction

Noting the software engineering problems of classical "batch-style" graph algorithms, such as ones to find spanning forests and cheapest paths, Weide et al., have noted the importance of designing data abstractions to solve these problems incrementally [WOS94, SWLO00]. The concept for finding cheapest paths allows multiple questions to be answered, once the edges of a graph have been inserted. This paper explores the issues in generalizing the concept to allow for edge addition and deletion after some shortest paths have been computed, and implementation issues in optimization. The generalization and optimization are essential for the abstraction to be widely useful for computing shortest paths in communication networks (whose topologies change due to link failures and recovery) and transportation networks (that change due to road constructions and blockage). At the same time, they make it non-trivial to specify and implement, allowing us to explore the scalability of Resolve principles in more complicated situations.

2. The Problem

Computation of shortest paths is among the most-widely studied graph problems with practical applications in network communication and in transportation networks. Conventionally, for static graphs, the shortest path problem has been solved in "batch style" following a procedural approach, using algorithms for finding either Single Source Shortest Paths (SSSP) or All Pair Shortest Paths (APSP) [CLR90, Dij59, PR02, RR97, RTR96]. A data abstraction solution proposed in [SWLO00] overcomes key shortcomings of the procedural approach, but it still demands static or unchanging graphs. This paper considers interface design problems in permitting graphs to evolve and implementation design problems in updating shortest paths efficiently when graphs evolve, so that the results are amenable to automated verification.

3. The Position

Data abstraction components for evolving graphs have practical applications. However, specification and optimized implementations of such components are non-trivial. So they can serve as useful exercises for demonstrating the scalability of Resolve principles.

4. Justification

Specification Issues

Figure 1 reproduces a specification of the *Cheapest_Path_Template* from [Ogd00]. In the specification, *Vertex*, *Edge_Universe*, and *Edge_Info* are purely mathematical notions. The provided program type *Graph_Holder*, is mathematically modeled as a Cartesian Product:

1. *Is_Gr_Edge* says whether an edge from *Edge_Universe* (which stands for a collection of unique identification for edges) belongs to the graph under discussion;
2. *E_Info* holds the information corresponding to an edge, including its cost, its source and target vertices, and other information, termed, *Edge_Label*;
3. *Accepting* is a Boolean flag that is true if and only if edges may be added to the *Graph_Holder*.

Initially, a graph holder GH is accepting edges and the predicate *Is_Gr_Edge* is false for all edges in the *Edge_Universe*. When using this concept to find a cheapest path between two vertices of a graph, a caller begins inserting edges using the *Add_Edge* operation. The operation *Stop_Accepting_Edges* needs to be called before finding shortest paths. To find the shortest path between two vertices, the caller needs to invoke the operation *Get_First_Edge_for* to get each edge repeatedly, using the previously returned vertex as the origin for the next invocation. We have omitted the details of definitions, such as *Is_Cheapest_Connecting_Path*, because they are not directly relevant for the issues raised here.

Concept Cheapest_Path_Template(**type** Edge_Label; **evaluates** Vertex_Max, Max_Edge_Count: Integer);

uses Std_Integer_Fac, Std_Real_Num_Fac;

requires 0 < Vertex_Max **and** 0 < Max_Edge_Count;

Definition Vertex: $\tilde{A}(\mathbb{N}^{+})$ = [1..Vertex_Max];

Defines Edge_Universe: Set;

constraint ‖Edge_Universe‖ ³ Max_Edge_Count;

Definition type Edge_Info = **Cart_Prod**

Src, Tgt: Vertex;

Cost: $\mathbb{R}^{30}$;

Label: Edge_Label

end;

```
    Type Family Graph_Holder Í Cart_Prod
                                        Is_Gr_Edge: Edge_Universe®B;
                                        E_Info: Edge_Universe®Edge_Info;
                                        Accepting: B
                end;
                exemplar GH;
                Def const Edge_Count( GH: Graph_Holder ): ℕ = (‖{ E:
Edge_UniverseôGH.Is_Gr_Edge(E) }‖ );
                constraint Edge_Count(GH) £ Max_Edge_Count;
                initialization
                        ensures GH.Accepting and Edge_Count(GH) = 0;


        -- other defintions


        Def const Is_Cheapest_Connecting( s, d: Vertex, r: Str(Edge_Universe), GH: Graph_Holder ): B
                = ...


        Oper Add_Edge( evaluates s, t: Integer; evaluates C: Real; alters L: Edge_Label; updates GH:
Graph_Holder );
                requires GH.Accepting and s, t Î Vertex and C Î ℝ^30 and Edge_Count( GH ) <
Max_Edge_Count;
                ensures $ UE: Edge_Universe ' Agrees_Elsewhere(GH, #GH, UE) and
                        ¬ #GH.Is_Gr_Edge(UE) and GH.Is_Gr_Edge(UE) and GH.E_Info(UE).Src = s
                        and GH.E_Info(UE).Tgt = t and GH.E_Info(UE).Cost = C and
GH.E_Info(UE).Label = #L;


        Oper Stop_Accepting_Edges( updates GH: Graph_Holder );
                requires GH.Accepting;
                ensures ¬ GH.Accepting and
                        GH.Is_Gr_Edge = #GH.Is_Gr_Edge and
                        GH.E_Info = #GH.E_Info;


        Oper Get_First_Edge_for( evaluates Origin, Dest: Integer; restores GH: Graph_Holder;
            replaces Fst_Vrtx: Integer; replaces E_Cst: Real; replaces E_Lbl: Edge_Label );
                requires ¬ GH.Accepting and Origin ¹ Dest and $ r: Str(Edge_Universe) '
Is_Path_Connecting( Origin, Dest, r, GH );
```

```
        ensures $ s: Str(Edge_Universe), $ E: Edge_Universe ' Fst_Vrtx = GH.E_Info(E).Tgt and
                E_Cst = GH.E_Info(E).Cost and E_Lbl = GH.E_Info(E).Label and
                Is_Cheapest_Connecting(Origin, Dest, áEñ◦s, GH);


    -- other operations
end Cheapest_Path_Template;
```

Figure 1: Specification of Data Abstraction for Finding Shortest Paths

The change necessary to the concept to allow insertion of edges after some paths have been computed is relatively straightforward. We can replace the operation *Stop_Accepting_Edges* with a *Change_Mode* operation as shown below.

```
    Oper Change_Mode (updates GH: Graph_Holder);
        ensures GH.Accepting = ¬ #GH.Accepting and
                GH.Is_Gr_Edge = #GH.Is_Gr_Edge and
                GH.E_Info = #GH.E_Info;
```

After inserting graph edges and finding some shortest paths, a client can invoke *Change_Mode* operation and add new edges to the graph.

Allowing edges to be deleted makes the specification a much more challenging problem, because in turn it requires the ability to identify and search edges. One possibility is to replace the conceptual edge identification based on *Edge_Universe* in the specification with a concrete program type. So the graph holder will provide an *Edge_Id* type in addition to the *Graph_Holder*. The *Add_Edge* operation would plausibly return a (unique) id when a new edge is added. A *Delete_Edge* operation would take an id as a parameter as would an operation to search and retrieve information pertaining to an id. If the id is introduced, it would probably also make sense for edge ids to be returned when a cheapest path is needed rather than the edge information. This design essentially leads to a concept that combines an abstraction for searching with the one for finding a shortest path. At a certain level, this may be the appropriate design because the problem requires combining the ability to find shortest paths with database-type queries. At the same time, it raises the question if the next generation of Resolve concepts would be such non-trivial inseparable combinations of abstractions conceived earlier. There are also corresponding language design questions.

Implementation Issues

The impact of the minor change from *Stop_Accepting_Edges* to *Change_Mode* in the specification is significant on implementations that must optimize to be efficient. We begin with the discussion of an implementation based on Dijkstra's algorithm using an adjacency-list representation to hold graph edges. Figure 2 shows key details of the set up including the representation and correspondence assertion. To establish the correspondence with the mathematical model of graphs shown in Figure 1, we have used *adjunct variables* and give unique identifications to edges. Though the edge numbers are abstract, for a given representation value, it seems necessary for the correspondence to relate it to the same abstract edges from the *Edge_Universe*. A predicate *Is_E_Info_For_Edge* has been used to extract the information corresponding to an edge id.

```
Realization Dijkstra_Realization for Cheapest_Path_Template (
        Operation Copy_Label (replaces Copy: Edge_Label; restores Orig: Edge_Label)
            ensures Copy = Orig; );
    uses Static_Array_Template, Record_Template, One_Way_List_Template, Prioritizer_Template;
```

```
Definition Edge_Universe: Ã(N+) = [1.. Max_Edge_Count];


Type Edge_Info_Stored = Record
        Aux Edge_Num: Integer;
        Src, Tgt: Vertex;
        Cost: Real;
        Label: Edge_Label;
    end;


Type Graph_Holder = Record
        Facility List_Fac is One_Way_List_Template (Edge_Info_Stored,
                                                    Max_Edge_Count) realized by Shared_Realiz;
        Aux Edge_Counter: Integer;
        Adj_Lists_Holder: Array (1.. Max_Vertex) of List_Fac.List_Position;
        -- solution structures omitted
    end;
  Conventions …
Definition Is_E_Info_For_Edge (E: Edge_Universe, EI: Edge_Info, GH: Graph_Holder): B = (
          $ v: Integer, $EIS: Edge_Info_Stored ' 1£ v £ Max_Vertex and
          <EIS> Is_Substring_of GH.Adj_List_Holder[v].Rem and
          E = EIS.Edge_Num and
          EI = (EIS.Src, EIS.Tgt, EIS.Cost, EIS.Label)
    );
Correspondence
        Conc.Graph_Holder.Is_Gr_Edge (E) iff 1£ E £ GH.Edge_Counter and
        Conc.Graph_Holder.E_Info (E) = {EI, if 1£ E £ GH.Edge_Counter and Is_E_Info_For_Edge
    (E, EI, GH)
                                                (1, 1, 0.0, L)}  and
        Conc.Graph_Holder.Accepting = GH. Accepting;
  -- code for operations omitted
end Dijkstra_Realization;
```

Figure 2: Outline of a Realization

Notice the use of auxiliary variables *Edge_Num* and *Edge_Counter*. If we want to allow a delete edge operation, then both of these variables would be concrete variables.

Presumably, if edge deletions are allowed, the auxiliary field *Edge_Num* would become a concrete part of the representation.

Several versions of optimization are necessary for the concept to be realized efficiently. The optimizations in turn demand non-trivial *representation invariants* or conventions. In the simplest case, when no new edges are allowed to be added or deleted, the representation must maintain the shortest paths that have been computed, so that eventually no new calculations will be necessary.

When an edge is inserted, after some shortest paths have been computed, it is necessary to discard or recompute only some of the calculated paths. In the case of fully-dynamic graphs (when edges are inserted and deleted), one approach when an edge is deleted, is to update only the sub-shortest paths to the vertices that are descendents of the target node of the deleted edge [NST00]. In the process, it is necessary to check, if the deleted edge will divide the graph into two disconnected sub-graphs [ES81].

While implementation optimization is essential for the class of problems discussed here, it also raises fundamental questions about developing performance profiles suitable for component users.

5. Related Work

To our knowledge, outside the Resolve community, no one has explored the problems of formal specification of data abstractions for graph algorithms. However, much research has been done on computing shortest paths for dynamic graphs. For example, [Weighe01] discusses adaptability of algorithm component by discussing the shortest path problem whereas [AISN90] examine APSP for semi-dynamic graphs. [SP75] discusses big-oh estimates for finding and updating shortest paths. [FINP98] present experimental study for shortest paths in dynamic graphs.

6. Conclusion

While the problems of specification and implementations are non-trivial to handle dynamically evolving graphs, fortunately, we need to develop suitable reusable components only once. The objective of this paper is to motivate a discussion on design and specification of data abstractions, optimizing implementations, and their performance profiles at the workshop. Verification of correctness of such components will help validate the scalability of Resolve verifying compiler techniques and principles.

Acknowledgments

The specification used in this paper is due to Bill Ogden. This research is funded in part by a grant from the National Science Foundation (CCR-0113181) and a grant from the National Aeronautical and Space Administration through SC Space Grant Consortium.

References

[CLR90]

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, “Introduction to algorithms”, Published by The MIT Press and McGraw-Hill Book Company, Fourth edition, 1990.

[Dij59]

E. W. Dijkstra, “A note on two problems on connexion with graph”, *Numerische mathematik 1*, 1959, pp. 269-271.

[ES81]

S. Even and Y. Shiloach, “An on-line edge-deletion problem”, *Journal of the ACM (JACM)*, January 1981, Volume 28, Issue 5, pp. 1-4.

[FINP98]

D. Frigioni, M. Ioffreda, U. Nanni, and Giulio Pasqualone, “Experimental analysis of dynamic algorithms for the single source shortest paths problem”, *The ACM Journal of Experimental Algorithmics*, January 1998, Volume 3, Article 5, pp. 1-20. **http://doi.acm.org/10.1145/297096.297147**.

[NST00]

P. Narváez, Kai-Yeung Siu, and Hong-Yi Tzeng, “New dynamic algorithms for shortest path tree computation” *IEEE/ACM Transactions on Networking (TON)*. Volume 8, Issue 6, December 2000, pp. 734-746.

[Ogd00]

W. F. Ogden, “The Proper Conceptualization of Data Structures”*, Computer and Info. Science*, The Ohio State University, 2000

[PR02]

S. Pettie and V. Ramachandran, “Computing shortest paths with comparisons and additions”, In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, January 2002. **http://ce.sharif.edu/~ghodsi/archive/downloaded-papers/ACM%20SODA/2002/Computing%20shortest%20paths%20with%20comparisons%20and%20additions.pdf**.

[RR97]

R. Raman, “Recent results on the single-source shortest paths problem”, *ACM SIGACT News*, Volume 28, Issue 2, June 1997, pp. 81-87. **http://doi.acm.org/10.1145/261342.261352**.

[RTR96]

G. Ramalingam and T. Reps, “An Incremental Algorithm for a Generalization of the Shortest-Path Problem”, *Journal of Algorithms*, Volume 21, Issue 2, September 1996, pp. 267-305. **http://citeseer.ist.psu.edu/ramalingam92incremental.html**.

[SWLO00]

M. Sitaraman, B. W. Weide, T. J. Long, and W. F. Ogden, “A Data Abstraction Alternative to Data Structure/ Algorithm Modularization”, *Generic Programming '98, LNCS 1766*, Springer-Verlag Berlin Heidelberg, 2000, pp. 102-113. **http://www.cse.ohio-state.edu/rsrg/documents/2000/00SWLO.pdf**.

[SP75]

P. M. Spira and A. Pan, “On Finding and Updating Spanning Trees and Shortest Paths”, *SIAM J. Computing*, Volume 4, No. 3, September 1975, pp. 375-380. **http://locus.siam.org/fulltext/SICOMP/volume-04/0204032.pdf**.

[WOS94]

B. W. Weide, W. F. Ogden, and M. Sitaraman, “Recasting Algorithms to Encourage Reuse”, *IEEE Software*, September/October 1994, Volume 11, No. 5, pp. 80-88. **http://www.cse.ohio-state.edu/rsrg/documents/1994/94WOS.pdf**.

[Weighe01]

K. Weihe, “A software engineering perspective on algorithmics”, *ACM Computing Surveys (CSUR)*, Volume 33, Issue 1, March 2001, pp. 89-134. **http://doi.acm.org/10.1145/375360.375367**.

Walking the Line between Java and Resolve: Tako and the Verification Grand Challenge

Jyotindra Vasudeo
Gregory Kulczycki
Dept. of Computer Science
Virginia Tech
7054 Haycock Road
Falls Church, VA 22043 USA

vasudeo@vt.edu
gregwk@vt.edu
Phone: +1 703 538 3758
Fax: +1 703 538 8348
URL: http://www.directreasoning.org/wikitako/

Abstract

The Resolve language is designed with verification in mind. Language features are not added to Resolve unless the designers know how to reason about them. Java and C# were not designed with verification in mind, but significant research efforts have been made to facilitate ad hoc specification and reasoning. This paper presents Tako 1.0, a Java-like language that incorporates the alias-avoidance features of Resolve, including automatic initialization, a swap operator, in-out parameter passing, and a pointer component. With Tako, we hope to explore the possibility of a value-based verification system for an object-oriented language, leveraging the research done on verifying compilers in both the Resolve and Java/C# communities.

Keywords

Tako, object-oriented, language design, verification

Paper Category: position paper
Emphasis: research

1. Introduction

One of the main motivations behind the Tako project was the desire to answer the following question: Can language design principles used in Resolve be successfully and independently applied to the redesign of existing languages to make them more tractable to formal reasoning? Tako 1.0 represents a first step in answering that question ? it is a redesign of Java 1.4 that incorporates the alias-avoidance features of Resolve. These features include automatic initialization, alternative data assignment operators such as swapping and initializing transfer, in-out parameter passing and an initialization scheme that avoids the repeated argument problem, and a built-in pointer component that allows programmers to implement linked data structures. This paper gives a brief description of these how these features work in Tako 1.0. It discusses the implications for future work in the specification and verification of Tako and looks at the impact of the Tako language on a verifying compiler.

2. Language Features of Tako 1.0

Tako 1.0 is very similar to Java in terms of syntax and semantics while also borrowing some key concepts from Resolve, which make Tako programs easier to reason about than normal Java programs. The following sections describes how Tako is similar to Java and also other features central to Tako.

```
public class Stack {

    private Object[] contents;
    private int top;
    private final int MAX;

    public Stack (int n) {
        MAX := n;
        contents := new Object[MAX];
        top := -1;
    }

    public void push(Object x) {
        assert depth() < MAX;
        top++;
        contents[top] :=: x;
    }

    public void pop(Object x) {
        assert depth() > 0;
        x :=: contents[top];
        top--;
    }

    public int depth() {
        result := top + 1;
    }
}
```

Listing 1. An implementation of a stack in Tako

2.1 Similarities with Java

The Tako compiler is based on the Kopi Java compiler. Our goal for version 1.0 of Tako was to introduce only those Resolve-like features into Java that facilitated alias avoidance. Therefore, most of the object-oriented features of Java remain intact. Perhaps the most important of these is that classes remain the fundamental unit of modularity in Tako. They export a single type and they can contain private and public attributes and methods. In addition, as discussed below, we did not want to modify Java's approach to inheritance or polymorphism. Though the Tako compiler essentially translates Tako programs to Java 1.5 programs (and then compiles the Java code), we did not implement key features in Java 1.5, such as generics, enum types, or "auto boxing," which automatically wraps primitive types when they need to be treated like objects. Assertions, which are a feature of Java 1.4, are intended to be a part of Tako 1.0, though we have not yet implemented them. As any Java programmer can see from Listing 1, the Tako code for a stack object is not radically different than Java code for a stack class.

2.2 Automatic Initialization

In Tako, all variables are initialized when they are declared. Some Tako language features that we provide to avoid aliasing require automatic initialization, such as initializing transfer and the initializing approach for parameter passing. For primitive types, initial values are the same is in Java. For objects, variables are initialized using the default constructor, which is created automatically if the programmer does not supply one. When interface variables are declared (and not assigned to) they are not associated with an object. Therefore, we simply initialize these variables with null. When arrays are declared they are initialized with a zero length array. Therefore, programmers need to initialize them with the appropriate length. Tako 1.0 will automatically initialize each cell of the array, but this feature is currently not implemented. Programmers still must be aware when they are declaring an object, whether its default constructor is really what they want. For example, based on Listing 1, if a programmer declares `Stack s;` the contents array will have length zero by default, which is probably not what the programmer wants. Therefore, the Tako programmer should assign an explicitly constructed stack object to this variable: `Stack s := new Stack(20);`

2.3 Data Assignment

Reference assignment is not available in Tako. The primary means of data transfer is swapping (`:=:`) and initializing transfer (`<-`). Both swapping [**Harms91**] and initializing transfer [**Tan02**] avoid aliasing and thus simplify reasoning of Tako programs. In Listing 1, the swap operator is used in the push method in the statement `contents[top] :=: x;` Tako reports a compile-time error if the variables are not of the same type. If programmers want to transfer an object a specified type to an object of a supertype, the initializing transfer operator can be used. Apart from the above operators, Tako also provides a function assignment operator (`:=`) that can effectively be used as a "copy" operator. As its name implies, the function assignment operator is used for function assignment, as in `str := s.toString();` It is also used for assigning an explicitly constructed object to a variable, as in `contents := new Object[MAX];` If the operator is used with a variable instead of a function (or constructor) on its right-hand side, the compiler interprets the variable as having an implicit call to a special "replica" function. For example, `Stack s := t;` is interpreted as `Stack s := t.replica();` If no replica function is defined, the compiler reports an error. In functions, the special **`result`** variable ensures that a new object is always

created and aliasing is therefore avoided. Function always return the object currently held in the result variable. Therefore, the statement **return** `s;` is not permitted in Tako as it is in Java. However, the statement **return;** (without indicated what is returned) may be used to exit the function before it would normally terminate. The type of the **result** variable is the same as the return type of the function, and it is has an initial value at the beginning of the function. Programmers can roughly think of primitives as objects with special syntax that obey these same rules (as in `MAX := n;` and **result** `:= top + 1;`) but this view is not *entirely* sound in Tako 1.0. It will be implemented in a later version of Tako, but currently, primitives behave like they do in Java.

2.4 Parameter Passing

The most visible difference between parameter passing in Java and Tako is that Tako has in-out parameter passing. Thus, instead of writing the pop method as a side-effecting function (as Java requires), Tako's pop method can be written as a procedure (a void method). In the pop method in Listing 1, the parameter x gets swapped with an item of the `contents` array. The changes made to x are passed out to the actual parameter due to the in-out parameter passing mechanism in Tako. Conceptually, we can think of the parameters as being transferred in before the call and transferred out after the call using the initializing transfer operator. This avoids the repeated arguments problem described in [Kulczycki05]. Once the variable is transferred in, it gets initialized and consequent attempts to use the variable as a parameters in the same method will lead to passing in an initial value. In Tako, as in Java, the **this** variable denoting the current object is also an implicit parameter to the method (in non-static methods) and therefore must also behave as an in-out parameter. All parameters in Tako are in-out by default unless declared using the **eval** mode. (Note: the implicit **this** parameter cannot be an **eval** parameter). The **eval** mode indicates that a function rather than a variable is expected. If a variable is passed in, the compiler assumes the object is invoking its replica method (similar to function assignment).

Tako uses a special scheme to avoid run-time cast errors during parameter passing that might occur if a niave approach to in-out parameter passing is used. Tako does not have generics, so suppose a programmer wants the "pseudo-generic" stack described above to hold circle objects. She writes the following code:

```
Circle c;
Stack s = new Stack(20);
s.push(c);
```

After the declarations of the circle and stack, c holds and initial circle object and s holds and initial stack. Though conceptually the stack is empty, the contents array in the stack's representation holds 20 initialized objects of type "Object." When the push method is invoked with argument c, the circle object held by c is passed to the formal variable x of type Object. Then x gets swapped with the first element in the contents array. So now x holds and initialized object of type "Object," and contents[0] holds the circle object. Since Tako has in-out parameter passing, x's object value is transferred back to c. This means that an initialized object of type Object is passed to a variable c of type Circle, which clearly cannot be permitted.

To avoid this inconsistency in Tako, the variable c is initialized with a new Circle object instead of assigning the object of type "Object". Thus, for in-out parameter in Tako the variable to which the outgoing object is assigned, may be initialized, if the object cannot be held by the variable.

2.5 Pointer Component

Tako, like Resolve, provides a pointer component to implement linked structures. Listing 2 shows an example of a linked list implmented using the pointer component. The special syntax (including `->`, `^`, `*:=:`, and **allocate**) are short for methods in the pointer component, but it is implemented efficiently in the compiler using Java references, as illustrated in Table 1. For more information on the Tako pointer component, see http://www.directreasoning.org/wikitako/index.php/Pointer_behavior_through_the_position_class.

```
public class LinkedList {

     class Node is Object ^(next);

     private Node head, pre, last;

     private int left_length, right_length;

     public LinkedList() {
        allocate head;
        pre -> head;
        last -> head;
     }

     public void insert(Object x) {
        Node post, new_pos;
        post -> pre^next;
        allocate new_pos;
        new_pos *:=: x;
        pre^next -> new_pos;
        new_pos^next -> post;
        if (right_length = 0) {
            last -> last^next;
        }
        right_length++;
     }

     /* other list methods */

}
```

Listing 2. A portion of a linked list implementation in Tako

Tako syntax	Position class meaning	Java translation
`class Node is Object ^(next);`	`class Node is PositionType<Object>(1) implemented with Default`	`class Node extends PointerType { Node next = null; Object contents = new Object(); }`
`allocate p;`	`p.takeNew();`	`p = new Node();`
`p -> q`	`p.moveTo(q);`	`p = q;`
`p -> q^next;`	`p.moveTo(q); p.follow(1);`	`p = q.next;`
`p^next -> q;`	`p.redirect(1, q);`	`p.next = q;`
`p *:=: s;`	`p.swapObject(s);`	`Object temp = p.contents; p.contents = s; s = temp;`

Table 1. Tako pointer syntax and its meaning

2.6 Discussion

Tako maintains the most important object-oriented properties of Java, while introducing important alias-avoidance concepts of Resolve. In the 2002 Resolve workshop, Tan presented a number of issues that needed addressed in "the creation of an object-oriented version of Resolve" [Tan02]. The bullets below briefly state the relevant issues and tell how Tako handles them.

- How do classes relate to Resolve modules? ? We chose to keep the class as the fundamental unit of modularity in Tako. A class exports only one type, just as in Java.
- Should OO resulve use reference or value semantics? ? Since Resolve-like alias avoidance was the main feature we wanted to add to Java, Tako uses value semantics.
- How should OO Resolve handle inheritance? ? We tried to keep inheritance as close to Java-style inheritance as possible. Java has no multiple inheritance, so neither does Tako.
- How will OO affect type checking? ? Like Java, parameter types in Tako have to be invariant for method overriding, which is consistent with Tan's suggestion that in-out parameter types should be invariant. However, since Tako relies on Java type-checking, even eval mode parameters are invariant.
- How will exeptions be handled? ? Though Tako 1.0 technically handles exception just as Java does, it also takes addvantage of assertions in Java 1.4, as can be seen in the stack example in Listing 1.

3. Verification Issues in Tako

Tako includes language features from both Java and Resolve. Both the Java and the Resolve communities have significant research efforts in verification. We would like to be able to leverage this research in the verification of Tako programs.

3.1 Approaches to Verification

The following represent approaches to verification for which there is a significant body of research in either the Java or Resolve communities.

Runtime Assertion Checking

JML specifications are written in a functional subset of Java, and is intended to support both runtime assertion checking and formal verification [**Leavens05**]. The specifications can include model variables, abstraction functions, invariants, and basic requires and ensures assertions. The JML tool can check many (but not all) of these assertions at runtime.

ESC/Java.

The ESC/Java [**Leino00**] can statically check Java code for common errors such as the possible occurrence of null pointer exceptions. It also understands a subset of JML and can statically verify some lightweight specifications.

LOOP compiler.

The LOOP compiler [**van der Berg01**] is intended for full verification of sequential Java programs. To prove the correctness of the Java code with respect to its JML specification, the compiler translates both the code and the specification into a heap-based logic. The resulting logical theories are then proved with the help of the PVS theorem prover.

Resolve verification.

The RESOLVE verifying compiler focuses on modular, heavyweight verification using a value-based semantics [**Kulczycki06**]. No heap structures or special proof rules for pointers are needed. The state space at any point in the program is comprised of the programming and conceptual variables and their values. The frame property restricts the effects of a operation invocation to its actual parameters and the global variables declared in the updates clause of the operation declaration.

Ideally, we would like to be able to implement runtime assertion checking for Tako as well as automatic static checking of lightweight specifications in the spirit of ESC/Java. For full verification, however, we aim for a Resolve-style system with its simpler value semantics, state space, and frame property. It is an open question whether we can achieve this goal. In particular, we are not sure how to reason about Java-style inheritance using the simple frame property given in Resolve style semantics.

4. Conclusion

Tako 1.0 is a first step toward a Java-like language that includes Resolve-style features. Future work includes implementing constructed classes in Tako that correspond roughly to Resolve facilities.

References

[Harms91]
Harms, D. E., Weide, B. W., "Copying and Swapping: Influences on the Design of Reusable Software Components. IEEE Transactions on Software Engineering", 17, 5, May 1991.

[Kulczycki06]
Kulczycki, G., Duckworth, S., Sitaraman, M., Weide, B. W. "Abstracting Pointers for a Verifying Compiler," Technical Report RSRG-06-01 Clemson University (2006) http://www.cs.clemson.edu/~resolve/reports/RSRG-06-01.pdf

[Kulczycki05]
Kulczycki, G., Sitaraman, M., Ogden, W. F., Weide, B. W., "Clean Semantics for Calls with Repeated Arguments", Technical Report RSRG-05-01, Clemson University (2005) http://www.cs.clemson.edu/~resolve/reports/RSRG-05-01.pdf

[Leavens05]
Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., Cok. D. R., "How the Design of JML Accommodates both Runtime Assertion Checking and Formal Verification. Science of Computer Programming", Vol. 55. Elsevier (2005) 185?205

[Leino00]
Leino, K. R. M., Nelson, G., Saxe, J. B., "ESC/Java User?s Manual. Technical Note 2000-002", Compaq Systems Research Center (2000)

[Tan02]
Tan, R. P., "Design Issues Toward an Object Oriented RESOLVE". In Proceedings of the RESOLVE Workshop 2002 , June 2002.

[van der Berg01]
van den Berg, J., Jacobs, B., "The LOOP Compiler for Java and JML", In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, Springer (2001)