Integrating Math Units and Proof Checking for Specification and Verification

Hampton Smith
Kim Roche
Murali Sitaraman
Clemson University
School of Computing
Clemson, SC 29634
1-(864)-656-3444
{hamptos | kroche | murali}
@clemson.edu

Joan Krone
Denison University
Mathematics and Computer Science
Granville, OH 43023
1-(740)-587-6484
krone@denison.edu

William F. Ogden
Ohio State University
Computer and Information Science
Columbus, OH 43210
1-(614)-292-1517
ogden@cse.ohio-state.edu

ABSTRACT

A formal system for specification and verification of component-based software must allow extension of the mathematical units available for specification with new mathematical theories just as modern programming languages allow software developers to extend a core collection of data types with new ones by developing reusable software components. These extensions enrich the specification language and lead to simpler specifications. New theory development must also include suitable theorems so that it can be used to support automated proofs of verification conditions (VCs) for correctness arising from annotated implementations. We distinguish between straightforward proofs of VCs and the more nuanced proofs for the theorems in the mathematical units themselves, which often cannot be automated. We explain the need to separate the interface of a mathematical unit (précis) that will be used by software developers and automated provers, from the proof units that contain proofs of theorems. In addition, we describe a mathematician-friendly language for presenting proofs and a proof checker that we have developed to check these proofs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*class invariants*, *correctness proofs*, *formal methods*, and F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*invariants, mechanical verification, pre- and post- conditions*

General Terms

Design, Human Factors, Standardization, Languages, Theory, Verification.

Keywords

Specification, Verification, Proof Checking, Formal Methods

1. INTRODUCTION

The goal of automatically verifying software components with respect to a specification presents a fundamental dilemma. Requiring programmers to engage in a fine level of proof activity is unlikely to lead to wide-spread verification. On the other hand, the limitations of automated theorem proving often require substantial human intervention. Addressing this dilemma is the focus of this paper. We partition the problem of verification to distinguish the roles of software developers from mathematicians and automated provers from proof checkers.

Formal verification ultimately involves the insights of programmers (e.g., specifying invariants), the insights of mathematicians (e.g., discovering non-trivial theorems and furnishing proofs to support them), and the more straightforward task of proving verification conditions of implementation correctness based on these insights. Some verification conditions (VCs) correspond to checking the insights of programmers to eliminate unsoundness that may arise from poor programmer insights. In the scenario we envision, programmers would not be involved in any proving activity beyond documenting their insights. The proving activity would instead be partitioned into two sub-tasks:

1. Proofs of verification conditions to establish the correctness of code.
2. Proofs of supporting theorems from mathematics.

The former would be straightforward and only involve various simplifications so that an automated prover could discharge them without requiring human intervention. The latter would generally require proof steps from mathematicians.

Unlike the proof of VCs arising from code, where the goal is complete automation, the focus of extending the mathematical library is on enabling mathematicians to improve the expressiveness of the theories available to the specification and verification subsystems. It is therefore not necessary that these theorems be automatically verifiable. While a class of theorems can be discharged automatically by automated provers [5, 7, 11, 13], in general proofs of theorems require mathematical insights that cannot be discovered automatically. Obviously, limiting allowable theorems to those that can be automatically proved would in turn limit the class of programs that can be proved. To address this problem, some current systems (e.g., Isabelle) allow some theorems to be taken for granted without proofs, but clearly this can only be a temporary solution.

To address the sub-problem of proving non-trivial theorems, we have a developed a mathematician-friendly language for writing proofs and a proof checker for checking these proofs. We intend that these proofs will be written by mathematicians, not programmers.

The rest of this paper is organized as follows: In Section 2, we illustrate the need for a verification system to strike a balance between automated theorem proving and mechanically-checked (but user-provided) proofs. In Section 3, we discuss practical consequences of this balance and suggest ways in which the problem may be managed by applying traditional software design tactics such as modularity to the proof subsystem. We support these ideas with examples from the design of our own verification system, RESOLVE. In Section 4, we detail the workings of the proof language and its associated proof checker. In Section 5, we discuss related work and summarize our conclusions.

2. PROOFS OF VCS VS. THEOREMS

To illustrate the distinct issues in proving VCs arising from code and proving theorems in mathematics, we consider a component verification example. In particular, we consider an operation to reverse a given Stack object. A specification and an implementation of the operation are given below in RESOLVE, an integrated specification and programming language [12]. The issues discussed in this paper, however, are language independent.

Specification:

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

Code:

```
Realization Obvious_F_C_Realiz for Flipping_Capability
    of Stack_Template;

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

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

        S := : S_Flipped;
    end Flip;

end Obvious_F_C_Realiz;
```

The specification of Stack_Template on which the Flip enhancement (called an extension operation in other systems) is based imports the mathematical unit String_Theory and conceptualizes a Stack object as a mathematical string. The ensures clause, which defines the behavior of this operation, is used for verification and thus the variables in the clause stand for their mathematical values. In this case, #S refers to the mathematical string that represents the value of the stack S when this operation is called, while S refers to the mathematical string that represents the value of the stack S when this operation exits. Rev is a mathematical function that takes a string and returns it in reverse order.

The While statement in the code for Flip is annotated with three clauses that make the insights of the programmer regarding the correctness of the code explicit. For our purposes, it doesn't matter whether a programmer uses tools to identify and document such assertions (e.g., the work of [3] in identifying loop invariants automatically) or does so herself. The changing clause indicates those variables whose values are permitted to change inside the loop. Implicit is that any variable not mentioned will not change. The maintaining clause provides a loop invariant. The "o" in this line is intended to be read as $\circ$, the concatenation operator on mathematical strings. The decreasing clause documents the progress metric, i.e., the programmer's rationale for why the loop would terminate.

At the end of the loop, we use the :=: operator, which swaps the values of S and S_Flipped, thus transferring the stack containing the reversed contents to the parameter stack S. The motivation for using swapping and avoiding unnecessary aliasing is the topic of [4].

When the code for Flip is analyzed, the usual syntax-checking and type-checking is performed and, assuming it passes these checks, the code continues to a verifier, which generates VCs that must be proved in order for the code to be considered correct. The VCs generated using the RESOLVE VC generator [9] are shown in an appendix.

The verifier includes a flag to generate Isabelle-friendly assertions (not shown in this paper). Our experience in proving VCs automatically using Isabelle is the topic of [6]. Other example verification benchmarks are given in [15]. All the VCs for the present example can be discharged automatically by the Isabelle prover. Specifically, beyond documenting loop invariants and progress metrics, programmers are not involved at all in verification.

The VCs correspond to checking correctness of programmer-supplied invariants and progress metrics, checking the preconditions of called operations (e.g., Pop), and the postcondition of the operation that is being verified. We discuss the automated verification of one of the VCs to distinguish simplification from theorem proving activities. It is the third VC from the Appendix and it corresponds to the inductive step of establishing the correctness of the invariant. This VC (after removing assumptions that are not necessary) is shown below:

```
((|S| <= Max_Depth) and (S = (Rev(?S_Flipped) o ??S) and
(|??S| /= 0 and ??S = (<?Next_Entry> o ?S))))

=====================>

(Rev(?S_Flipped) o ??S) =
(Rev(<?Next_Entry> o ?S_Flipped) o ?S)
```

The VC is an implication. All variables in a VC are mathematical. For example, S is a String of Entries, not a Stack. In this VC, |S| indicates the length of the string S, <X> indicates the string with X as its sole element, and "o" is the concatenation operator on strings.

Variables in the VC prepended with a question mark are verifier-generated and simply represent the values of the variables at different points in the code. So, for instance, S represents the initial value of the stack S, ?S represents its value at the beginning of each loop, and ??S represents its value at the end of each loop.

Automated provers, such as Isabelle, would begin with a substitution in proving this VC:

$$(?S\_Flipped^{Rev} \circ ??S) = ((<?Next\_Entry> \circ ?S\_Flipped)^{Rev} \circ ?S)$$

given

$$(?S\_Flipped^{Rev} \circ \mathbf{(<?Next\_Entry> \circ ?S)}) = ((<?Next\_Entry> \circ ?S\_Flipped)^{Rev} \circ ?S)$$

by substitution

From here, the provers will rely on two important theorems from String_Theory to complete the proof:

Theorem 1:
$\forall\alpha$:String of E, $\forall x$:E, $(\alpha \circ <x>)^{Rev} = (<x> \circ \alpha^{Rev})$.

Theorem 2:
Is_Associative($\circ$)

Theorem 2 uses the higher order predicate Is_Associative that is defined in a separate math unit named Basic_Function_Properties. This unit defines several other related predicates and is reused by several mathematical units.

Clearly, proving the VC given these theorems is a qualitatively different activity from proving the theorems themselves. Given these theorems, proving the VC is a simple process of repeated substitution. The proofs for the theorems themselves are significantly more involved.

There are certainly automated theorem provers, particularly of the inductive variety, that could provide proofs of Theorems 1 and 2 on their own. ACL2 [1] is one such prover, though it is limited to first order assertions. However, there are many other theorems where automated provers would be unable to make the required logical leap. Indeed, we could imagine writing code that relies on Fermat's Last Theorem for its correctness.

Providing proofs for such theorems, in general, is a process for mathematicians. Programmers cannot be and should not be involved in proving theorems. The simpler task of applying these theorems as part of proving a VC is left for an automated prover. It is our hypothesis that String_Theory can, through careful experimentation and expansion, be fitted with sufficient theorems to make verifying the vast majority of programming concepts based on strings, such as Stacks, Queues, Lists, and others, a task of repeated substitution and thus within the capabilities of a modest automated prover.

Reusing mathematical notions such as strings to specify a wide variety of concepts makes it possible to eliminate the need for Larch-style theories [17] where the theory of Queues is separate from the theory of Stacks, with each different from the theory of Lists.

3. PRÉCIS AND PROOF UNITS

Any code verification system that is complete must provide a mechanism by which arbitrary new theorems can be added; any system that is to be sound must provide a mechanism for providing and checking proofs in support of those theorems. These results in mathematics are reusable in verifying a variety of software artifacts and need to be proved only once. Clearly, developing these proofs is beyond the expertise of typical programmers and should be left to trained mathematicians. This observation suggests a clear division of labor in which programmers are concerned only with immediate details and insights about proving their programs to be correct, whereas mathematicians are involved in proving more general theorems. Like programmers, mechanical provers of VCs need not be concerned with proofs of these theorems.

By linking all programming objects to the mathematical world, mathematical results become applicable in programming contexts. This means, however, that the automated prover is no longer the only entity that needs access to mathematical definitions and results. Software developers also need to be aware of them for use in specifying and verifying software components. However, in both cases they just need to know *what* the results are, but not how they were derived. Therefore, clean, modular, and component-based techniques derived from the world of programming must be applied to the mathematical world of proofs. This is the motivation for separating interfaces of math units (précis) from their corresponding proof units.

Since most readers are familiar with the preliminaries necessary to do proofs with number theory, we use the associativity result on the plus operator on natural numbers as our illustrative example (instead of the string $\circ$ operator). Many automated provers could, of course, dispatch such a theorem easily. We use it here as an accessible example for when automated proving is not possible.

It is easy to imagine the need for a theorem on the associativity of plus by conceiving of a simple piece of code such as I := (K + L) + M after which, for whatever reason, we need to confirm that I = K + (L + M). Clearly, the validity of this code relies on the associativity of plus on the natural numbers (in the same way the validity of the Flip code in the previous section relies on the associativity of ∘ on strings.) The précis for Natural_Number_Theory contains the definition of the set N, symbols, such as 0 and suc, and several theorems. We list below one definition and a theorem from this précis:

```
Précis Natural_Number_Theory;
    uses Basic_Function_Properties,
    Monogenerator_Theory...

    ...

    Inductive Definition on i : N of (a : N) + (b) : N is
        (i) a + 0 = a;
        (ii) a + suc(b) = suc(a + b);

    Theorem N1:  Is_Associative( + );

    ...
end Natural_Number_Theory;
```

A précis is an interface for theory users (both humans and mechanical provers). It provides a summary of the theorems in the theory—everything required to *use* the theorems without any of the details that support the theorems.

This arrangement has obvious analogues to both the header files of C and forms of documentation such as Javadocs. However, unlike C headers, which are primarily intended for use by the compilation system, or Javadocs, which are intended for human consumption, these précis are intended to aid both the verification system and human users. The verification system makes use of proven theorems to verify VCs, mathematicians use them to support new theorems with established ones, and programmers use them to to better tailor their specifications to the available body of mathematical truth. None of these entities need be concerned with the details of supporting proofs. The strict separation of précis from proof unit, enforced by the system, ensures that both documents are always available and synchronized.

The proof for N1 is found in the proof unit Natural_Number_Theory_Proofs. It relies on the definition of a natural number above and reads as follows:

```
Proof unit Natural_Number_Theory_Proofs for
        Natural_Number_Theory;
        Uses ...

Proof of Theorem N1:

Goal for all k, m, n: N, k + (m + n) = (k + m) + n;
Def S1: Powerset(N) =
    { n: N, for all k, m: N, k + (m + n) = (k + m) + n };
Goal S1 = N;
Goal 0 is_in S1;
Goal for all n: S1, suc(n) is_in S1;
Goal for all n: S1, if n is_in S1 then suc(n) is_in S1;
(Base_case) Goal 0 is_in S1;
Goal for all k, m: N, k + (m + 0) = (k + m) + 0;
Goal for all k, m: N, if k is_in N and m is_in N then
    k + (m + 0) = (k + m) + 0;
Supposition k, m: N;
    Goal k + (m + 0) = (k + m) + 0;
    k + (m + 0) = k + m
                                        by (i) of Definition +;
    k + m = (k + m) + 0
                                        by (i) of Definition  +;
Deduction if k is_in N and m is_in N then
    k + (m + 0) = (k + m) + 0;
[ZeroAssociativity] For all k: N, for all m: N,
    k + (m + 0) = (k + m) + 0
                                    by universal generalization;
[ZeroInS1] 0 is_in S1
                                        by ZeroAssociativity;
(Inductive_case) Goal for all n: N, suc(n) is_in S1;
Goal for all n: N, if n is_in S1 then suc(n) is_in S1;
Supposition n: S1;
    [InductiveSupposition] For all k, m: N,
        k + (m + n) = (k + m) + n
                                            by Definition S1;
    Goal suc(n) is_in S1;
    Goal for all k, m: N,
        k + (m + suc(n)) = (k + m) + suc(n);
    Goal for all k, m: N,
        if k is_in N and m is_in N then
        k + (m + suc(n)) = (k + m) + suc(n);
    Supposition k, m: N;
        Goal k + (m + suc(n)) = (k + m) + suc(n);
        k + (m + suc(n)) = k + suc(m + n)
                                        by (ii) of Definition +;
        (k + suc(m + n)) = suc(k + (m + n))
                                        by (ii) of Definition +;
        suc(k + (m + n)) = suc((k + m) + n)
                                    by InductiveSupposition;
        suc((k + m) + n) = (k + m) + suc(n)
                                        by (ii) of Definition +;
    Deduction if k is_in N and m is_in N then
        k + (m + suc(n)) = (k + m) + suc(n);
    [SucNAssociativity] For all k, m: N,
        k + (m + suc(n)) = (k + m) + suc(n)
                                    by universal generalization;
    suc(n) is_in S1
                                        by SucNAssociativity;
Deduction if n is_in S1 then suc(n) is_in S1;
for all n: N, suc(n) is_in S1
                                    by universal generalization;
0 is_in S1 and (for all n: N, suc(n) is_in S1)
                                        by ZeroInS1 & and rule;
N = S1
                        by Definition Monogeneric_Pty_3 &
                                                modus ponens;
For all k, m, n: N, k + (m + n) = (k + m) + n
                by Definition S1 & universal generalization;
Is_Associative( + )
                                by Definition Is_Associative( + );
QED

end Natural_Number_Theory_Proofs;
```

The proof language uses a syntax that mimics the traditional style of a mathematical proof. Provers such as Isabelle use a programming language-like syntax for expressing mathematics to enable ease of automation. Unfortunately, this very reason may make it less intuitive for traditional mathematicians. Because we have drawn a clear separation between automated verification of VCs and proof checking for theorems, we can use a language for writing proofs that is more intuitive for mathematical users. To this end, "Goals" are comments to state what the proof will try to do next; "Supposition/Deduction" pairs provide a mechanism for establishing implications; "definitions" can be introduced on the fly; and the "by" keyword introduces the rationale for the next step. A line of the proof can be given a label in square brackets for future reference.

This proof begins by stating a number of *Goal*s. The proof establishes a set, S1, which is defined to be the power set of N, for which the property of associativity already holds. The proof then proceeds with an induction over the natural numbers to prove that the set S1 is the same set as N. The base case of this induction is to prove that the natural number 0 is in S1, which is accomplished by using the identity property inherent in the definition of + to show that when adding zero, at least, + is associative, and thus 0 is also in S1

To see how straight forward the task of mechanization by the proof checker is, consider the italicized line that makes use of the "and" rule.

```
0 is_in S1 and (for all n: N, suc(n) is_in S1)
                                by ZeroInS1 & and rule;
```

It is simply the conjunct of the assertion labeled *ZeroInS1* with the assertion in the previous line.

A basic tenant of this proof checker is to approach a minimal basis of justifications to explain the transitions from step to step within a proof. These justifications act in concert with references to provide the rationale for a single step. A reference names one of the following kinds of entities:

- **Lemmas**, which are found in math précis or locally in a proof unit;
- **Theorems**, which are found in math précis;
- **Suppositions** that were established earlier in the proof;
- **Labels** that were given earlier in the proof;
- **Definitions/Corollaries**, which may be found inside a theory or defined earlier in the proof.

The available justifications are split into three groups based on the number of references they act on. The justifications requiring two references are as follows:

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

The justifications requiring one reference are:

- Equality
- Reductio ad absurdam
- Existential generalization
- Or rule
- Conjunct elimination
- Quantifier distribution
- Definition ∃!
- Universal instantiation
- Existential instantiation

Finally, the only justification requiring no references is:

- Excluded middle

4.PROOF CHECKER

Each justification has a well defined meaning that allows the proof-checker to determine if it is valid. Consider the following example of the semantics of *modus ponens*:

```
Γ, δ ⊢ [Label] B
    by [Reference2, ] Reference1 & modus ponens
    where {A, A → B} ⊆
        Extract{[Reference2, ] Reference1}, Γ, δ}
```

Γ represents the theories (with comprising theorems) currently in scope of the proof; δ represents the derivation of the proof so far; ⊢ is an operator indicating that the following application of a justification is valid; *A* and *B* are simply mathematical expressions; and the Extract function returns the set of mathematical expressions that have been assumed so far that correspond to the given names within either Γ or δ, or that was assumed in the immediately preceding line of the proof. Square brackets indicate an optional part.

So, overall this line is to be read, "A justification by *modus ponens* is permitted for establishing *B* if the given references and the expression of the immediately preceding line are sufficient to establish, from Γ and δ, that (*A* → *B*) and (*A*). In the future, the expression *B* may itself be referenced as *Label*."

As another example, here is the semantics of *alternative elimination*:

```
Γ, δ ⊢ [Label] A
    by [Reference2, ] Reference1 & alternative elimination
    where {B} ⊆ Extract{[Reference2, ] Reference1}, Γ, δ}
        and {A or B, B or A} ∩
        Extract{[Reference2, ] Reference1}, Γ, δ} ≠ ∅
```

This is to be read, "A justification by *alternative elimination* is permitted for establishing *A* if the given references and the expression of the immediately preceding line are sufficient to establish, from Γ and δ, ¬*B*, and at least one of (*A* or *B*) or (*B* or *A*) can be established in the same way. In the future, the expression *A* may itself be referenced as *Label*."

The current version of the proof checker is able to verify valid proofs (including the proof of associativity provided in Section 3), though higher-order theorems such as Is_Associative are not yet implemented. In addition, it is able to recognize and produce appropriate error messages for attempts to apply faulty justifications. We provide two examples of simple proofs containing errors in logic and the output of the proof checker when run on each.

First consider this working example:

```
Corollary Identity: a : N and a + 0 = a;

Proof of Theorem Nothing:
    Supposition k, m: N;
        (k + m) + 0 = k + m
                        by Corollary Identity & equality;
    Deduction if k is_in N and m is_in N then
        (k + m) + 0 = k + m;
QED
```

This proof simply establishes that given the identity property of addition on natural numbers, if two numbers, *k* and *m*, are natural numbers, then *(k + m) + 0 = (k + m)*.

Now consider an invalid application of the Identity Corollary:

```
Supposition k, m: N;

    (k + m) + 0 = m + 0 by Corollary Identity & equality;

Deduction if k is_in N and m is_in N then
    (k + m) + 0 = k + m;
```

When run through the proof-checker, this produces the following output:

```
Error: Simple.mt(10):
Could not apply substitution to the justified expression.
    (k + m) + 0 = m + 0 by Corollary Identity & equality;
```

Next, consider an invalid choice of justification to an otherwise valid step:

```
Supposition k, m: N;
    (k + m) + 0 = k + m by Corollary Identity & or rule;
Deduction if k is_in N and m is_in N then
    (k + m) + 0 = k + m;
```

When run through the proof-checker, this produces the following output:

```
Error: Simple.mt(10):
Could not apply the rule Or Rule to the proof expression.
      (k + m) + 0 = k + m by Corollary Identity & or rule;
```

5. RELATED WORK AND CONCLUSIONS

5.1 Isabelle

Isabelle is a proof assistant implemented in Standard ML and based on the specification language *Isar* [13, 16]. Its focus is on interactive proof development. It is also able to complete proofs automatically. Unlike earlier versions that closely resembled ML syntax, more recent versions have begun to put more of an emphasis on human readability of proofs. Even with these improvements, proofs contain artifacts of programming languages. For example, consider the following (trivial) proof that A and B → B and A where A and B are complicated expressions (called large_A and large_B in the proof) modified from [14]:

```
lemma assumes AB: "large_A ∧ large_B"
    shows "large_B ∧ large_A" ( is "?B ∧ ?A")
    using AB
proof
    assume "?A" "?B" show ?thesis ..
qed
```

While penetrable, it is harder to follow for those whose background is purely mathematical. Also, Isabelle proofs include statements to help ease automation, often interspersed with steps of the proof itself. By separating proofs that are merely checked from those that are totally automated, this difficult can be avoided.

Isabelle differs from RESOLVE as a language for proofs primarily with respect to its syntax, which maintains a programming language flavor. Another difference is that Isabelle provides no specific support for syntactically separating theorems from their proofs, though tools are provided for auto-generating documentation that serves much the same purpose. Also, Isabelle permits the bodies of proofs to be elided using a "sorry" command, which may allow unsound theorems to be introduced into the system.

5.2 Coq

Both Coq and RESOLVE share an emphasis on a small but extensible logical core and a specification language tailored for the tool itself (in the case of Coq, this language is called *Gallina*) [5]. Coq has limited automatic proving capabilities and a syntax more reminiscent of a programming language than a mathematical proof. As an example, consider a proof in Coq that A and B → B and A modified from [5]:

```
Variables A B C : Prop.

Lemma and_commutative : (A /\ B) -> (B /\ A).
    intro.
    elim H.
    split.
    exact H1.
    exact H0.
Save.
```

As with Isabelle, there is no explicit syntactic mechanism for separating theorems from their proofs; though, again, tools exist to automatically generate documentation. Also, like Isabelle, Coq provides a "trust me" command, which allows a proof to be elided.

5.3 PVS

PVS exists somewhere between a proof assistant and a theorem prover [2, 10, 11]. It uses a library of definitions and theorems to support the SMT solver *Yices* for the automatic verification of arithmetic expressions and equalities. Unlike RESOLVE, PVS has almost no emphasis on human-readable proofs. PVS's type checking system occasionally defers to the proof-checker to resolve ambiguous proof conditions. This contrasts with RESOLVE, where code, specifications, and proofs must all pass type checking before moving on to verification.

5.4 Nuprl

Like Isabelle, Nuprl is based on ML. Unlike RESOLVE, it does not perform type-checking on proofs [7, 8]. Nuprl relies on a built-in set of theories such as integers, function, and sets, which can only be extended by the use of tuples, unions, and lists. This contrasts with RESOLVE where only a minimal set of theories is provided (namely Boolean theory and a small portion of Set theory) from which other theories are built.

5.5 Conclusions

Software verification is a challenging problem. To address it effectively, a formal verification system that includes a verifying compiler needs to bring together the insights of programmers and mathematicians with advances in prover technology for mechanizing straightforward proofs, and proof checking for non-trivial theorems. Here we have presented a framework for addressing this challenge along with a summary of our efforts in proof checking. We plan much more experimentation with the ideas and tools presented here in order to make progress toward a sound and complete verification system.

6. ACKNOWLEDGMENTS

This research is funded in part by NSF grants DUE-0633506, DMS-0701187, DMS-0811748, and CCF-0811748. We thank the members of the RESOLVE/Reusable Software Research Group at Clemson and Ohio State for discussions on the topics presented in this paper. Special thanks are due to Jeremy Avigad at CMU, and Harvey Friedman and Bruce Weide at Ohio State. We thank the referees whose comments have helped improve the paper. We also thank the attendees of the RESOLVE meeting held at Clemson in 2007 when Kim Roche presented a draft version of these ideas.

7. APPENDIX

7.1 Stack Specification

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

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

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

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

  Operation Depth(restores S: Stack): Integer;
    ensures Depth = (|S|);

  Operation Rem_Capacity(restores S: Stack): Integer;
    ensures Rem_Capacity = (Max_Depth - |S|);

  Operation Clear(clears S: Stack);

end Stack_Template;
```

The concept Stack_Template is parameterized by a type, Entry, comparable to a generic in Java, and a Max_Depth that ensures each stack never becomes deeper than some capacity.

A type, Stack, is introduced, which is modeled on a mathematical string of Entrys. The constraints clause introduces a class invariant: the depth of a stack may never exceed Max_Depth. The initialization ensures clause guarantees that all implementations of Stack will ensure that new Stacks begin empty.

Next comes a list of the usual operations on Stacks. Each operation has a requires clause, which states the operation's pre-condition; and an ensures clause, which states its post-condition. In the ensures clause, a variable like S refers to the outgoing value of S while #S refers to the initial, incoming value of S. In addition to a type, each parameter in an operation has a *parameter passing mode*, such as alters or updates. These modes make certain assurances about the way in which a given parameter will be used. For instance, the alters mode indicates that the incoming value of the parameter is meaningful (and thus that variable may appear in the requires clause), but that the outgoing value of that parameter is undefined (and thus referring to the outgoing value in the ensures clause is illegal.) Updates indicates that both the incoming and outgoing values are defined. The others are similar.

As an example, the Pop operation takes a Stack, S, and an Entry, R, into which to pop the top entry. The requires clause states that there must be at least one Entry on S, and the ensures clause states that when we have finished, prepending R onto the final value of S will have the same value as the initial value of S.

7.2 VCs Resulting from Obvious_F_C_Realiz

```
Free Variables: Max_Depth:*Z, min_int:*Z, max_int:*Z,
S:*Str(*Entry), ?S:*Str(*Entry), ?Next_Entry:*Entry, ?
S_Reversed:*Str(*Entry), Next_Entry:*Entry,
S_Reversed:*Str(*Entry)

((((min_int <= 0) and (0 < max_int)) and ((min_int <= 0) and
(0 < max_int) and (Max_Depth > 0))) and (|S| <= |
Max_Depth|))
======================>
S = (Rev(empty_string) o S)

(((((min_int <= 0) and (0 < max_int)) and ((min_int <= 0)
and (0 < max_int) and (Max_Depth > 0))) and (|S| <= |
Max_Depth|)) and (S = (Rev(?S_Rev) o ?S) and |?S| /= 0))
======================>
((1 + |?S_Reversed|) <= Max_Depth)

(((((min_int <= 0) and (0 < max_int)) and ((min_int <= 0)
and (0 < max_int) and (Max_Depth > 0))) and (|S| <= |
Max_Depth|)) and (S = (Rev(?S_Reversed) o ?S) and |?S| /
= 0))
======================>
(Rev(?S_Reversed) o ?S) = (Rev((<?Next_Entry> o ?
S_Reversed)) o ?S)

(((((min_int <= 0) and (0 < max_int)) and ((min_int <= 0)
and (0 < max_int) and (Max_Depth > 0))) and (|S| <= |
Max_Depth|)) and (S = (Rev(?S_Reversed) o ?S) and |?S| /
= 0))
======================>
(|?S| < |?S|)

((((min_int <= 0) and (0 < max_int)) and ((min_int <= 0) and
(0 < max_int) and (Max_Depth > 0))) and ((|S| <= |
Max_Depth|) and (S = (Rev(?S_Reversed) o ?S) and |?S| =
0)))
======================>
?S_Reversed = Rev((Rev(?S_Reversed) o ?S))
```

8. REFERENCES

[1] "ACL2 Version 3.4: The User's Manual." http://www.cs.utexas.edu/users/moore/acl2/v3-4/acl2-doc.html#User%27s-Manual

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

[3] M. D. Ernst. Dynamically discovering likely program invariants. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[4] D. E. Harms and B. W. Weide, Copying and Swapping: Influences on the Design of Reusable Software Components, *IEEE Transactions on Software Engineering*, Vol. 17, No. 5, May 1991, pp. 424 - 435.

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

[6] H. Kirschenbaum, K. Harton, and M. Sitaraman, A Case Study in Automated Verification, *Proceedings of CAV/AFM Workshop*, Princeton, NJ, July 2008.

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

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

[9] RESOLVE Compiler and Verifier. http://www.cs.clemson.edu/~resolve/compiler-verifier.html.

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

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

[12] M. Sitaraman, and B. Weide, eds., Special Feature: Component-Based Software Using RESOLVE, *Software Engineering Notes 19*, 4 (October 1994), 21-22.

[13] T. Nipkow, L. C. Paulson, M. Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic." New York: Springer-Verlag, 2008, Sections 1.1, 1.2, and 2.3.

[14] T. Nipkow. "A Tutorial Introduction to Structured Isar Proofs," http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/isar-overview.pdf.

[15] B. Weide, M. Sitaraman, H. K. Harton, B. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum and D. Frazier. Incremental Benchmarks for Software Verification Tools and Techniques. *Proceedings of VSTTE 2008*, Toronto, CA, Oct 2008, to appear.

[16] M. Wenzel, "Isabelle/Isar: Reference Manual", June 2008, www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/isar-ref.pdf. Section 4.4.

[17] J. M. Wing. Using Larch to specify Avalon/C++ objects. *IEEE Transactions on Software Engineering*, Vol. 16, No. 9, September 1990, pp. 1076-1088.