Incremental Benchmarks for Software Verification Tools and Techniques

Bruce W. Weide[1], Murali Sitaraman[2], Heather K. Harton[2], Bruce Adcock[1],
Paolo Bucci[1], Derek Bronish[1], Wayne D. Heym[1], Jason Kirschenbaum[1],
and David Frazier[2]

[1] The Ohio State University, Columbus, OH 43210, USA
{weide,adcockb,bucci,bronish,heym,kirschen}@cse.ohio-state.edu
[2] Clemson University, Clemson, SC 29634, USA
{murali,hkeown,dfrazie}@cs.clemson.edu

Abstract. This paper proposes an initial catalog of easy-to-state, relatively simple, and incrementally more and more challenging benchmark problems for the Verified Software Initiative. These benchmarks support assessment of verification tools and techniques to prove total correctness of functionality of sequential object-based and object-oriented software. The problems are designed to help evaluate the state-of-the-art and the pace of progress toward verified software in the near term, and in this sense, they are just the beginning. They will allow researchers to illustrate and explain how proposed tools and techniques deal with known pitfalls and well-understood issues, as well as how they can be used to discover and attack new ones. Unlike currently available benchmarks based on "real-world" software systems, the proposed challenge problems are expected to be amenable to "push-button" verification that leverages current technology.

1 Introduction

> This, however, is my doctrine: whoever would learn to fly must first learn to stand and walk and run and climb and dance: one cannot fly into flying!
>
> — Friedrich Nietzsche, in *Thus Spake Zarathustra*, Third Part, 1884

> You can't get to the moon by climbing a tree.
>
> — William F. Ogden, paraphrase of S.E. Dreyfus in *Mind Over Machine*, p. 10, 1986

As the two views above reveal, there are (at least) two kinds of ambitions: ones for which incremental gains gradually lead to their achievement, and ones for which no merely incremental gains can possibly lead to their achievement.

Is it possible to reach incrementally the ultimate goal of the Verified Software Initiative (VSI), i.e., routine verification of practical software systems? There are certainly paths that are doomed not to reach this target no matter how much incremental progress can be made. For example, a proof system that is inherently not modular simply is not going to scale up to verification of practical software systems of the sort

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 84–98, 2008.

the VSI needs to tackle—regardless of how much incremental progress is made on its details. Yet history suggests that the lofty goals of the VSI, like similar goals in other disciplines, are more likely to be achieved by repeated acts of "standing on the shoulders of giants" (and others) than by an ambitious tree-climber suddenly striking out in a space ship and flying to the moon.

The benchmarks proposed in this paper, therefore, define strategically placed milestones along a path toward the goal of verified software. The spacing and nature of benchmarks to define these milestones have been *designed* to capture the anticipated incremental nature of the community effort. Two criteria have guided their development:

- They should be similar enough that incremental progress toward the goal can be recognized—and explained to the community as simply as possible.
- They should be designed especially for verification, so that progress in overcoming specific known barriers to achieving the goal can be readily demonstrated—and explained to the community as simply as possible.

1.1 Contributions

The primary contribution of this paper is its proposed set of eight *purposefully designed, incremental, early benchmarks for tools and techniques to prove total correctness of sequential object-based and object-oriented software*. These benchmarks come with a classification of the issues they raise for automated software verification and a methodology for reporting progress. They are intended to precede (in difficulty) and complement rather than replace earlier challenge problems (e.g., [1,2,3,4]); they certainly should not be seen as diminishing the significance of claimed solutions to those problems (e.g., [3,5,6]). The proposed benchmarks differ from earlier challenge problems in key respects. Most important, they are far simpler. If a system cannot be used to verify the proposed benchmarks, it is unlikely to be powerful enough to verify software that operates an e-commerce site, or implements a file system, or controls a non-trivial embedded device such as a heart pacemaker. Put otherwise, if a verification system is capable of solving complex real-world benchmarks, then there should be no problem applying it to these simpler benchmarks and using this opportunity to explain how the approach overcomes each of a number of known bumps in the road for software verification.

The proposed benchmarks are also highly focused, so tools and techniques that are not intended to address every aspect of a real-world software system can still be applied and evaluated for some or all of these benchmarks. This means that individuals or groups who are able to contribute a few pieces of the big puzzle are empowered to play. Solutions to these benchmarks can become verified software in the VSI repository. Once several systems using different languages, approaches, etc., have been applied to a given benchmark, it will be easier for the community to evaluate their relative merits and eventually to design new tools and techniques that leverage the best attributes of different efforts.

The above features also mean it is reasonable to hope for complete automation and "push-button" technology in benchmark solutions. Once humans formalize the requirements in a benchmark problem statement into specifications (supported as necessary by new or extended mathematical theories), write code (annotated as necessary with assertions expected by the proposed technique), and provide any other necessary inputs, no further human intervention should be needed to complete the verification. The result might be a simple "proved" or "not proved", or in the latter case it might be reported in a form that simplifies debugging.

1.2 Limitations

There certainly are other benchmarks that would be appropriate for the territory covered by the ones proposed here. We do not attempt to justify that these are the "best" ones, only that they are a decent starting point. We do not even begin to suggest benchmarks that would help assess progress on other paths leading toward the VSI goal, or later benchmarks on this path: proving properties entailed by, but other than, total functionality correctness (e.g., absence of null dereferences); performance correctness (e.g., compliance with execution time or memory usage specifications); concurrency issues; distributed systems issues; graphical user interface issues; meta-level issues such as soundness and relative completeness of proof systems; proofs in programming language meta-theory [7]; and so on. The proposed new benchmarks venture nowhere near the latter areas.

There is not necessarily a total order among the issues that are raised by the proposed benchmarks. One might be able to solve a later benchmark problem before an earlier one. The order shown is consistent with a natural partial order among the primary issues that we identify as important problems to be addressed by the proposed benchmarks. In fact, some of these benchmarks have already been "solved", though we do not cite such claims. This is fine; the more different solutions the better, so the community can compare their pros and cons. And if someone chooses to start from scratch rather than building on other work to attack some of these challenges, then there is relatively little risk of massive lost effort, but there might be potentially high payoff: all the proposed benchmarks are close to the beginning of the journey and might help illustrate new ideas that are not derived as variations on previous work.

The rest of the paper is organized as follows. Section 2 presents the benchmark problem requirements statements, identifies the major issues related to each, and lists some variations of each challenge that might be considered by those proposing solutions. Section 3 describes two possible solutions to one part of the simplest benchmark, so prospective benchmark solvers have a couple of examples of what we mean. Section 4 reviews and concludes.

2 Benchmark Problems

All the proposed benchmark problems require that tools and techniques purported to solve them should have the following features:

- The proposed solution should include:
    - o all formal specifications relevant to the benchmark problem requirements, including mathematical definitions, theories, and similar artifacts developed for and/or used in the specifications;
    - o all code subjected to the verification process;

  - o all verification conditions (VCs) involved in the verification process;
  - o descriptions of the verification system proof rules employed, tools used, and techniques applied.

  As some of this information may be unwieldy or impossible to include in an exposition of acceptable length, a web site with details should be provided so readers and reviewers can check any details they deem important. (Section 3 contains links to such sites for the example solutions outlined there.)

- The proposed solution should involve both an automatic proof of total correctness of a correct solution, and evidence that the tools and techniques can automatically detect that a "slightly" incorrect solution is incorrect. Specifically, in addition to verified correct code, a benchmark solution should present a perturbed version of the code—which plausibly could have been written to meet the same specification—along with a demonstration that a bug is found automatically. For example, perhaps the proposed solution generates meaningful counterexamples as test cases that lead to failure for defective code. This requirement also could be regarded as a kind of mutation test.
- The proposed verification approach should be modular. In other words, a proof that a program unit *P* implements a specification *S* should be based only on *P*, *S*, and the specifications of the program units that *P* depends on (not their implementations). Moreover, the verification should need to be performed only once, not again for every context in which *P* is used for *S*.
- Verified software from benchmark solutions should be submitted formally to the VSI repository [8].

2.1 Benchmark #1: Adding and Multiplying Numbers

Problem Requirements: Verify an operation that adds two numbers by repeated incrementing. Verify an operation that multiplies two numbers by repeated addition, using the first operation to do the addition. Make one algorithm iterative, the other recursive.

Issues: Addition is straightforward. It is a single operation and involves a single numeric type: either integers or natural numbers, which may be a built-in type in the programming language or a user-defined abstract data type (ADT). It also involves simple specifications, and presumably results in VCs from a decidable mathematical theory (Presburger arithmetic). Multiplication explicitly adds the requirement for modularity for a program unit that is a single operation, i.e., procedural or functional abstraction, and enough richness that the underlying mathematics is undecidable. Since one operation involves iteration and the other recursion, loop invariants and termination both become concerns. There are alternative algorithms for multiplication based on addition, e.g., the Egyptian algorithm [9] as well as the more obvious and familiar ones. All the standard procedural programming constructs are likely to arise in these two pieces of code.

Variations: Other operations on integers or natural numbers, e.g., computing powers and roots, involve similar issues and should illustrate similar verification capabilities. The programming type used for numbers may or may not be bounded. Numerical problems of this kind that involve floating point numbers are very interesting as well, raising especially difficult issues about how to specify their behavior that do not arise with either bounded or unbounded integers or natural numbers.

2.2 Benchmark #2: Binary Search in an Array

Problem Requirements: Verify an operation that uses binary search to find a given entry in an array of entries that are in sorted order.

Issues: Arrays are the traditional first "collection" type, which may be a built-in type in the programming language or an ADT. A mathematical theory sophisticated enough to handle the array specification is an additional complication [10]. Unless this theory contains special operators that hide them, quantifiers are involved in the specification; to simplify automated verification in the presence of existential quantifiers, ghost (or adjunct) variables may be needed in the specification and/or implementation. This code also involves at least two types: the numerical index type and the array type, and possibly a separate entry type. A recommended candidate for the incorrect version of the code is any close cousin of the Java binary search function whose defect was pointed out recently in a Google blog "after lying in wait for nine years or so" [11]. The numerical index type for the array must be considered bounded to manifest the defect in that code.

Variations: An array may have fixed bounds, or they may be dynamically adjustable. If the type of the array elements is not fixed but rather is a parameter to the specification(s) and code, then additional issues arise here, before benchmark #3 brings them front and center. In particular, the actual entry type might be an ADT itself; the specification and computation of the ordering among entries and equality of entries become interesting.

2.3 Benchmark #3: Sorting a Queue

Problem Requirements: Specify a user-defined FIFO queue ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an operation that uses this component to sort the entries in a queue into some client-defined order.

Issues: Dealing with a generic collection type is one new issue here. In addition, a mathematical theory suitable for specifying and verifying queue behavior may be needed—perhaps different than that used for arrays in benchmark #2 [12]. Parameterizing the sort operation to account for the specification and computation of the ordering among entries is central. Implementations that involve nested loops may require ghost variables to simplify writing loop invariants.

Variations: Some variations that delay the inevitable generic type issues might keep the problem easier, while leaving it incrementally further along the path to the goal than benchmark #2. For example, even if the queue entries are a fixed type with a fixed ordering for sorting, the mathematical definitions and/or new theory to address even this simplified version of the benchmark might be troublesome.

2.4 Benchmark #4: Layered Implementation of a Map ADT

Problem Requirements: Verify an implementation of a generic map ADT, where the data representation is layered on other built-in types and/or ADTs.

Issues: This may be viewed as "recasting" benchmark #2 as an ADT. The issues of proof of correctness of data representation [13] are now involved, including representation invariants and abstraction relations [14]. There are many versions of "map" behavior, including impoverished ones that obscure important issues. The map designed for this benchmark should be as useful to clients as, say, maps in the *java.util* package. If any of the map operations involves relational behavior, a careful definition of correctness might be subtle.

Variations: A simple data representation such as a queue with linear search as the primary algorithm is an obvious starting point. There are many other data structures and algorithms with better performance that are more realistic (e.g., hash tables of various ilks, binary search trees with or without balancing). Each of these alternatives might involve specifying other interesting ADTs as the basis for the data representation. Additional issues might arise from this exercise.

2.5 Benchmark #5: Linked-List Implementation of a Queue ADT

Problem Requirements: Verify an implementation of the queue type specified for benchmark #3, using a linked data structure for the representation.

Issues: Pointers/references may be either built-in types in the programming language or user-defined types. Specifications for their behavior, and verification of linked data structures that use them, raise many interesting questions [15,16,17]. Most obvious is that control of aliasing becomes an issue. If this is not addressed carefully, retaining modularity of verification might be difficult.

Variations: A linked data structure might be encoded in an array with integer indices, so issues related to linked data structures are separated from issues related to language-defined pointers/references as a way of introducing indirection. If memory is allocated dynamically, then memory might be assumed to be unbounded or bounded, the latter raising the specter of memory allocation that does not always succeed. The queues themselves might be specified to be unbounded, or bounded in various ways (e.g., uniformly bounded so each queue has the same maximum length, or communally bounded so the sum of the lengths of all queues is bounded).

2.6 Benchmark #6: Iterators

Problem Requirements: Verify a client program that uses an iterator for some collection type, as well as an implementation of the iterator.

Issues: There are many proposed designs for iterators, each of which raises a number of interesting specification and verification issues that involve coupling between an underlying collection type and an iterator for it. Specifying iterators (though not verifying anything about them) was a challenge problem issued for the *SAVCBS Workshop* in 2006 [18,19].

Variations: Iterators may be active or passive [20]. Each design has its own set of problems [21]. A passive iterator, where an operation is passed to the iterator and the iterator applies it to each entry, starts to raise issues similar to those involved in callbacks. None of the benchmarks proposed here deals with callbacks.

2.7 Benchmark #7: Input/Output Streams

Problem Requirements: Specify simple input and output capabilities such as character input streams and output streams (with the flavor of C++ streams, for example). Verify an application program that uses them in conjunction with one of the components from the earlier benchmarks.

Issues: Technically, modeling input and output might involve concurrent processes operating alongside an application program. It is far from clear this is the best way to view the situation, though, and the modeling and specification issues that arise could take a solution in any of a number of intriguing directions.

Variations: A version of I/O streams involving "standard input and output" streams that can be redirected from/to files is one way to keep I/O in a program separate from the notion of a file system. A variation that includes the possibility of opening I/O streams to files by their filenames entails coupling with the file system, raising a host of issues related to persistent data. Going this direction might involve connections with a challenge problem for the *ABZ 2008 Conference*, i.e., specification and verification of the POSIX file-store interface of Unix [1]. A variation that allows I/O streams to serve as an interprocess communication mechanism raises yet other issues that lead into the full gamut of concurrency questions.

2.8 Benchmark #8: An Integrated Application

Problem Requirements: Verify an application program with a concisely stated set of requirements, where the particular solution relies on integration of at least a few of the previous benchmarks. For example, verify an application program that does the following: Given input containing a series (in arbitrary order) of terms and their definitions, output an HTML glossary that presents all the terms and their definitions, with (a) the terms in alphabetical order, and (b) a hyperlink from each term that occurs in any definition to that term's location in the glossary.

Issues: This benchmark adds the issues involved in composing a number of user-defined types and operations. Code for a glossary generator, for instance, might use the sorting operation from benchmark #3, the map type from benchmark #4, I/O streams from benchmark #7, and perhaps others. Modularity implies that implementations of all the components being integrated need not be verified, too, but of course they must be specified in order for modular verification of the application program to proceed.

Variations: If the I/O stream specifications from benchmark #7 include access to the file system, then an interesting variation of the glossary generator requirements is to structure the glossary as a single index page plus a set of HTML pages, one per term, with hyperlinks between the pages.

3 Two Solutions to Part One of Benchmark #1

Two solutions to the "addition by repeated incrementing" part of benchmark #1 are presented as guides to an appropriate level of detail for a proposed solution. These solutions use syntactically slightly different, though semantically identical, dialects of the RESOLVE language [22,23,24]. The first solution is recursive and uses a VC generator under development at Clemson; the VCs are proved using the automated proof assistant Isabelle [25]. For variety, the second solution is iterative and uses a different set of tools under development at Ohio State to generate and prove the VCs. The benchmark requirement to show a "slightly" incorrect version is not illustrated here, but such code is available at the respective web sites mentioned below.

3.1 Recursive Procedure for Addition by Repeated Incrementing

For more details, see:

http://www.cs.clemson.edu/~resolve/benchmarks

Input Specification and Code for Verification: The specification and implementation of the Add_to operation are given with access to a specification of an Integer type and operations. Parameter modes, only **updates** and **evaluates** in this code, are specification constructs. The **updates** mode indicates the parameter may be modified in any way permitted by the **ensures** clause. The **evaluates** mode allows an expression to be passed as the corresponding argument, i.e., it indicates a "value" parameter. In an **ensures** clause, the prefix # for a formal parameter denotes the incoming value. Recursive code is annotated with a progress metric using the keyword **decreasing**.

```
Operation Add_to(updates i:Integer; evaluates j:Integer);
    requires min_int <= i + j and i + j <= max_int and j >= 0;
    ensures  i = #i + j;
```

```
Procedure Add_to(updates i:Integer; evaluates j:Integer);
    decreasing |j|;
        Var z: Integer;
        If (not Is_Zero(j)) then
            Increment (i);
            Decrement (j);
            Add_to (i, j);
        end;
end Add_to;
```

In RESOLVE, all types (including those built-in to most languages) are specified, used, and verified in a uniform way. Integer is specified in Integer_Template shown below, where Z denotes mathematical integers; this is defined in a mathematical module Integer_Theory (not shown) along with mathematical notations such as 0, + , and -. This specification **defines** two constants min_int and max_int. Every Integer variable is constrained to be within these bounds, and is initially 0.

```
Concept Integer_Template;
    uses Integer_Theory, Std_Boolean_Fac;

    Defines min_int: Z;
    Defines max_int: Z;
    Constraint min_int <= 0 and 0 < max_int;

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

    Operation Is_Zero(evaluates i: Integer): Boolean;
        ensures  Is_Zero = ( i = 0 );

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

    Operation Decrement(updates i: Integer);
        requires min_int <= i - 1;
        ensures  i = #i - 1;
     …
end Integer_Template;
```

Verification Conditions: VCs are generated using a tool that implements the proof rules described in [26,27]. Briefly, there is a VC for each state in the program where the next statement involves establishing a precondition for the next operation, a loop invariant or progress metric, or the postcondition for the operation being verified. The 8 VCs are shown below. All variables are of type Z. Each VC is independent of the others, and consists of a goal that needs to be proved (below the line) using several hypotheses (above the line). For example, VC 3 arises from the ordinal-valued **decreasing** clause to show termination.

```
(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
______________________________
((i + 1) <= max_int)

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
______________________________
(min_int <= (j - 1))

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
______________________________
(|(j - 1)| < |j|)

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
______________________________
```

```
(min_int <= ((i + 1) + (j - 1)))

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
_____________________________
(((i + 1) + (j - 1)) <= max_int)

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
_____________________________
((j - 1) >= 0)

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and not(j = 0))))
_____________________________
((i + 1) + (j - 1)) = (i + j)

(((min_int <= 0) and (0 < max_int)) and ((((min_int <= j) and (j <=
max_int)) and (((min_int <= i) and (i <= max_int)) and (((min_int <=
(i + j)) and ((i + j) <= max_int)) and (j >= 0)))) and (P_val = |j|
and j = 0)))
_____________________________
i = (i + j)
```

Verification Results: Isabelle-friendly versions of the VCs are generated to provide proofs. Proofs of all VCs are completed automatically by Isabelle with "apply force".

3.2 Iterative Procedure for Addition by Repeated Incrementing

For more details, see:

http://www.cse.ohio-state.edu/rsrg/benchmark-results/add-mult

Input Specification and Code for Verification: The numbers in this version of the problem are unbounded natural numbers, which also are not built-in. The parameter mode **restores** means there is no net change to the parameter's value, i.e., it means implicit addition of a conjunct such as m = #m to the **ensures** clause.

```
procedure Add (updates n: Natural, restores m: Natural)
    ensures
        n = #n + m
```

```
procedure Add (updates n: Natural, restores m: Natural)
    variable k, z: Natural
    loop
        maintains n + m = #n + #m  and  k + m = #k + #m  and
                  z = 0
        decreases m
    while not AreEqual (m, z) do
        Increment (n)
        Increment (k)
        Decrement (m)
    end loop
    m :=: k
end Add
```

```
contract UnboundedNaturalFacility

    math subtype NATURALMODEL is integer
        exemplar n
        constraint n >= 0

    type Natural is modeled by NATURALMODEL
        exemplar n
        initialization ensures n = 0

    procedure Increment (updates n: Natural)
        ensures  n = #n + 1

    procedure Decrement (updates n: Natural)
        requires n > 0
        ensures  n = #n - 1

    function AreEqual (restores m: Natural,
                       restores n: Natural): control
        ensures  AreEqual = (m = n)

    function IsGreater (restores m: Natural,
                        restores n: Natural): control
        ensures  IsGreater = (m > n)

    function Replica (restores n: Natural): Natural
        ensures  Replica = n

end UnboundedNaturalFacility
```

The algorithm for addition is the obvious iterative one. The only possibly unusual part of the code is the swap statement after the loop, which is a RESOLVE staple that replaces assignment as the built-in data movement operator.

Verification Conditions: VCs are generated in a locally defined XML format, using a tool we have developed that implements the proof rules described in [28,29]. This XML is then translated into human-readable unicode format and into input for Isabelle [25], and also is piped directly into SplitDecision (see below).

As in Section 3.1, there is a VC for each state in the program where the next statement involves establishing a pre-condition for the next operation, a loop invariant or progress metric, or the post-condition for the operation being verified. However, the proof rules in this version involve profligate use of mathematical variables: one for each program variable in each program state (between consecutive statements). So, for example, x_i stands for the value of program variable x in state i. In order to relieve the burden of automatic case analysis by the back-end prover, the VC generator divides each VC into cases based on the path structure of the code. In addition, it does a number of simplifications that do not rely on knowledge of particular mathematical domains, but rather depend only on general logical properties (e.g., substitution of equals). There are too many VCs to show here (12 total, the largest having 23 hypotheses before any simplification and 6 afterward); a representative VC is reproduced below both in a human-readable form and in the form generated for Isabelle. This VC is from state 5, just after the `Decrement` call inside the loop, and arises because we must show that (each clause of) the loop invariant holds at the end of the loop body.

Lemma #6 (state index: 5)

$$\begin{array}{ll}
\hline
 & n_0 \geq 0 \\
\wedge & m_0 \geq 0 \\
\wedge & m_2 \neq 0 \\
\wedge & n_2 + m_2 = n_0 + m_0 \\
\wedge & k_2 + m_2 = 0 + m_0 \\
\wedge & m_2 > 0 \\
\hline
\Rightarrow & n_2 + 1 + (m_2 - 1) = n_0 + m_0
\end{array}$$

```
lemma 6:
"[|
     (n_0::int) >= 0 ;
     (m_0::int) >= 0 ;
     ~(m_2::int) = 0 ;
     (n_2::int) + m_2 = n_0 + m_0 ;
     (k_2::int) + m_2 = 0 + m_0 ;
     m_2 > 0
  |]
  ==>
     n_2 + 1 + (m_2 - 1) = n_0 + m_0"
apply (((simp only: simp_thms),clarify?)+)?
apply (force+)?
done
```

Verification Results: SplitDecision is a simplifier we are developing that uses mathematical properties involving theories used in RESOLVE specifications to further simplify VCs, based on knowledge of predicate calculus with equality and using special-purpose decision procedures we have designed for relevant fragments of the mathematical theories of integers and strings (so far). Proofs (simplifications all the way to “true”) of all the VCs for this problem are completed automatically and quickly by both Isabelle and SplitDecision.

4 Discussion and Conclusions

As others also have suggested, albeit without elaborating a set of benchmarks [30], it will be helpful to have several reachable milestones along the road to the VSI goal of routinely verifying real-world software systems. In fact, one of the reviewers of this paper summarized the point very clearly: “A software verification system failing the benchmark suite can hardly be ready for proving the correctness of serious software.” The benchmarks and methodology proposed here for documenting incremental progress constitute a first step in defining specific milestones and capabilities.

While we have focused on code verification in our sample solutions, it is clear that the benchmarks also could be used as suitable targets for illustrating issues in specification, generating internal assertions, teaching formal verification, or for proving certain classes of verification conditions. An important feature toward this end is that the proposed benchmarks are stated in terms of brief requirements statements. Why not start with “reference formal specifications”? The software community still does not agree about how formal specifications should be written, or even about the design and intended behavior of the simplest components; if there is any doubt about this, compare the JML [31] and RESOLVE/C++ [32] specifications for a stack component. It is, therefore, unlikely that a verification system that is required by benchmark statements to adopt someone else's software designs and specifications would ever be tried—let alone succeed—on such benchmarks. A similar conclusion applies to the programming language used for implementations. While it will be exciting if and when the VSI effort ultimately succeeds in creating tools that can verify software that is not designed for verification, e.g., open-source software, we submit this is not the place to start. The hope is that the proposed benchmarks *will not appear daunting* to those who claim to have automated verification systems for proving total correctness, and that this very feature will encourage everyone to participate and thereby facilitate comparisons among different approaches along many dimensions of the problem.

We trust the VSI community also will find it an acceptable challenge to improve upon and augment this set of benchmarks to illustrate other important issues in relatively simple and incremental ways. For example, if there is widespread interest in moving in such directions, then those interested in exploring them should devise additional benchmarks—meeting the general requirements outlined at the beginning of Section 2—to address verification of open-source software, software specified in certain styles and languages, software written in specific commercial languages or with specific features, etc.

Acknowledgments

The authors appreciate the contributions of Jeremy Avigad, Harvey M. Friedman (whose decision procedure for strings with some restrictions is used in SplitDecision), Greg Kulczycki, Bill Ogden, and Anna Wolf. This work is supported in part by the National Science Foundation under grants DMS-0701187 and DMS-0701260.

References

1. ABZ call for papers on the POSIX pilot project in the grand challenge (viewed April 27, 2008), http://www.abz2008.org
2. Pacemaker formal methods challenge (viewed April 27, 2008), http://www.cas.mcmaster.ca/sqrl/pacemaker.htm
3. Stepney, S., Cooper, D., Woodcock, J.: An electronic purse: specification, refinement, and proof. PRG Technical Monograph PRG-126 (2000), 231 pages (viewed April 27, 2008), http://web2.comlab.ox.ac.uk/oucl/publications/monos/prg-126.html
4. Woodcock, J., Banach, R.: The verification grand challenge. Journal of Universal Computer Science 13(5), 661–668 (2007)
5. Schmitt, P.H., Tonin, I.: Verifying the Mondex case study. In: Fifth IEEE Intl. Conf. on Software Engineering and Formal Methods, pp. 47–58. IEEE, Los Alamitos (2007)
6. Haneberg, D., Schellhorn, G., Grandy, H., Reif, W.: Verification of Mondex electronic purses with KIV: from transactions to a security protocol. Formal Aspects of Computing 20(1), 41–59 (2007)

7. The POPLmark challenge (viewed April 27, 2008), http://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark/index.php?title=The_POPLmark_Challenge
8. Bicarregui, J.C., Hoare, C.A.R., Woodcock, J.C.P.: The verified software repository: a step towards the verifying compiler. Formal Aspects of Computing 18(2), 143–151 (2006)
9. Ancient Egyptian multiplication (viewed April 27, 2008), http://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication
10. Bradley, A.R., Manna, Z.: The Calculus of Computation. Springer, Heidelberg (2007)
11. Bloch, J.: Extra, extra – read all about it: nearly all binary searches and mergesorts are broken (2006) (viewed 27 April 2008), http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
12. Zhang, T., Sipma, H.B., Manna, Z.: Decision procedures for queues with integer constraints. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 225–237. Springer, Heidelberg (2005)
13. Hoare, C.A.R.: Proof of correctness of data representations. Acta. Inf. 1(4), 271–281 (1972)
14. Sitaraman, M., Weide, B.W., Ogden, W.F.: On the practical need for abstraction relations to verify abstract data type representations. IEEE Transactions on Software Engineering 23(3), 157–170 (1997)
15. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proc. 17th Annual IEEE Symp. on Logic in Computer Science, pp. 55–74. IEEE, Los Alamitos (2002)
16. Kulczycki, G.: Direct Reasoning. Dept. of Computer Science, Ph.D. thesis, ClemsonUniversity, Clemson, SC (2004)
17. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: ACM Conference on Programming Language Design and Implementation, pp. 349–361. ACM Press, New York (2008)
18. Challenge problem: iterator specification (viewed April 27, 2008), http://www.eecs.ucf.edu/~leavens/SAVCBS/2006/challenge.shtml
19. Leavens, G., (ed.): SAVCBS 2006 Proceedings: Specification and Verification of Component-Based Systems (viewed April 27, 2008), http://www.eecs.ucf.edu/~leavens/SAVCBS/2006/SAVCBS06-proceedings.pdf
20. Booch, G.: Software components with Ada. Benjamin Cummings, Redwood City, CA (1987)
21. Weide, B.W., Edwards, S.H., Harms, D.E., Lamb, D.A.: Design and specification of iterators using the swapping paradigm. IEEE Transactions on Software Engineering 20(8), 631–643 (1994)
22. Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., Weide, B.W.: Specifying components in RESOLVE. Software Engineering Notes 19(4), 29–39 (1994)
23. Bucci, P., Hollingsworth, J.E., Krone, J., Weide, B.W.: Implementing components in RESOLVE. Software Engineering Notes 19(4), 40–52 (1994)
24. Kulczycki, G., Sitaraman, M., Yasmin, N., Roche, K.: Formal specification. In: Encyc. of Computer Science and Engineering. John Wiley & Sons, Chichester (to appear, 2008)
25. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
26. Krone, J.: The Role of Verification in Software Reusability. Ph.D. dissertation, Dept. of Comp. and Inf. Science, Ohio State Univ., Columbus, OH (1988)
27. Harton, H., Krone, J., Sitaraman, M.: Formal program verification. In: Encyc. of Computer Science and Engineering. John Wiley & Sons, Chichester (to appear, 2008)

28. Heym, W.D.: Computer Program Verification: Improvements for Human Reasoning. Ph.D. dissertation, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH (1995)
29. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S., Hollingsworth, J.E.: Reasoning about software-component behavior. In: Frakes, W.B. (ed.) ICSR 2000. LNCS, vol. 1844, pp. 266–283. Springer, Heidelberg (2000)
30. Zhang, J.: Program verification in the small. In: Proc. First Asian Working Conf. on Verified Software, UNU/IIST Report #348, pp. 83–84 (2006)
31. JML BoundedStackInterface (viewed July 15, 2008), `http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/org/jmlspecs/samples/stacks/BoundedStackInterface.html`
32. RESOLVE/C++ Catalog, AT/Stack/Kernel.h (viewed July 15, 2008), `http://www.cse.ohio-state.edu/sce/rcpp/RESOLVE_Catalog-HTML/AT/Stack/Kernel.html`