Engineering Specifications and Mathematics for Verified Software

---

A Dissertation Proposal by

Hampton Smith

8 December 2011

Submitted to the graduate faculty of the
School of Computing
in partial fulfillment of the requirements
for the Dissertation Proposal and
subsequent Ph.D. in Computer Science

Approved By:
Murali Sitaraman, Committee Chair
Brian C. Dean
Jason O. Hallstrom
Roy P. Pargas

Outline

Abstract

At the heart of the argument for formal, mathematical methods of software quality assurance is that increased energy spent to develop formal specifications and prove software components against those specifications is amortized over the lifetime of the verified component. Thus, modularity and reuse are central prerequisites of practical verification. Because of this, there are two strategies for reducing effective energy invested in a verified component: 1) decrease the amount of effort required to verify it, and 2) increase its reusability and thus its lifetime.

While many modern verification systems exist, few seem to have been designed with modularity and reuse in mind. On the one hand are systems built on industrial, object-oriented languages, which provide modularity in the programming world, but whose specifications rely on mathematics that do not support these goals. On the other hand are extensible, generic mathematical systems that are not integrated with programming languages that support component reuse. The result, on both sides, is the creation of components insufficiently generic and extensible to be reused.

As we wish to verify components of increasing complexity, we must build these components out of smaller subcomponents. If these subcomponents display poor modularity, they will compose poorly or not at all, resulting in complex interactions and proof obligations that are difficult to satisfy. To date, modern systems have addressed this complexity by placing the onus of verification on a suite of sophisticated, industrial-strength automated theorem provers that are on the bleeding edge of artificial intelligence design.

This seems paradoxical, however, as programmers do not often rely on deep mathematical results when they reason about the correctness of components. We posit that by shifting the burden to the design of good components and specifications, as supported by a flexible mathematical and specification subsystem, proof obligations should become much more obvious and more easily-proved. In addition to influencing the design of our own system, RESOLVE, such exploration would benefit other systems as well, by informing specification and theory design across the board.

The intent of this proposal is three-fold. First, to develop a flexible mathematical framework for program specification designed with modularity and reuse in mind. Second, to experiment with the design and implementation of a minimalist prover sufficiently flexible to operate on that mathematical framework and determine the actual practical requirements for program verification of well-specified components. Third, to develop a diverse library of components specified using a variety of specification styles in order to identify best-practices in specification engineering by measuring the complexity of resultant VCs.

Description of Proposal Area

A verifying compiler ideally eliminates the need for testing to reveal functional bugs by accomplishing what testing cannot: demonstrating the *absence* of any bugs. Unlike testing and informal reasoning, formal verification demonstrates that code behaves as specified under every possible valuation and along every possible path of execution.

Unfortunately, a number of limitations to full, automatic verification mean that testing remains the norm for quality assurance. As demonstrated by a number of high-profile software disasters in the past decade, much mission-critical software, upon which human lives may depend, has errors despite rigorous testing. Only verification can demonstrate that code meets its specification exactly.

Verification operates by viewing code as a series of mathematical transformations and comparing the result with some formal specification of desired behavior. As a result of this, an important component of any verification system is an automated theorem prover, which takes mathematical statements as its input and attempts to prove them automatically. However, proving a mathematical statement in general is an undecidable problem and in many modern systems this process must be assisted by a human expert working interactively with the proving system.

Because of the complexity of the problem, these automated provers are on the cutting edge of algorithmic and artificial intelligence design and represent the current bottleneck in automated software verification. Since mathematical statements are generated from the program's specifications and the specifications are written in the language of the mathematics of the system, the choice of mathematical model has a direct impact on the success or failure of an automated proof attempt. Often, specifying code one way will yield automatically-verifiable code, while another equivalent specification will not.

The goal of a mechanical verifier has existed since the 1960s and the earliest days of computing[18]. Despite this, modern verification systems require tremendous effort on the part of highly-trained mathematicians and software specialists to dispatch the proof obligations resulting from even simple programs. Ameliorating this bottleneck by experimenting with design decision upstream from the prover is the focus of this research.

Explanation of Problem

A number of open theoretical, practical, and design problems exist in the area of mechanical verification. Despite a large variety of tools and techniques for verification, a number of these problems span all kinds of systems. One is the issue of scalability and modularity—to justify the effort required to verify the component, it must enjoy broad reuse so that said effort may be amortized over time.

This research proposes to experiment with factors that contribute to reuse. We hypothesize that well-engineered, modularly verified code and extensible, reusable mathematics should contribute to more easily verified client systems. Unfortunately, current verification tools do not generally provide the kinds of tools required to test this hypothesis. Confirming or rejecting this hypothesis would benefit many existing systems by informing the style of specification used and the choice of where in the verification life-cycle to focus effort to achieve best results. To successfully explore the problem, a confluence of a flexible mathematical system for specification, an associated prover, and a well-designed library for experimentation is required.

The goal of this dissertation is to experiment with a modularity-focused specification system as well as the consequences of such a design, including the use of a more minimalistic, general automated prover.

1 Introduction

The verifying compiler is a grand challenge in computing, perhaps most famously stated by Tony Hoare in 2003[19], but based on research into program correctness stretching back to the 1960s[18]. Such a compiler would ideally eliminate the need for testing to reveal functional bugs by accomplishing what testing cannot: demonstrating the *absence* of any bugs. Unlike testing and informal reasoning, formal verification demonstrates that code behaves as specified under every possible valuation and along every possible path of execution.

As a powerful demonstration of the weakness of traditional testing and informal reasoning, we consider the case of binary search. Binary search is a simple, well-understood, and widely-implemented algorithm. Yet, Joshua Bloch of Google Research wrote a blog post[4] in 2006 about a subtle bug in the standard Java library's implementation of binary search—an implementation that had been in place for nine years and was based on a version of the algorithm "proven" correct (via informal reasoning) by Jon Bentley of Carnegie Melon University in his famous *Programming Pearls*[3]. Certainly, such a straightforward implementation of such a simple algorithm, backed, as it was, by a proof from a respected algorithms guru and in wide deployment for nine years would be considered by most as a *mature*, *well-tested* component—one suitable for use in critical deployments. And yet it contained a subtle, latent overflow bug revealed not by a nit-picking graduate student but by a client in the wild whose code broke when the component failed to meet its specification (specifically causing an `ArrayIndexOutOfBoundsException`, which, in a C or C++context, would be a recipe for a buffer overflow attack in addition to a potential crash.) This bug, so subtle and resilient against traditional testing and human reasoning, becomes extremely shallow under formal reasoning, where bounded, programmatic integers are well-specified in a mathematical way.

Several systems for formal verification exist, including some built on Java which may have caught this and other bugs. These systems are traditionally built as a pipeline in which code and its associated specification are translated into an intermediate *assertive code*, which is then translated into a series of *verification conditions* (VCs), which express the proof obligations of the code in a purely mathematical way, before finally being sent to one or more *automated provers* which attempt to dispatch the VCs. This process is illustrated in Figure 1.

Despite many successes, all existing systems require a great deal of effort, either in the form of interactive proving or carefully contrived hints to an automated prover, in order to verify even simple programs. This is counter-intuitive since most programs contain straight-forward logic that the programmer feels assured of without calling upon complex mathematical reasoning. The problem

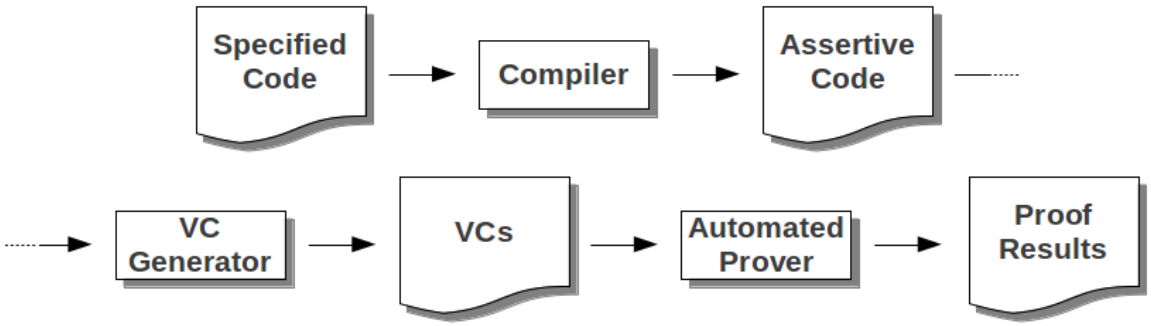

Figure 1: The Verification Pipeline

is compounded by a lack of integration between between modern, modular programming languages and expressive, flexible mathematics that prevents the design of reusable components, which could amortize this intense verification effort over time. We hypothesize that a well-integrated, flexible and extensible mathematical and specification subsystem would permit specifications that more closely reflect the programer's intuition and the usual patterns of reuse, resulting both in more straightforward proofs and more generic, longer-lived components. This proposal seeks to develop such a system and test this hypothesis.

1.1 Existing Systems: Practical vs. Pure

Existing verification systems largely fall into two categories: those with a focus on practical, automated verification and those with a focus on pure mathematics. Examples of the former include Jahob[27], based on Java; Spec#[1], based on C#; and ACL2[21], based on Lisp. Examples of the latter include Coq[31], based on the Calculus of Inductive Constructions; and Isabelle[33], based on a higher-order, intuitionistic logic. Practical systems often take advantage of a limited, hardcoded mathematical universe that corresponds closely to or is conflated entirely with programming constructs. Pure systems permit an extensible, flexible mathematical framework with clear separation of programming concepts from mathematics. Because of their narrower focus, the former often permit easier mechanical verification than the latter[1], which tend to emphasize interactive user-guided verification.

[1]Some systems even permit efficient decidable verification algorithms for certain domains or properties. See, e.g., information on SplitDecision in [36].

1.1.1 Example Practical System: Jahob

Let's consider a version of Java's `ArrayList` verified in Jahob[2]. Jahob combines code written in a subset of Java with specifications written in Isabelle. Listing 1 shows the `contains()` operation of an ArrayList.

Listing 1: ArrayList.contains()

```
public boolean contains(Object elem)
    /*: requires "init"
        ensures "(result = (EX i. (i, elem) : content))";
    */
    {
        int index = indexOfInt(elem);
        //: noteThat PosIndex: "0 <= index --> ((index,elem) : content)";
        //: noteThat NegIndex: "index = -1 --> ~(EX i. (i,elem) : content)";
        //: noteThat IndexLemma: "0 <= index | index = -1";
        boolean res = (0 <= index);
        /*: note ResultLemma: "res = (EX i. (i, elem) : content)"
                from PosIndex, NegIndex, IndexLemma;
        */
        return res;
    }
```

The list is modeled as a set of $(index, element)$ pairs. Assertions are provided inside Java comments (meaning that Jahob-specified Java programs remain compilable using a standard Java compiler) that begin with a colon. Note that mathematical assertions are set off in quotes, a syntax inherited from Isabelle. `init` is a predicate defined elsewhere in the class. Jahob encourages the inclusion of inline "hints" targeted at the backend provers[47] (of which it supports many) and we see several of those here interspersed in the Java code.

Utilizing a suite of provers, the Jahob system is able to dispatch the resultant VCs and yield a fully-verified `ArrayList` implementation suitable for generic use—an impressive feat. In fact, in a recent result, the Jahob team verified a handful of different linked data structures[46]. A number of design decisions support this ability. First, Jahob has a flexible set of syntactic tools for specification, including pre- and post- conditions, auxiliary variables[22], and conceptual definitions that permit an intuitive specification. Second, while Jahob supports higher-order specifications, where possible it uses a standard first-order logical representation that can be translated into the input format of

[2]Jahob does not yet work with Java generics and so the version of ArrayList verified is the pre-Java-1.5 version that operates on `Objects`.

multiple back-end proving systems including CVC3[2] and Z3[9]. Those specifications that cannot be translated down to a first-order representation can still be sent to Isabelle for interactive proving. By using multiple prover backends, Jahob can take advantage of multiple proving paradigms including interactive, algebraic, boolean satisfiability (SAT) solvers, and efficient decision procedures. Third, as already stated, Jahob permits in-line mathematical assertions that allow the programmer to guide backend provers by stating useful intermediate results.

However, despite this success, a number of the realities of this system (and others like it) are counter-intuitive and seem geared toward short-term verification of small Java programs rather than long-term, scalable verification of complex, modular applications.

1.1.1.1 Complex Proof Obligations for Simple Methods.

When using the Jahob system, VCs that result from even simple programs are often extremely complex. Jahob's intermediate VC syntax is not intended to be human readable, so we do not reproduce any VCs here, but we can get a feel for their complexity via sheer volume: the three-line boolean method `contains()` generates VCs that span over 150 lines[3].

A number of factors contribute to this VC complexity. First, Jahob is built on top of an existing programming language that includes many features that are not amenable to verification, including null pointers and uncontrolled aliasing, both of which complicate reasoning[43]. These complexities must be accounted for in its VCs. Second, Jahob's inline assertions must themselves be verified to preserve soundness. The intent of these assertions is that they simplify the proving process by proving intermediate steps, creating additional (presumably easier) VCs that in turn lower the difficulty of the original proof obligations. Still, why intermediate results should be required for a simple `contains()` method is unclear. Third, while perhaps subjective, the system encourages the use of awkward mathematical models for components. In the list example, the choice of a set for mathematical model requires additional invariants to establish that, for example, no index in the list appears twice with different elements. Presumably a set was chosen because it was the closest mathematical object that already had a well-developed Isabelle theory, but in an ideal world such a system would provide a first-class, integrated mechanism for extending the mathematical universe so that a more directly analogous mathematical object could be used—perhaps a function mapping or a finite sequence abstraction.

[3] When formatted to standard 80-character lines.

1.1.1.2 Lack of Support for Modular Design.

Jahob stands nearly alone amongst the practical systems for supporting higher-order mathematics. However, practical issues with the integration of Isabelle into Java hamstring the usefulness of this feature. Among the Java features not supported by Jahob are Java generics and dynamic dispatch. These unsupported features preclude many important patterns of reuse that the mathematics of Isabelle are otherwise ready to support. Components cannot be parameterized from the outside with definitions (this is not supported directly and programmatic work-arounds like the Strategy and the Template design patterns rely on dynamic dispatch.) Data structures cannot make parameterizable guarantees about the properties of contained elements (which would require Java generics.) Indeed, the majority of the polymorphism pillar of object-orientation is precluded, seriously reducing the reusability of components and thus the amortization of verification effort over time.

A particularly illustrative example of this lack of modularity appears in a `Map` data structure verified by the Jahob team in [46]. The trouble of aliased keys is dealt with by specifying that *all objects* are immutable with respect to the built in Java `hashcode()` method—a restriction that does not appear in the original `Object` contract and further implies that all objects are immutable with respect to `equals()`. The correctness of the `Map` implementation requires changes to external components, rather than being part of its self-contained specification. We discuss this problem in more detail in [7].

With lengthy, complex VCs, syntax for guiding back-end provers, and a system that encourages a trade-off of mathematical flexibility for prover diversity, the Jahob design seems to assume that the onus of verification is on the provers. It shifts the complexity of verification to the part of the pipeline highlighted in Figure 2. However, as the strength of automated provers is the current bottleneck of verification, this requires a great many sacrifices to the limitations of this bleeding-edge part of the verification toolchain.

1.1.2 Example Pure System: Coq

Let's consider a recursive implementation of the integer division operator, specified and implemented in Coq. Despite the fact that Coq is a mathematical system, unable to execute code, we could use its ability to "extract" a program into various target languages[4] for execution. Coq separates specification from implementation; in Listing 2 we see a pair of predicates for specifying integer division.

[4] At time of writing: OCaml, Haskell, and Scheme.

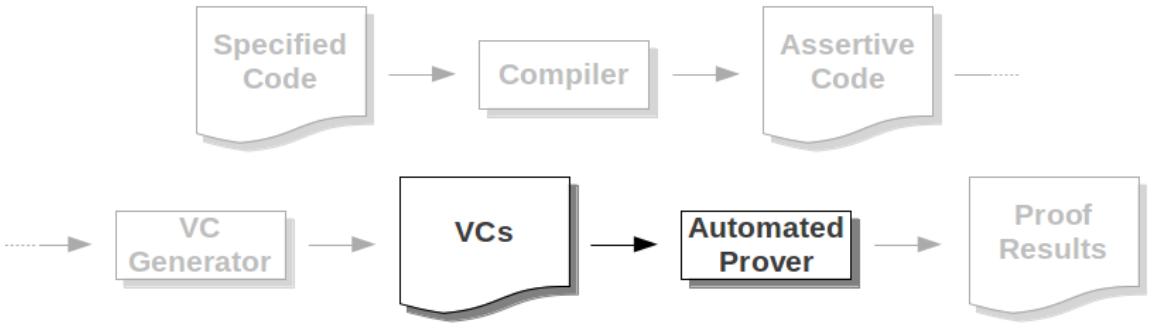

Figure 2: Shifting the Burden to the Prover

Listing 2: Division Predicates

```
Definition divPre (args:nat*nat) : Prop := (snd args)<>0.

Definition divRel (args:nat*nat) (res:nat*nat) : Prop :=
    let (n, d):=args in let (q,r):=res in q*d+r=n /\ r<d.
```

The `divPre` predicate represents the precondition on division—that the second argument is not 0. The `divRel` predicate specifies the relational behavior of division—that the result will consist of two natural numbers, `q` and `r`, such that `q * d + r = n` and `r < d`, i.e., `q` is the quotient and `r` the remainder.

We then implement division recursively as shown in Listing 3.

Listing 3: Division Implementation

```
Function div (p:nat*nat) {measure fst} : nat*nat :=
    match p with
    | (_,0) => (0,0
    | (a,b) => if le_lt_dec b a
               then let (x,y):=div (a-b,b) in (1+x,y)
               else (0,a)
    end.
```

Coq's `Function` keyword introduces a recursive function with an appropriate implicit fixpoint. The `measure` keyword provides a progress metric—namely, that `fst` is decreasing. Note that, despite the fact that an input matching `(_,0)` is disallowed by the precondition, Coq does not permit non-total functions, so we provide a nonsense return value for this case. The `le_lt_dec` is simply a less-than-or-equal-to predicate on natural numbers. Defining this function immediately raises a termination proof obligation, which one of Coq's built-in proof scripts can dispatch automatically.

To demonstrate the correctness of the implementation, we can assert the theorem given in Listing 4. That is, that for all inputs, if the inputs meet the precondition, then the behavioral relation holds between the inputs and the result of applying `div`.

Listing 4: Division Functional Correctness Theorem

```
Theorem div_correct : forall(p:nat*nat), divPre p -> divRel p (div p).
```

Proving this theorem is more complicated than the termination proof and none of Coq's built-in tactics can dispatch it automatically. Instead we enter interactive mode and prove it live with the sequence of tactics given in Listing 5.

Listing 5: Division Functional Correctness Proof

```
unfold divPre, divRel.
intro p.
functional induction (div p); simpl.
intro H; elim H; reflexivity.
replace (div (a-b,b)) with (fst (div (a-b,b)),snd (div (a-b,b))) in IHp0.
simpl in *.
intro H; elim (IHp0 H); intros.
split.
change (b + (fst (x,y0)) * b + (snd (x,y0)) = a).
rewrite <- e1.
omega.
change (snd (x,y0)<b); rewrite <- e1; assumption.
symmetry; apply surjective_pairing.
auto.
Qed.
```

A high-level sketch of this proof is that it applies induction by cases, proving that each of the `match` branches and each of the `if` branches maintains the correctness of the implementation. Definitions are repeatedly expanded to take advantage of their hypotheses. Obviously, however, this syntax is far from readable without being intimately familiar with Coq and certainly looks nothing like a mathematical proof as it might be conceived by a mathematician.

Systems like this have been used to excellent effect verifying complex programs. Coq has been used to verify a C compiler[30], while Isabelle has been used to verify an operating system kernel[25].

This success is due to a number of factors. Unlike in the practical programming systems, Coq and other pure systems provide a rich, extensible mathematical universe permitting higher-order logic and user-created mathematical theories. This enables a hierarchy of abstraction similar to the

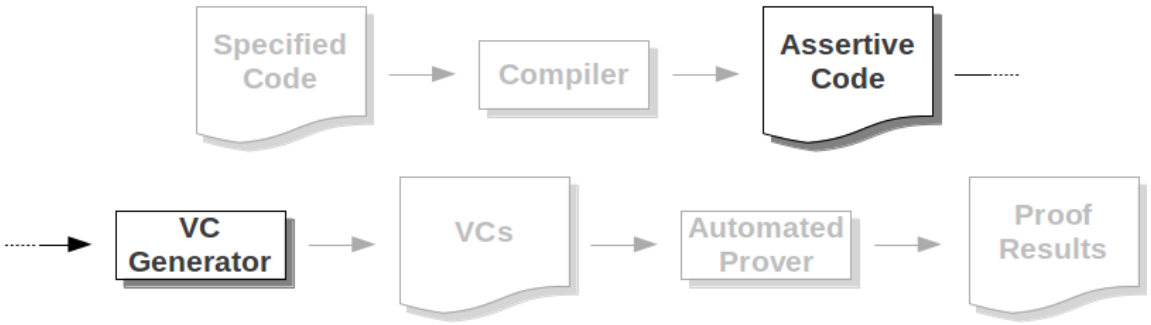

Figure 3: Shifting the Burden to the VC Generator

development of object oriented code in which complex mathematical objects are repeatedly decomposed into smaller and smaller mathematical objects and an "interface" of theorems is provided for working with the high level objects. In addition, Coq's programming model is functional, eschewing a number of complexities pervasive in industrial languages—pointers, aliasing, and referential opacity to name a few. Automated proof systems are used to jump small steps, but interactive mechanisms are provided for splitting complex proof obligations into multiple smaller tasks. It's as though the human user becomes part of the assertive code and VC generation step, pointing out useful decompositions of existing proof obligations until they become small enough that the automated prover can take it from there. In a sense, such a system shifts the onus of the verification process to the part highlighted in Figure 3.

Unfortunately, given the value of a highly-trained mathematical and programming professional's time, a user-guided strategy is likely cost-prohibitive if all but the smallest proof obligations must be discharged by hand. A number of factors contribute to the difficulty of these proof obligations.

One is that, in order to exploit the flexibility of the mathematical system, the automated prover must be more general, unable to take advantage of the numerous domain-specific tricks that cutting-edge narrowly-applicable theorem provers exploit. Another is that the divide between the mathematical world of Coq and the programming world of an industrial language is large. We see an example of this in the division example, where time and effort must be expended proving that the behavioral relation is total (even though the method it specifies is not!) and that an irrelevant program branch maintains invariants.

Compounding this problem, pure systems have no awareness of the underlying programmatic structure they are being applied to and thus inherently view verification as operating on procedural rather than object-oriented code. Modular mathematics are therefore not applied to modular code and the result is that, impressive as a verified compiler is, the next complex component must be

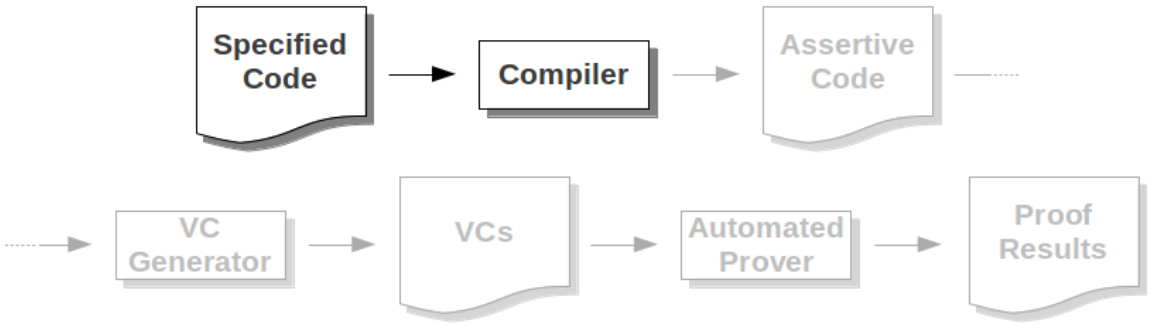

Figure 4: Shifting the Burden to Specification

verified from scratch as it is unlikely that any component from the compiler will be generic enough to be reusable.

1.2 Best of Both Worlds?

At the core of this research is the question of whether or not a system that combines the best parts of practical systems and the best parts of pure systems might permit specifications that are more amenable to automatic verification. A programming system that eschews certain complexities of industrial languages and avails itself of a tightly-integrated, flexible and extensible mathematical system designed to work with it could be used to create modular components based on modular mathematics. We hypothesize that in such a system, a focus on modularly verified components would yield less complex proof obligations while at the same time encouraging the creation of reusable components that amortize the verification effort invested in them over time. Such a system would place the onus of the verification on the design of modular specifications supported by a flexible compiler, i.e., that part of the pipeline highlighted in Figure 4. Highly trained programmers and mathematicians are still required, but the fruits of their labors will be generic and reusable, unlike proofs of correctness.

Developments such as these would also benefit existing systems of all types. Increased understanding of the importance and techniques of modular reasoning can be applied to systems like Jahob to increase the provability of large components via better-engineered subcomponents. A richer understanding of the importance of levels of mathematical abstraction to long-term verification goals would assist those working in pure system like Coq in making better up-front choices to pay long-term dividends.

1.3 Problem Statement

This proposal seeks to determine if the mechanical verifiability of software components can be improved by better-engineered mathematics and specifications and, if so, explore which techniques might best support these goals. In the process it will address the following open problems in the area of formal methods:

- Design of an extensible, flexible mathematical framework and a well-integrated associated specification framework, supporting verifiability by allowing specifications based on the *best* mathematical model, rather than simply the most convenient, and supporting scalability through mathematical reuse.

- Architecture of and experimentation with a minimalist rewrite prover to support reasoning in the above framework and determine those prover capabilities practically necessary to mechanically verify well-engineered, modular components.

- Creation of a diverse library of components of the sort found in standard programming libraries, specified and modeled using a broad set of techniques to enable experimentation on mathematical and specification best-practices for practical verification.

1.4 Research Approach

Building on previous work on specification language design, VC generation, and automated prover development, this research seeks to augment the existing RESOLVE[36] system with a flexible mathematical subsystem and modular specification subsystem in order to experiment with the specification and mathematics best-practices in support of modularity.

Unlike practical systems, where bringing modular mathematics to bear in support of modular programming is difficult at best and where special-case constructs like references cause complex reasoning even in situations where they ought not be relevant, the mathematical system proposed here will be tightly integrated and based on an extensible Morse-Kelley Set Theory, extended to include higher-order definitions, and will permit specification of programmatic constructs (e.g., references) only as modeled in that pure mathematical system.

Unlike pure systems, where the specification system that bridges between mathematics and programming is ad-hoc and not designed to take advantage of the structure of a program, the specification system proposed here will cooperate with the mathematical system by design, thus

permitting it to take advantage of the same modularity and genericity used in a well-designed component.

Armed with such a system, we will test its flexibility by designing, specifying, and implementing a library of programming components ranging from simple stacks to more complicated tree structures, along with the algorithms for manipulating these structures. These structures and algorithms will be modeled and specified using a variety of techniques in order to gain insight into how these techniques impact verifiability and scalability. We will analyze resulting VCs to determine which prover capabilities are strictly necessary, designing and building a minimalist rewrite-based prover based on a plug-in architecture to support only those capabilites necessary for practical VCs.

The evaluation of the system as a whole will include classification of VCs by the minimalist prover based on a number of proof metrics including number of required theorems, number of required proof steps, and proof time; as well as subjective, qualitative metrics derived from field tests with programming and mathematics students and professionals.

1.4.1 Contributions

Such a system would address several open problems in the area of formal methods, both practical and theoretical:

First, a flexible, extensible, and intuitive mathematical subsystem would significantly improve on a key component of the verification tool chain. The day when an AI can infer the intent of a program and verify it without rigorous specification and significant mathematical development is extremely distant. Until that time, formal verification will be a joint effort between highly trained programmers and mathematicians. While improvements are being made[45], the current generation of specification systems (both practical and pure) largely use ad-hoc mathematical syntax and esoteric mathematical foundations that programmers may find convenient but mathematicians find confusing and unweildy. Beyond simply engendering collaboration between programmers and mathematicians, such a math-centered language design will encourage the full body of modern mathematical developments to be used directly. Additionally, since it is not grounded in any programming constructs, it will not be tied to any particular language and may itself be used as a component outside of RESOLVE.

Second, a well-integrated and flexible specification and mathematical system would open up the development of verified components to modular, reusable techniques by providing first-class syntax and flexible mathematical semantics for mapping such programmatic ideas into the mathematical realm. As an example, the lack of useful higher-order logic in practical systems prevents components from being designed to take mathematical assertions or abstract operations as parameters, severely

limiting one dimension of reuse: genericity[7]. Our hypothesis is that a well designed, modular system should better exploit programmer intuition and allow for more straightforward proof obligations, permitting slower, but more expressive, automated provers to be used. The development of such techniques would be a boon to existing systems, as well. For example, as part of our research so far, we have explored the concept of quantifier elimination and techniques that can be used to specify and verify in their absence. In a recent paper from the Jahob team, we find a quote highlighting the need for such research, here in the context of an attempt to verify an implementation of a Java `ArrayList`: "Unfortunately, the provers are unable to automatically prove the post-condition of remove. What makes the problem...difficult is that the assumptions contain universally quantified formulas while the post-condition contains an existentially quantified formula." Our research on engineering mathematics addresses this and other issues and may be helpful in elliminating such complications.

Thirdly, such a system will permit the above hypothesis to be tested and demonstrated in a rigorous, mechanical environment. The current state of the art in verification suggests that verification is hard because programs are complicated. We believe that well-engineered programs are not complicated and that, by extension, if augmented with well-engineered specifications, the resulting proof obligations should not be difficult. Thus, a system designed to support well-designed programs is more important to successful verification than one designed to support the limitations of a state-of-the-art prover. What such a design entails and what it means to be a "well-engineered specification" are open questions addressed by this research.

1.5 Thesis Statement

In a verification system, an extensible, flexible mathematics and specification subsystem enables better-engineered component specifications and thus more straightforward proof obligations that are more easily dispatched by even minimalistic automated theorem provers.

1.6 Proposal Organization

The remainder of this proposal is organized as follows: Section 2 gives an overview of the state-of-the-art in each of the problem areas, with information on work related to this research, Section 3 provides background on the RESOLVE verification system that is the platform for this research, Section 4 presents the proposed research including work already completed, Section 5 explains the criteria against which the work will be evaluated, and Section 6 will offer some concluding thoughts.

2 Verification Background and Related Work

In order to justify the proposed research as well as place it in context, it is important to consider the full breadth of available mathematical systems, provers, and verification libraries. In the following three sections, we break related work and necessary background down along those lines.

2.1 Specification and Mathematical Systems

Early systems for program specification included Z[10] and Larch[14]. Both are first-order predicate logics not grounded in any particular programming system, though each has been adapted for many different langauges. Larch provides a two-tiered specification style to ease adapting it to a new language. Because it is based on Zermelo-Fraenkel set theory, Z is one-sorted (i.e., it has only one "type": the *set*). Larch, on the other hand, is multi-sorted, but assumes all sorts are disjoint (i.e., it does not permit sub-"types", among numerous other "type" relationships.) These restrictions complicate the mapping of programming concepts in modern languages to their mathematical universes. Modern systems for specification in practical systems often address the limitations on sorts, but, as we will see, the first-order restriction remains ubiquitous, severely restricting the genericity (and thus the reusability) of mathematical theories and specifications, as discussed in some detail in Section 4.1.1.1.

An extremely straightforward method of specification in practical systems is to permit them to be written programmatically in the target language–i.e., the specification is *executable*. This has the immediate advantage of permitting dynamic checks to be compiled into the code if static verification is not possible. However, it raises a number of disturbing possibilities including specifications that have side effects or that require proofs of "termination". In addition, if the language in question does not have a formal semantic, then the specification is only as clear as our understanding of the meaning of the underlying language. A number of systems use this method as their basis (though in some cases they offer non-executable extensions).

One prominent example of this style is the Java Modeling Language (JML)[28]. The majority of functions used in a JML specification are, in actuality, Java methods. However, in order to defeat the side-effecting problem, they are required to be declared *pure*, i.e., they must be demonstrably side-effect free (this does not, however, speak to their termination.) "Model methods" may also be provided, which define a method for the purposes of specification, but contain no code—however, their behavioral specification is often still expressed in terms of (non-executable) Java code (i.e., in the formal comment there is commented-out Java code expressing the behavior of the model

method.) The logic of JML is first-order (i.e., we cannot quantify over methods or types), but we are able to take advantage of the flexibility of including "code" in our mathematical specification to achieve higher-order methods for the purpose of, for example, passing them as a parameter[5].

Because of the tight integration between "specification" and "code" in JML, the mathematical realm becomes restricted to ideas expressible in Java. E.g., the only type-relationship are Java type-relationships. While this simplifies the mathematical realm and makes it consistent with the programmer's understanding, we posit that the necessary complexity of mathematical development required for true mathematical modularity necessitates a mathematical specialist. Thus, the mathematical realm should target the universe familiar to such a specialist, rather than cater to a programmer. Many restrictions necessitated by the computability requirements of code are irrelevant in the mathematical sphere and are both arbitrary-feeling (e.g., no multiple subtyping) and lead to a decrease in the allowable genericity of mathematical theories, curtailing their reuse.

Another system of specification along these lines is Why[5]. Why is ML-like and focussed on reasoning about built-in structures like arrays rather than general data abstractions. By using a variety of translation front-ends, Why can be an intermediate specification and programming representation for different programming languages including Java, C, and ML.

JML and Why are not alone in using this specification style: additional systems in this style include the specification language of Spec#[1], built on C#, and ACL2[21], built on a dialect of Common Lisp.

Another popular style of specification in practical systems is separation logic[34]. Because it models the heap directly, this style provides mechanisms for reasoning about aliasing, pointer arithmetic, et al. In order to bring these complex problems into the range of modern theorem provers, separation logic allows operations to explicitly state which areas of the heap they will modify, along with which arbitrary properties they expect the heap to maintain while they operate. This requires the heap to be modeled in its entirety, along with specialized logical systems for reasoning about this model. A notable system focussed on this style is Verifast[20], a system for verifying Java and C programs that uses separation logic expressed in a custom specification language and targets the Z3 prover.

A third style of specification in practical systems is to use a strictly separate mathematical language for specification. Such systems often provide more generality than others, since the mathematical language can usually be arbitrarily extended to permit new bases for mathematical models and provide generalized machinery for reasoning about these new classes of mathematical objects.

[5]We effectively are able to include an instance of the *Strategy* design pattern in our specification.

Examples of this kind of system are Jahob, the RESOLVE system developed here at Clemson, and the RESOLVE system being developed at The Ohio State University. As already discussed in Section 1.1.1, Jahob uses Isabelle as its specification language, granting it higher-order logic and a rich theory of sorts for free. However, the usefulness of these features is hamstrung somewhat by a lack of support for the associated programmatic features in Java. RESOLVE strives to provide a sorted, higher-order mathematical language, though the complete design and implementation of this language is part of the topic of this proposal. OSU's RESOLVE implementation does make a clear delineation between mathematics and programming, and permits higher-order definitions, but does not permit the mathematical universe to be extended, reasoning, instead, about a series of built-in sorts.

An interesting comparison of some additional practical systems with respect to their specification and reasoning systems can be found in [17].

Amongst the pure systems, the style of specification and mathematical representation becomes more uniform, at least in their almost-universal inclusion the features we feel are crucial for a component specification platform (higher-order definitions; first-class types; and a rich, extensible theory of sorts.) By far the most popular systems are Coq, described in detail in Section 1.1.2, and Isabelle.

Isabelle is notable for providing a very small logical core from which all other theories and logics are built, permitting the system to be soundly extended with diverse logics for which it was not designed. For example, the most popular variant, Isabelle/HOL, Isabelle with Higher Order Logic, provides multi-sorted higher-order logic, expressed in a reasonably intuitive mathematical syntax called Isar[45].

2.2 Automated Theorem Provers

We may broadly partition automated theorem provers into three categories—those based on decision procedures, those based on boolean satisfiability (SAT)[6], and those based on term rewriting. The distinction between these is not clear-cut, however, as many (indeed, most) systems combine all three to some extent, using term rewriting as a preprocessing step before passing the problem off to a SAT-solver or a decision procedure, or using a feedback loop between a term-rewrite prover and a SAT-solver.

[6]Technically, SAT is a decision problem with finite solution space, and thus its solvers are decision procedures. However, this problem is NP-hard, so we use "decision procedure" here to mean an algorithm that is *efficiently decidable*.

2.2.1 Decision Procedures

The decision procedures are the least flexible but often the most efficient. They take advantage of information about an extremely specific domain to arrive at conclusions quickly. Examples include linear arithmetic solvers of the kind found in Z3[9] and OSU's SplitDecision[36] system for reasoning about finite sequences. Because they exploit domain-specific information, these systems are necessarily hard-coded and cannot be generalized to new domains.

2.2.2 SAT-Solvers

Provers in this class simply attempt to find valuations of boolean variables to satisfy a given formula. While in general this is NP-hard, a number of branch-and-bound techniques can be used to drastically reduce the problem space for practical formulae. Almost universally these use the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, which is a constellation of specific heuristics and refinements that reduce the search space considerably in many cases. For a good overview of this algorithm along with some recent refinements, the interested reader is directed to [32].

A number of SAT-solvers exist. Z3 and Yices[11] are popular provers based at their core on SAT-solvers. It may seem difficult, at first, to apply provers for boolean formulae to complex programmatic proof obligations, but many techniques exist for translating complex domains into equisatisfiable boolean representations. Indeed, because of the blazing speed and mature implementations of SAT, such translations are an active area of research (see, for example, [13] and [35].) SAT-solvers that extend themselves to include the Maximum Satisfiability (Max-SAT) problem are able to provide useful debugging information in the form of counter-examples: indicating precise valuations under which a theorem does not hold. Yices is an example of a SAT-solver with this capability. The general applicability of boolean logic (particularly in the field of computing) makes SAT-solvers significantly more general, since they may be applied to any theory for which a translation into boolean logic exists.

2.2.3 Term-Rewrite Provers

These provers are the most flexible, but also the most difficult to optimize. They are by far the most similar to how a human being would proceed with a proof—by matching theorems against a set of known facts, transforming them until we derive the result we want. This general pattern-matching means that they can be applied to any domain about which a set of theorems has been established. An excellent example of this kind of prover being used in a practical-style system is the prover

used in the ACL2 system. It utilizes a complex series of hints, provided by the person developing the theory, about which theorems should be applied in which way, then applying them sequentially without backtracking. This differs from the approach of our minimalist prover—where we employ backtracking and eschew encoding information about how the prover should apply theorems, as we would prefer mathematicians reason about mathematics rather than about how a particular prover will apply it. Additionally, while our rewrite-prover strives to operate over our distinct mathematical subsystem, ACL2's mathematics are much more tightly bound with its underlying language, Lisp. A noteable feature of the ACL2 prover is its ability to automatically discover inductive proofs.

Many SAT-solver-based provers, including Z3 and Yices, employ term-rewriting as a preprocessing step to simplify formulae or translate them into a heuristically-convenient form.

2.3 Benchmarks, Libraries, and Specification Engineering

A number of verification benchmarks and libraries exist, and at least one verification effort has tangentally acknowledged the specification engineering question.

The Jahob team has made a concerted effort to verify a number of data structures in a form as close to how they appear in the Java standard library as possible. In a recent paper [46], they published about a subset of these which were linked data-structures, including `ArrayList`, `HashMap`, and `PriorityQueue`. While small (on the order of ten components), this library represents the sort of effort we would like to make here—a library of components geared toward an exploration of a specific facet of verification. This library is geared towards linked data structures and the expressiveness not to have to compromise a component design from what we see in normal industrial systems.

Additionally of interest from the Jahob team is a recent paper[6] that deals directly with specification engineering, noting that quantifiers consistently complicate verification and engineering the specifications of a set of data structures to do without, enabling them to be translated down to a first-order logic, then passed to an efficient first-order prover. This result, however, focussed on the translation process rather than the novelty of the importance of specification style. To our knowledge, the team has not embarked on a systematic exploration of this facet of verification, nor do they have any plans to create a library focussed on this dimension.

Since 2010, the Verified Software: Tools, Theories, and Experiments conference has run a verified software competition. Attendees are permitted to enter and are given a short amount of time to complete a handful of challenges. Afterwards, the various solutoins are made public along with

discussion by their implementers.

The nature of these challenges have been all over the map, ranging from averaging an integer array to implementing a binary tree data structure. The intent has been to get verification projects communicating and sharing ideas rather than our purpose here: to compare different styles of specification.

This competition has revealed some interesting realities about modern verification systems *vis-á-vis* modularity. For example, in 2010, despite the fact that an earlier challenge required teams to create a list data structure, no team used that data structure in a subsequent challenge to implement an amortized queue with a linked list, instead re-implementing a linked list, presumably because they required different properties to achieve verification.

In 2008, researchers here at Clemson and at the Ohio State University compiled and published[44] a set of incremental benchmarks intended to be representative of the breadth of verification complexity, starting with simple integer addition and spanning system I/O, design patterns such as Iterator, and finally an integrated application. The focus of these benchmarks was on demonstrating the capabilities considered essential to any verification system that was to be useful in practice. At a subsequent VSTTE conference, the Microsoft verification team published their solutions to some of these benchmarks as implemented in Dafny[29], which revealed a number of interesting properties of that system. Our hope is that the library developed for this research can have a similar effect, eliciting discussion and comparison of different verification systems by encouraging other verification efforts to explore the effect of differing specification on properties of the verification.

3 RESOLVE Background

The RESOLVE[37] Verifying Compiler is an attempt to create a full verificatin pipeline, beginning with an integrated specification and programming language and continuing through to a back-end prover. The focus of RESOLVE is on modular verification, which attempts scalability by ensuring that each component is verified in isolation and need not be re-verified regardless of deployment. This means that encapsulation must be strictly assured and thus certain treacherous programmatic constructs such as aliases must be tightly controlled.

As an example, let's consider a Stack abstraction. We are interested in designing a system for complete verification—including constraints like memory, so we will use a bounded stack that takes as a parameter its maximum depth. Listing 6 shows part of a RESOLVE concept describing such a component.

Listing 6: A Stack Concept

```
Concept Stack_Template(type Entry; evaluates Max_Depth: Integer);
    uses String_Theory;
    requires 1 <= Max_Depth;

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

    Operation Push(alters E: Entry; updates S: Stack);
        requires |S| + 1 <= Max_Depth;
        ensures  S = <#E> o #S;

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

    (* Other operations omitted for brevity. *)
end;
```

RESOLVE permits parameterized types. In this case `Entry` is a generic type, permitting `Stacks` to operate over any other RESOLVE type, and `Max_Depth` is an integer indicating the maximum number of elements that can ever be on the stack. The `evaluates` parameter mode indicates that the parameter is pass-by-value (parameters are ordinarily pass-by-reference.)

Applying modular lessons from the world of programming to mathematics, we store theories in separate files with their own scopes that can be imported individually[41]. The *uses* line includes one such file—the theory of strings, i.e., finite sequences.

The `Family` clause introduces a family of abstract types called `Stacks`, modeled conceptually by strings of `Entrys`. `exemplar` introduces a name to be used for a hypothetical stack, which is then used in the `constraint` clause to indicate that not *all* strings of `Entrys` are valid stacks, but rather only those of length `Max_Depth` and less. The `initialization` clause notes that all `Stacks` are assured to be empty when they are first created.

We then define some operations on `Stacks`—`Push()` and `Pop()`, with their associated pre- and post-conditions, introduced by `requires` and `ensures` clauses, respectively. Parameters to operations are preceded by *parameter passing modes*, which summarize the effect the operation will have on a parameter—a parameter that's in `alters` mode is passed a meaningful value that will be changed to an arbitrary value by the end of the call; an `updates` parameter is given a meaningful value that will be changed to a new meaningful value as specified in the `ensures` clause; a `replaces` parameter will be passed an arbitrary value and be changed to a meaningful value by the end of the call.

`requires` and `ensures` clauses are mathematical assertions and their variables refer to the mathematical models of the parameters. So, while `Push` operates on physical `Stacks`, `S` in its `ensures` clause refers to the mathematical representation of the passed stack—a string of `Entrys`. Because mathematical variables can have only one, unchanging value, we use the pound sign to indicate the value at the beginning of the call. Thus, `#S` refers to the value of the `Stack` at the beginning of the call and `S` refers to its value at the end. Inside a `requires` clause there is no conception of the the values of parameters at the end of the call and so pounds are not used and all variables refer to the values of parameters at the beginning of the call.

Angled braces and the `o` operator come from `String_Theory` and represent the singleton-string constructor and the concatenation operator respectively.

Once a concept has been described, an implementation can be provided in a *realization*. Listing 7 provides a realization of a `Stack` using an array. Note that while we provide some syntactic sugar for arrays, a pre-processing step translates all array manipulation into interactions with a normal component and thus all reasoning is accomplished via a specification that models arrays as functions from integers to entries rather than hard-coded or specialized array reasoning. This contrasts with every other practical programming verification system we are aware of that supports arrays and provides an example of how often-primitive structures can be modelled normally within

the framework of RESOLVE's flexible mathematical subsystem[7].

Listing 7: An Array Realization

```
Realization Array_Realiz for Stack_Template;

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

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

    Procedure Pop(replaces R: Entry; updates S: Stack);
        R :=: S.Contents[S.Top];
        S.Top := S.Top - 1;
    end Pop;

    (* Other operations omitted for brevity *)
end;
```

We represent our Stack programmatically as a *record* (similar to a *struct* in C) containing an array of contents and a top index. A `convention` represents an *invariant* that must hold after initialization, before finalization, and before and after each method. In this case, the top must be at a valid index (save index 0, where it is permitted to reside if the Stack is empty.)

A `correspondence` provides a mapping between the physical `Stack` and its mathematical conceptualization, in this case using a mathematical definition `Concatenation`, which is to string concatenation what big Σ is to integer addition, to build a string of all its elements.

A procedure is provided for accomplishing both `Push()` and `Pop()`, taking advantage of standard integer and array operations.

[7]In addition to this array-based implementation, we are experimenting with a linked-structure mathematical component in order to provide, in this case, a linked-list based implementation. The details of this exploration can be found in [26].

Using the specifications of these operations, the RESOLVE compiler is able to generate VCs establishing the correctness of this `Stack` implementation. The details of this generation process is the topic of [16]. As an example, this is the VC establishing the convention at the end of a call to `Push()`:

```
Goal:
(0 <= (S.Top + 1)) and ((S.Top + 1) <= Max_Depth)

Given:
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (1  <= (Max_Depth  + 1))
5: (min_int <= Max_Depth) and (Max_Depth <= max_int)
6: (min_int <= 1) and (1 <= max_int)
7: (1  <= Max_Depth)
8: (min_int <= Max_Depth) and (Max_Depth <= max_int)
9: (0 <= S.Top) and (S.Top <= Max_Depth)
10: (|Reverse(Concatenation i:Integer
        where (1 <= i) and (i <= S.Top), <S.Contents(i)>)|  + 1) <= Max_Depth)
11: E' = S.Contents((S.Top  + 1))
12: S'.Contents = lambda j: Z ({{E if j = (S.Top  + 1)
S.Contents(j) otherwise
}})
```

Note that in Given #10, the length of the result of the concatenation is clearly equal to `S.Top`, thus we know that `S.Top + 1` is less than `Max_Depth`, satisfying half the goal. Then, by Given #9, we see that `0 <= S.Top`, so clearly `0 <= (S.Top + 1)`, satisfying the other half.

Once generated, VCs are passed to RESOLVE's integrated minimalist prover. This prover was built as an extensible prover platform for verification experimentation[39] and is discussed more thoroughly in Section 4.2.

Concepts may further define *extension operations*, which are secondary operations describing useful operations that not all realizations will be able to implement practically or efficiently. As an example, `Stack_Template` has an extension operation called `Flip`:

```
Enhancement Flip_Capability for Stack_Template;
        Operation Flip(updates S: Stack);
                ensures S = Reverse(#S);
end;
```

Just as with a concept, enhancements may have multiple associated realizations, for which the

VC-generation and proving process is the same. Consider the realization of `Flipping_Capability` given in Listing 8.

Listing 8: A Realization of Flip

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

        While (Depth(S) /= 0)
            changing S, S_Flipped, Next_Entry;
            maintaining #S = Reverse(S_Flipped) o S;
            decreasing |S|;
        do
            Pop(Next_Entry, S);
            Push(Next_Entry, S_Flipped);
        end;

        S_Flipped :=: S;
    end;
end;
```

This flipping procedure establishes a new `Stack`, `S_Flipped`, then iteratively pops each element off the original stack and into the new one. Having done so, the `:=:` operator swaps the value of `S_Flipped` with `S`. Using swapping as its basic data movement operator enables RESOLVE to minimize undesirable aliasing[15].

Notice that the while loop requires a number of annotations. The *changing* clause establishes a frame property: only those values explicitly listed may change. The *maintaining* clause establishes a loop invariant expressing the logic of the loop. Finally, the *decreasing* clause establishes a termination metric to allow us to guarantee termination.

Such annotations are standard operating procedure for verification systems and can be seen across the board at, for example, the first VSTTE verification competition[24]. We note, however, that some work has been done on automatically discovering, e.g., loop invariants on the programmer's behalf[12]. RESOLVE's explicit invariants do not preclude such a method from being employed to fill them in.

Using RESOLVE's integrated minimalist prover, a component of this research, we are able to fully and mechanically verify this procedure, as will be discussed further in Section 4.

4 Research

In this document, we have proposed to design a new mathematical and specification subsystem for a mechanical verifier, experiment with the capabilities of a minimalist prover, and apply these to determine the efficacy of specification and mathematical engineering techniques over a library of verified components.

The design of this system will be based on our experience writing and verifying reusable components using the existing RESOLVE verifying compiler, as well as applying lessons from our previous research.

Work on the research proposed in this document will proceed along the path outlined in the introduction, addressing the three points of the problem statement directly in order to test our thesis.

4.1 Extensible, Flexible Mathematics for Specification

Because every component that is to be verified must be modeled mathematically, the expressiveness of the mathematical system has a direct impact on the facility of the modeling process. By extension, this impacts the facility of the specification of operations as well, as they must necessarily operate on these mathematical models. In order to permit components to be modeled and reasoned about at the best level of mathematical abstraction, rather than simply the most convenient, a verification system must permit an extensible mathematical language. In order to promote the amortization of verification effort over time, the mathematical language must be flexible enough to support patterns of reuse.

Most practical systems do not meet these goals, as their mathematics are limited or built in. Many pure systems meet them in the mathematical realm, but fail to support the levels of integration with the programmatic realm required to reap the anticipated benefits. We propose to address this by combining an extensible, flexible mathematical system like those found in pure languages with the modular, reusable techniques of a modern object-based language like those expected for practical systems.

4.1.1 Work Completed

As we have already published in [7], genericity and modularity are crucial attributes of a useful verified component (in that paper, a Map data structure) and systems with inflexible mathematics often sacrifice these attributes by focussing on verifiability alone, yielding verified, but not terribly

useful, components. The mathematical and specification subsystems of our current RESOLVE prototype already contain many features that exemplify the sort of mathematics for verification that we propose. However, further work is needed to make them extensible and intuitive and to increase the robustness of the implementation. After much experimentation, described in the various *Work Completed* sections, we have identified the current type system as a major weakness and barrier to extensible, flexible design. In conjunction with mathematicians here at Clemson and at Ohio State, we have designed a new type system that will be more flexible, more closely match a modern mathematician's conception of the mathematical universe, and, at the same time, be easier to implement because of its regular and reflective nature.

Our new design exploits four key design principles to permit an extensible, flexible system:

4.1.1.1 Higher-Order Logic. If our verified components are to be worth the time we spend verifying them, they must be suitably generic to ensure broad reuse. Many reuse patterns found in modern programming languages are difficult or impossible to specify or verify using the first-order logic dictated by most practical verification systems and automated provers. In particular, they make it difficult to apply the lessons of programming to the mathematical world.

Consider, for example the $foldr$ function ubiquitous in functional languages. $foldr$ takes as its parameters a starting value of type γ, a function of type $(\gamma * \delta) \rightarrow \gamma$, and a list of elements of type δ. Starting with the starting value and the first element of the list, the function is applied to yield a new value of type γ before repeating the procedure with the resultant value and the next element of the list. The result of the final function application is returned. A summing function for lists of integers could thus be defined as:

$sum(zs) = foldr(0, +, zs)$

The broad applicability of such a function for specification should be obvious. However, even simple theorems describing the mathematical properties of this function run afoul of the first-order restriction that functions may not be quantified over. For example, Theorem 1 states that $foldr$ applied with an initial value to an empty list simply returns the initial value:

Theorem 1. $\forall f : (\gamma * \delta) \rightarrow \gamma, \forall ds : List(\delta), (|ds| = 0) \Rightarrow (foldr(i : \gamma, f, ds) = i)$

Creating definitions that operate on functions is similarly curtailed, preventing development of reusable theories of, for example, transitive functions.

Because our minimalist prover leaves functions and definitions *uninterpretted*, quantifying over them becomes straightforward. A function or definition is uninterpretted when we do not expand it

to consider its definition—i.e., the mathematical subsystem looks at a function or definition variable as a black box, treating it no differently from an ordinary variable. The tradeoffs inherent in such a design decision are discussed more completely in [42].

4.1.1.2 First-class Types. First-class types are a feature of several mathematical systems and a handful of experimental programming languages, but were not a part of the original RESOLVE design. The current prototype treats types as a special case, which makes them difficult and inconsistent to manipulate, limiting the facility of mathematical extension.

The new design incorporates first-class types that are treated as normal mathematical values. They can be manipulated, passed as parameters, returned as the result of a relation, and quantified over. This provides both a great deal of flexibilty, as well as a straightforward mechanic for specifying certain generic programming paradigms (most obviously: parameterized type variables.)

For example, the following line would introduce a particular integer called **1**:

```
Definition 1 : Z = succ(0);
```

In exactly the same way, a new type called **N** could be defined:

```
Definition N : Power(Z) = {n : Z | n >= 0};
```

The symbol table maintains information about the kinds of elements that make up any existing class and can thus infer when the symbol introduced by a definition can safely be used as a type. Thus this is a valid sequence of definitions:

```
Definition N : Power(Z) = {n : Z | n >= 0};
Definition NAcceptor(m : N) : B = true;
```

But this one is not:

```
Definition 1 : Z = succ(0);
Definition OneAcceptor(o : 1) : B = true;
```

Note that the "type" `Power(Z)` is not actually a type, but rather a function that takes one type and returns another–itself an example of first-class types in action.

Type schemas and dependent types, which define generalized types parameterized by other values, also take advantage of the first-class nature of types and can be defined as normal relations that return a type, rather than using a special syntax. This requires values that are, in some sense, *of type Type*. We call this type **MType** as an abbreviation for *Math Type*[8].

[8] `MType` is comparable to * in Haskell, where * is the kind of a type.

So, as a complete example, consider the String type schema, which represents a finite sequence of elements of a certain type, presupposing the existence of a type **UntypedStr**, which contains all finite sequences of elements of possibly heterogenous type, and a function called `Contains_Only_Elements_Of_Type()` which returns `true` if and only if a given UntypedStr contains only element of the given type:

```
Definition Str(T : MType) : MType =
        {S : UntypedStr | Contains_Only_Elements_Of_Type(S, T)};
```

Because they are treated like any other values, such first-class types can be reasoned about by all the same mechanisms already in place for other mathematical values.

4.1.1.3 Mechanisms for Static Reasoning.

A practical problem with first-class types is the ability to specify types and type-matching situations that are undecidable. This is an issue common to all systems that permit dependent types–which a system of truly first-class types must necessarily do.

For example we could imagine two types defined by arbitrary predicates:

```
Definition T1 : MType = {E : MType | P(E)};
Definition T2 : MType = {E : MType | Q(E)};
```

Determining whether or not an object modeled by `T1` can be passed where one modeled by `T2` is expected is equivalent to asking if $\forall e$:`MType`$, Q(e) \rightarrow P(e)$, which is, of course, undecidable.

Some verification systems (e.g., Coq) take advantage of this very property as an aid to verification, reducing all programs to reasoning about complex types and casting verification as a type-matching problems. However, because our goal is to permit the use of flexible mathematics on top of a foundation that takes advantage of standard programming models, we would ideally like to provide the usual static type-safety expected by object-oriented programmers.

To this end, we will largely refuse to attempt matching arbitrary mathematical types, relying instead on the programmer or mathematician to provide an explicit mapping in cases where the reasoning is complicated.

In some cases, relationships can be easily inferred. For example, the case of simple sub-types:

```
Definition Z : MType = ...;
Definition N : Power(Z) = ...;
```

Here we can easily note in the symbol table that an `N` would be acceptable wherever a `Z` is required and permit such a shift in conception-of-type in certain well-defined cases.

However, hard-coding such relationships does not provide a general mechanism for reasoning about type relationships and with first-class types, we may quickly arrive in situations where we'd expect the ability to reason about complex type relationships without requiring explicit assertions from the programmer or mathematician. Consider this application of `Strs`:

```
Definition Average(S : Str(Z)) : Z = ...;
Definition SomeNs : Str(N) = <1, 10, 3, 19>;
Definition AverageOfSomeNs : Z = Average(SomeNs);
```

In an ideal world we would expect this to type-check. But it is not the case that for two general type constructors, if the parameters of the one are subtypes of the parameters for the other, then the results are themselves subtypes. Consider this (somewhat contrived) complement string type:

```
Definition ComplementStr(T : MType) : MType =
        {S : UntypedStr | For all i, where 0 <= i < |S|,
                Element_At(S, i) not_in T);
```

This is the type of all strings containing possibly-heterogenously-typed elements where no element is of type `T`. Clearly, *this* set of definitions should *not* typecheck:

```
Definition Average(S : ComplementStr(Z)) : Z = ...;
Definition NotSomeNs : ComplementStr(N) = <-1, -10, -3, -19>;
Definition AverageOfNotSomeNs : Z = Average(SomeNs);
```

However, the same thing that got us into this mess—first-class types—provides a road out. Because types are normal mathematical values and we already have a mechanism for asserting theorems about mathematical values, we can use that existing mechanism to provide information about type relationships.

Because such theorems must take a specific form if they're to be understood by the static type-checker, we add some special syntax, calling them `Type Theorems` instead of ordinary `Theorems`. This flags these theorems for the type checker and ensures that we can raise an error if they do not have the proper form. So, for example, here is a type theorem stating that, among other things, `Str(N)s` should be acceptable where `Str(Z)s` are required:

```
Type Theorem Str_Subtype_Theorem:
        For all T1 : MType, For all T2 : Power(T1),
        For all S : Str(T2),
                S : Str(T1);
```

As with any other theorem in the RESOLVE system, this one would require a proof to establish its correctness and maintain soundness. We assume the presence of a proof-checking subsystem and leave such proofs outside the scope of this research.

Sometimes, more complex relationships are required. For example, in some circumstances, providing a `Z` where an `N` is expected should be fine. We can use the same mechanism to provide for this case:

```
Type Theorem Z_Superset_of_N:
        For all m : Z, (m >= 0) implies m : N;
```

This permits us, under a limited set of circumstances, to provisionally accept a "sane" type reassignment, while raising an appropriate proof obligation that the value in question is non-negative. This is similar to Java, where sane typecasts (i.e., from a `List` to an `ArrayList`) are permitted, but cause a run-time check to be compiled into the code. Here, however, we pay the penalty only once—during verification-time—rather than with each run of the program.

This design splits the difference between a rich, expressive type system and the straightforward static typing programmers have come to expect. Simple cases can be covered without thought on the part of the programmer, while complex, undecidable cases are permitted by explicitely deferring to the prover.

4.1.1.4 Rich Theory of Mathematical Types.

When expressing types in a mathematical system, it is natural to look to the Set abstraction. However, in doing so one must be careful not to inherit any of the many paradoxes and inconsistencies that have dogged the development of theories of sets. Many schemes exist to correct the deficiencies in naive set theory, with most pure systems using theories based on inductive structures and intuitionistic logics, as these are often well suited to computation. However, these are quite disjoint from a modern mathematician's conception of the universe. We choose instead Morse-Kelley Set Theory to be our basis, augmented with higher-order definitions.

First, note that RESOLVE values come in two flavors—mathematical values like the empty set, ϕ, and programming values like the empty array. We may trivially show that there are a finite collection of progamming values (after all, there is only a finite number of machine states,) and thus, without loss of generality, we may confine our thinking to the mathematical values, trusting that we can, if nothing else, provide an explicit mapping between the two later. We thus dispense with distinguishing between them for the moment, using "value" to mean "mathematical value".

Morse-Kelley Set Theory (MK) defeats Russel's Paradox in the same way as, for example, von Neumann-Bernays-Gödel Set Theory, by imagining that sets are each members of a larger meta-set called the *classes* and admitting the existence of *proper classes*–i.e., set-like objects that are not sets. We then restrict sets to containing only other sets, while proper classes may contain sets (but

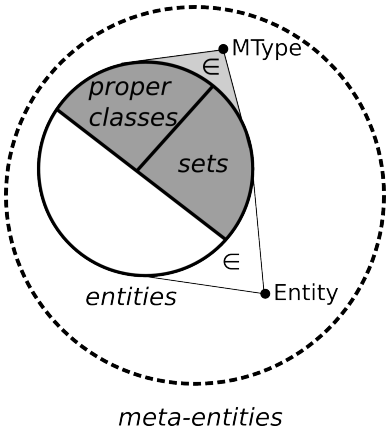

Figure 5: A High-level View of the RESOLVE Mathematical Universe

nothing may contain a proper class.) Under this light, we may rephrase the classic example of Russel's Paradox into "the class of sets that don't contain themselves", and view its contradiction not as an inconsistency, but rather as a proof that the class in question is proper.

MK permits us all the familiar and natural set constructors, restricted only by the necessity to reason carefully about what classes might be proper. This is ideal, since mathematicians need not be limited by glaring restrictions to class construction that exist only to eliminate corner-case inconsistencies. While, in general, only a formal proof can establish a given class as a set, in most cases we can infer it easily as most constructors are closed under the sets—e.g., the union of two sets is always a set.

We will imagine the universe of MK classes to be our universe of types—that is, `MType` from Section 4.1.1.2. Because describing the class that contains all classes would once again introduce Russel's Paradox, we imagine `MType` is a *meta-class* that exists "above" the classes, just as the classes exist "above" the sets.

We will imagine that the union of `MType` with all those things that can be elements of some class to be the universe of RESOLVE values. We will call this meta-class `Entity`. Note that while we may permit `MType` to be used as part of a type signature, we must be careful not to let it be passed as a parameter—it is not a value. Though we might permit it to be used informally by imagining that when used as a value it indicates some broad subset of types.

With all this in mind, the RESOLVE mathematical universe can be visualized as in Figure 5.

4.1.2 Work To Be Completed

The design of the new mathematical and specification subsystem has already been completed at a high level, though work remains to formalize it more completely. In particular, while we believe that the addition of higher-order definitions should not pose any soundness problems since uninterpreted definitions eschew many of the problems related with higher-order theories, this requires more research and theory development to establish. This said, formally establishing the soundness of the new theory is outside the scope of this research—we seek to demonstrate only the usefulness of this level of abstraction; future work may be required to refine the mathematical details.

Uninterpretted, higher-order definitions have been implemented and well-tested. The framework for the new type system has been laid and is awaiting the implementation of type theorems before it can be fully implemented and tested. Once in place, first-class types, the MK set theory, and type theorems will be tested together and refined against a set of target mathematical theories that will be designed to start simple and graduate eventually to the level of complication of generic theories of trees. While the simpler of these theories have already been developed, the higher-level ones remain.

4.2 Minimalist Prover

At the core of any mechanical verification system is an automated theorem prover responsible for discharging VCs. By definition, it is the last word on whether or not a particular technique is yielding more or less easily-proved VCs. As a result of this, most practical systems have focussed on incorporating the latest and greatest provers into their retinue to piggyback on the breakthroughs at the bleeding edge of proving and artificial intelligence and thus increase provability.

While we are happy to support the latest and greatest suite of provers, we hypothesize that in many cases *flexibility* may trump raw performance with respect to mechanically verifying well-engineered software by encouraging good specification and mathematical engineering that captures the programmer's intuition rather than compromising to work within the framework of a target prover.

In order to experiment with this hypothesis and identify those prover capabilities and performance tunings required to verify well-engineered software, we set out to create a *minimalist automated prover*, starting with only the bare essential capabilities and expanding only when a significant number of VCs appeared that could not be addressed with the prover as it stood. The result of this effort was RESOLVE's integrated rewrite prover. As we've refined our design, it has become a platform for prover experimentation within the group and we intend to use it as the yardstick

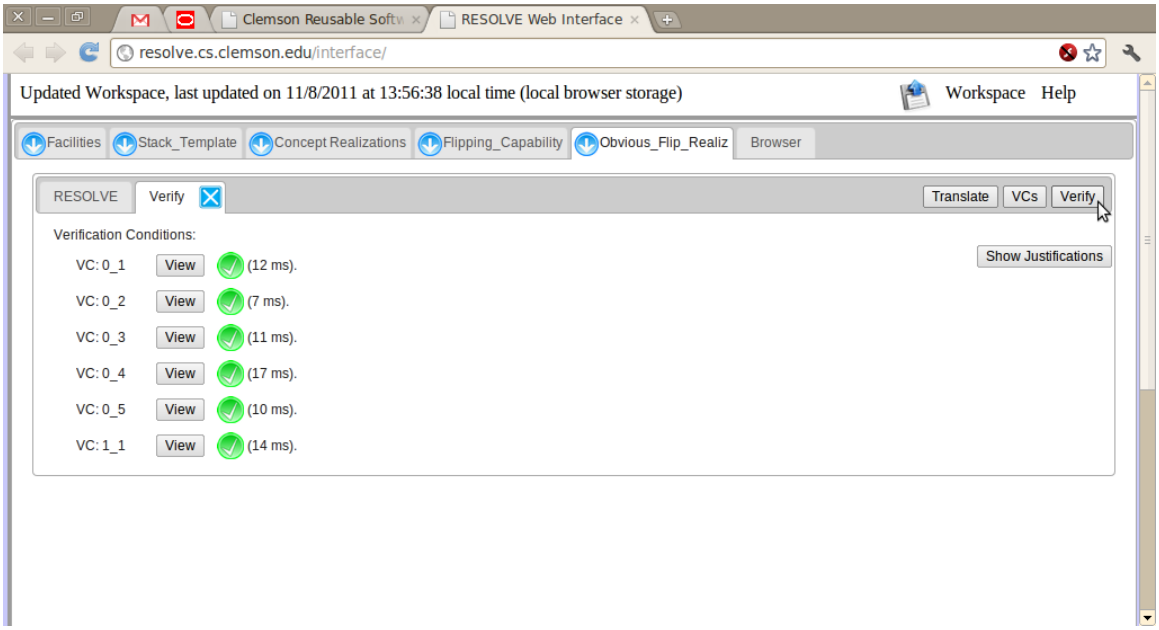

Figure 6: Our Minimalist Prover at Work

against which to measure our success using our new mathematical system to verify components.

4.2.1 Work Completed

In [39], we present our architecture for an extensible platform for minimalistic prover experimentation. The implementation of this prover is in a working form and has been used for several years as part of the RESOLVE toolchain, incuding as part of our education effort centered around the RESOLVE integrated web development environment[8]. This prover has been used in support of a number of our verification publications, including [36] and [38]. We see the results of a successful verification from the web interface in Figure 6. This is a verification of a realization of the `Flip()` operation introduced in Section 3. Enhancement realizations often involve a small number of mathematical types (in this case, almost entirely Strings) and are therefore more straightforward to verify. Data structure realizations, on the other hand, often have to map from one representation to another, introducing many related types. As a result, while we can prove them by hand, the mechanical verification of data structures is consigned to Work to be Completed.

As part of our experimentation to-date, we have utilized the extensible plug-in architecture of the prover to add a number of capabilities. These have included heuristics to make intelligent theorem choices based on the VC at hand, a graphical front-end that permits the user to guide the prover, and a number of preprocessing options to simplify and normalize incoming VCs. In [38] we perform a comparison of proof complexity (in terms of number of proof steps, including backtracks, taken

before reaching a complete proof) based on a simple component using a number of different feature configurations. This exploration was very illuminating as to which features gave good results.

4.2.2 Work To Be Completed

With the harness already in working order, the most important addition is that of facilities for mechanically characterizing VCs according to a number of different mechanisms. Our current depth-first search is more efficient than a breadth-first search, but means that we do not necessarily find the *shortest* proof. Work is needed to permit a breadth first option when we're willing to trade some efficiency for a more precise metrics. In addition, we imagine some metrics will be related to the theorems used and the current prover output takes the form of a plain-text ASCII dump, which is not ideal for processing. A better solution must be engineered so that proofs can be compared and processed mechanically.

Additionally, we would like to expand our efforts to collect data on different prover feature configurations by applying this analysis to a broader range of components. As mentioned in the Work Completed section, we are eager to expand mechanical verification from enhancement realizations to data structure realizations. This process will include the explorations of features that have yet to be identified and may be included upon discovering their usefulness for practical VCs. As an example of a capability we have identified but have yet to implement, we consider a snippet from a realization of a `Merge_Sort()` enhancement for queues. As part of a loop invariant we state:

```
Is_Permutation(#Q o #R, Merger o <Q_Min> o Q o R)
```

Clearly, in the context of a permutation, the concatenation operator is commutative. Unfortunately, any straightforward commutativity theorem requires us to play a complicated shell game to rearrange arbitrary chains of concatenation. We could use a torrent of theorems to try and cover all cases, but we suspect that at that point, we have followed minimalism to a ridiculous extreme. We would like, instead, to permit such context-sensitive properties to be stated in a general way, after which a better internal prover representation than a tree could be chosen to better reflect the properties of the conjunct.

We would also like to experiment with minimalistic proving algorithms. For example, in addition to the back-tracking, depth- or breadth-first search algorithm we establish in [39], we could imagine that we instead start with our set of antecedents and our set of consequents, then transitively build up a larger and larger set of known antecedents as we simultaneously build up a set of possible justifications for each conjunct of the consequents, dispatch a conjunct whenever its justification set intersects the antecedent set.

Finally, the prover will need to be adapted to work with the new mathematical type-system, though this modification should be straight-forward since complex reasoning about types is unnecessary in the prover, which need only be able to answer the questions "What is the type of this expression?" and "Does an expression of this type bind to this free variable?" The former question ought to be answered before the prover is ever invoked, while the latter should be a straightforward library call to the symbol table.

4.3 Specification and Mathematical Engineering

At its heart the question of specification engineering is one of how equivalent logical expressions affect practical attempts at verification. As a small example, we may express that a `Stack`, modelled as a string of `Entrys` as in Section 3, is empty via $S = \lambda$ or $|S| = 0$. Indeed, there are an endless number of ways to express this single idea. In cases where we are largely thinking in terms of contents of the stack, the former may be preferable. In cases where we are thinking in terms of stack length, the latter may be preferable. Many current verification systems encourage programmers to state facts many different ways to maximize the liklihood of verification. We see this as a barrier to usability and would instead like to suggest best practices so that a fact can be stated the *most appropriate way*. In the future, systems might apply an intelligent heuristic converting equivalent expressions as most convenient.

A more complex dimension is the way in which parameter transformations are formed. In *explicit* style, the final value of each parameter is defined as a relation of inputs, with the variable on the left hand side of an equality and the the function on the right. In *implicit* style, a relation relates the final value and inputs. As an example, assuming an operation `Substring()` that takes a string and a zero-indexed starting and ending index and returns the substring starting at the first index and end going up to but not including the ending index, we might specify `Pop()` on a `Stack` this way in explicit style:

```
Operation Pop(replaces E : Entry, updates S : Stack);
    requires S /= empty_string;
    ensures S = Substring(#S, 1, |S|);
```

Alternatively, in implicit style:

```
Operation Pop(replaces E : Entry, updates S : Stack);
    requires S /= empty_string;
    ensures #S = <E> o S;
```

4.3.1 Work Completed

We have already begun identifying and experimenting with different dimensions of specification and mathematical flexibility. In [23], we explore the complexity of VCs arising from well-engineered components, reaching the conclusion that the vast majority of them are simple bookkeeping rather deep mathematical results. We continued this work in [40], classifying VCs resulting from multiple different versions of a specification for the same programmatic component. We also explored some verifiability metrics to give a more accurate picture of proof difficulty than merely timing the verification attempt.

The metrics introduced in the latter paper were specific to rewrite-style provers (rather than those based on SAT solvers, for which many of these concepts do not apply). Among them were shortest proof length, i.e., the number of theorems that had to be applied before a successful proof was achieved; and theorem uniqueness, a measure of how specialized or general the required theorems were.

As an example of some recent success with specification engineering, we consider an enhancement operation to sort a queue. A naive specification involves universal quantifiers and might appear as in Listing 9.

Listing 9: A Naive Sort Specification

```
Enhancement Naive_Sorting_Capability(Definition LEQV(x, y : Entry) : B) for
        Queue_Template;
    requires for all a, b, c : Entry,
        ((LEQV(a, b) and LEQV(b, c)) implies (LEQV(a, c))) and
        ((LEQV(a, b) or LEQV(a, b));

    Operation Sort(updates Q : Queue);
        ensures for all i : Z such that i < |Q|,
                LEQV(Element_At(Q, i), Element_At(Q, i + 1)) and
            for all E : Entry,
                |Concatenation j: Integer
                    where 1 <= j <= |Q| and Element_At(Q, j) = E,
                    <S.Contents(i)>| =
                |Concatenation j: Integer
                    where 1 <= i <= |#Q| and Element_At(#Q, j) = E,
                    <S.Contents(i)>|;
end;
```

This enhancement is parameterized by a defintion `LEQV`, which is constrained to have the proper-

ties of a total preordering. The ensures clause of the sorting operation establishes the two important behavioral qualities of a sort: 1) that the elements are in order with respect to the provided operation and 2) that the resulting queue is a permutation of the original. Note that we must be careful to deal with repeated elements.

Complex as this spec looks, it is exactly of the kind seen almost universally in the verification literature and, to our knowledge, no sorting implementation of general entry types and using a general comparison function is mechanically verifiable. This is in part because of the difficulty proving mathematical statements involving quantified expressions, as discussed in Section 1.4.1. Because of our hypothesis that eliminating quantifiers would aid in simplifying verification we experimented with rewriting the specification to take advantage of uninterpretted definitions, as in Listing 10.

Listing 10: A Sort Specification with Uninterpretted Definitions

```
Enhancement Sorting_Capability(Definition LEQV(x, y : Entry) : B) for
                Queue_Template;
        uses Ordering_Theory;
        requires Is_Total_Preordering(LEQV);

        Operation Sort(updates Q : Queue);
                ensures Is_Conformal_With(Q, LEQV) and
                        Is_Permutation(Q, #Q);
end;
```

Notice that we include a theory of string orderings, `Ordering_Theory`, which contains a host of useful definitions and theorems. We now state an equivalent specification that is nonetheless much more succinct and easy to read from the human perspective. The only definition requiring any explanation at all is `Is_Conformal_With()`, which is a higher order definition that takes a string and a total preordering and returns *true* **iff** the elements in that string are ordered according to the ordering function. From the computational side, we exploit the uninterpretted nature of the definitions (i.e., they function only as a black box) to simplify verification. We do this by providing a set of theorems in `Ordering_Theory` for manipulating ordering definition; as, for example, Theorem 2.

Theorem 2. $\forall T : \text{MType}, \forall S : \text{Str}(T), \forall f : (T * T) \to \mathbb{B}, \forall E : T,$
$\text{Is\_Total\_Preordering}(f) \wedge \text{Is\_Conformal\_With}(S, f) \wedge f(\text{Element\_At}(S, |S| - 1), E) \Rightarrow$
$\text{Is\_Conformal\_With}(S \circ <E>, f)$

Theorems like these may represent deep mathematical results and thus are not automatically

provable in general. In these cases, proofs may be supplied manually and checked mechanically[41]. Note that, unlike proofs of specific programs, proofs of general theorems represents reusable effort. Such a mechanical proof-checker is assumed to be present for the purposes of this research and is not included in the scope.

Such a general theory may be called upon repeatedly to simply verification. In this case, RESOLVE is able to fully and automatically verify an implementation of selection sort against this specification, a feat that, to our knowledge, no other verification system has achieved.

4.3.2 Work To Be Completed

Conceptual model is only one dimension of mathematical flexibility. As already mentioned, the mapping of inputs to outputs and choice of equivalent value expressions are others. No doubt there are more yet to be clearly identified.

We would like to expand on the experimentation we've already done by creating addional components of a variety of kinds and then specifying them in ways that span the dimensions we've identified. One source for such components is our previously published set of verification benchmarks[44], which include a number of challenges ranging from toy examples to data structures to complex manipulations.

Once created, we can set our prover to verifying these components and collecting data in an attempt to identify patterns in the ease of verification (or lack there of) under different kinds of specification and mathematical expression and their various interactions.

5 Evaluation

In some sense, this research is itself an evaluation suite—a prover is an apparatus for collecting data about provability and a library of components specified in different ways along well-defined axes of variability is a set of good test cases. However, in order to get good data from our test suite, we must make sure it is an effective one.

5.1 Extensible, Flexible Mathematics for Specification

Since the motivation for the design of the mathematical and specification subsystem is specifically to enable the development of modular components specified using techniques from a flexible set of mathematical options, the creation of the component library in the third part of the problem statement will serve to evaluate the system. It will be evaluated by the handful of programming and mathematical professionals here at Clemson, at Ohio State, and elsewhere, who will contribute to this library. The system will be a success if components can be easily created with the modularity and genericity profiles we desire and require improvement if they cannot. Additionally, a number of benchmarks discussed in the related works section (Section 2.3) will be useful in demonstrating that we are able to specify components in reasonable ways.

5.2 Minimalist Prover

Our minimalist prover must be evaluated to be certain it is a prover worth exploring: after all, if it cannot prove any practical components, it can't collect useful data for us. Between the RESOLVE groups at OSU and here at Clemson, we already have a broad library of specified components on which to experiment and, as stated in Section 4.2, the prover has already been used to excellent effect in a number of deployments. In addition, the library we develop for the third part of our problem statement will provide an ideal set of components against which to evaluate if our prover can be reasonably called upon to verify practical components. As we discussed in the research section, a constellation of components that cannot be mechanically verified by our prover may indicate a missing feature to be implemented.

The final set of features for our prover will not be decided arbitrarily, but rather via quantitative data on the gains over a number of verifiability metrics (discussed further in the next section) under different feature configurations. This will permit us to experimentally arrive at a prover that strikes a good balance between simplicity and power.

In addition, in evaluating the components in the library, we will expect the prover to be able

to provide data on a number of proof metrics, which will require further prover developments. As we have many existing components, both mechanically verifiable and not, we will have a strong indicator of how accurate and useful our metrics are before ever applying them to the unknowns of our new library.

5.3 Specification and Mathematical Engineering

This is the meat of our evaluation, which the other two subproblems support. We would like to be able to draw conclusions about techniques for mathematical and specification development. With a diverse library of components developed, each specified in multiple different ways along well-defined axes of variability, we will be able to evaluate these techniques by applying our minimalist prover to gather verifiability metrics. In addition to time-to-verify, which is the standard metric used in the literature, in [40] we discuss two other potential metrics: required proof length and theorem specificity. Other metrics may present themselves as we explore and be folded into our prover[9]. These metrics will be collected for each component (including the most important metric of all—can it be mechanically verified?), then analyzed to draw some conclusion about the nature of specification.

[9] In that same paper we noted the usefulness of metrics that could be obtained even for proofs that could not be completed. We are on the lookout for these.

6 Conclusion

In conclusion, this research proposes to address a constellation of open problems in the design of a verification system and the engineering of mathematics and specifications within such a system. A system including an extensible, modular mathematical language, well integrated with a practical, component-based programming language with a focus on modular components would be a novel entry into the realm of available systems and would enable us to test our hypothesis that well-engineered components, with respect not only to their implementation, but their specification as well, are a key to mechanically verifiable software. This hypothesis will be established in part by demonstrating that with the aid of such engineering, a minimalist automated theorem proving style is sufficient to dispatch proof obligations arising from routine software.

By evaluating our design and implementation as we go against a well-design library specifically created to reveal the facility with which different specification paradigms can be realized, as well as their contribution to easy-to-verify VCs, we believe we can create a body of data and observations that will benefit not only users of RESOLVE, but also a wide variety of existing verification systems, both intended for practical software and deep mathematical development.

Bibliography

[1] Mike Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Leino, Wolfram Schulte, and Herman Venter. The Spec# programming system: Challenges and directions. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-69149-5_16.

[2] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[3] J.L. Bentley. *Programming pearls*. ACM Press Series. Addison-Wesley, 2000.

[4] Joshua Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken, June 2006.

[5] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[6] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. Using first-order theorem provers in the jahob data structure verification system. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-69738-1_5.

[7] Derek Bronish and Hampton Smith. Robust, generic, modularly-verified map: a software verification challenge problem. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 27–30, New York, NY, USA, 2011. ACM.

[8] Chuck Cook. A web-integrated environment for component-based software reasoning. Masters thesis, Clemson University, School of Computing, September 2011.

[9] Leonardo de Moura and Nikolaj Bjrner. Z3: An efficient smt solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-78800-3_24.

[10] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[11] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[12] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27:99–123, February 2001.

[13] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[15] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Softw. Eng.*, 17:424–435, May 1991.

[16] Heather Harton. *Mechanical and Modular Verification Condition Generation for Object-Based Software*. Phd dissertation, Clemson University, School of Computing, December 2011.

[17] John Hatcliff, Gary T. Leavens, K. Rustan, M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. 2009.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[19] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50:63–69, January 2003.

[20] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: a powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] Matt Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Softw. Eng.*, 23:203–213, April 1997.

[22] James Cornelius King. *A program verifier*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970. AAI7018026.

[23] Jason Kirschenbaum, Bruce Adcock, Derek Bronish, Hampton Smith, Heather Harton, Murali Sitaraman, and Bruce Weide. Verifying component-based software: Deep mathematics or simple bookkeeping? In Stephen Edwards and Gregory Kulczycki, editors, *Formal Foundations of Reuse and Domain Engineering*, volume 5791 of *Lecture Notes in Computer Science*, pages 31–40. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-04211-9_4.

[24] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st verified software competition: experience report. In *Proceedings of the 17th international conference on Formal methods*, FM'11, pages 154–168, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[26] Greg Kulczycki, Hampton Smith, Heather Harton, Murali Sitaraman, William F. Ogden, and Joseph E. Hollingsworth. The location linking concept: A basis for verification of code using pointers. In *Proceedings of VSTTE 2012 (to appear)*, Berlin, Heidelberg, 2012. Springer-Verlag.

[27] Viktor Kuncak and Martin Rinard. An overview of the jahob analysis system: project goals and current status. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 285–285, Washington, DC, USA, 2006. IEEE Computer Society.

[28] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998. `http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html`.

[29] K. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 112–126. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15057-9_8.

[30] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.

[31] LogiCal Project. *The Coq proof assistant reference manual*, 2004. Version 8.0.

[32] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53:937–977, November 2006.

[33] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[34] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[35] Hossein Sheini and Karem Sakallah. A sat-based decision procedure for mixed logical/integer linear problems. In Roman Bartk and Michela Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3524 of *Lecture Notes in Computer Science*, pages 847–849. Springer Berlin / Heidelberg, 2005. 10.1007/11493853_24.

[36] Murali Sitaraman, Bruce Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, David Frazier, Harvey Friedman, Heather Harton, Wayne Heym, Jason Kirschenbaum, Joan Krone, Hampton Smith, and Bruce Weide. Building a push-button resolve verifier: Progress and challenges. *Formal Aspects of Computing*, 23:607–626, 2011. 10.1007/s00165-010-0154-3.

[37] Murali Sitaraman and Bruce Weide. Component-based software using RESOLVE. *SIGSOFT Softw. Eng. Notes*, 19(4):21–22, 1994.

[38] Hampton Smith. Experimentation with a minimalist prover. In *Proceedings of RESOLVE 2009 Workshop*, RESOLVE 2009, 2009.

[39] Hampton Smith. Anatomy of a platform for prover experimentation. Technical report, Clemson University, 2010.

[40] Hampton Smith. Impact of specification abstractions on client verification. In *Proceedings of SAVCBS 2010*, SAVCBS 2010, November 2010.

[41] Hampton Smith, Kim Roche, Murali Sitaraman, Joan Krone, and William F. Ogden. Integrating math units and proof checking for specification and verification. In *Proceedings of the Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, pages 59–66, 2008.

[42] Aditi Tagore, Diego Zaccai, and Bruce. W. Weide. To expand or not to expand: Automatically verifying software specified with complex mathematical definitions. Technical report, The Ohio State University, September 2011.

[43] Bruce W. Weide and Wayne D. Heym. Specification and verification with references. In *Proceedings of 2001 OOPSLA workshop on Specification and Verification of Component-Based Systems*, SAVCBS 2001, pages 50–59, 2001.

[44] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce M. Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, pages 84–98, 2008.

[45] Markus M. Wenzel and Technische Universitt Mnchen. Isabelle/isar - a versatile environment for human-readable formal proof documents. In *TPHOLS*, pages 167–184, 1999.

[46] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 349–361, New York, NY, USA, 2008. ACM.

[47] Karen Zee, Viktor Kuncak, and Martin C. Rinard. An integrated proof language for imperative programs. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 338–351, New York, NY, USA, 2009. ACM.