Restrictions: Help in Documenting Client Code Under a Verified Software Paradigm

Jason Kirschenbaum  
Dept. of Computer Science and Engineering  
The Ohio State University  
Columbus, OH 43210, USA  
kirschen@cse.ohio-state.edu

Bruce W. Weide  
Dept. of Computer Science and Engineering  
The Ohio State University  
Columbus, OH 43210, USA  
weide@cse.ohio-state.edu

ABSTRACT
A novel programming language construct, restrictions, provides a mechanism to document abstract invariants of program variables and also may simplify program correctness proofs of the use of components. Examples illustrating the use and utility of restrictions are presented.

1. INTRODUCTION
It has long been claimed in some circles that software professionals cannot be expected to write mathematically rigorous descriptions of their code such as formal specifications and loop invariants [1]. This contention arguably underestimates the capabilities of software professionals—after all, most of them have not been taught either why or how to write such annotations, so it is not surprising they are currently unequipped to do so. Nonetheless, the perception has led to exploration of some promising mitigating techniques that might be useful under a verified software paradigm. One approach involves inferring invariants (e.g., loop invariants) either by dynamic or static analysis of code [2, 3, 4, 5, 6]. A complementary approach involves minimizing what needs to be written in mathematical language by providing special syntax for certain situations: syntax that looks more familiar and code-like to software developers. For instance, rather than demanding that the post-condition of an operation include a clause like \( x' = x = \text{old}(x) \) or \( x = #x \) to specify that the value of \( x \) does not change, most specification languages have tailor-made syntax for documenting this. JML [7] uses a modifies clause to list operation parameters whose values the parameters point to might be changed during the operation body. RESOLVE [8] offers (among others) a restores parameter “mode” to state that an operation parameter, while it might change temporarily during the body of the operation, has the same value at the end of the operation body as it had at the beginning.

Such mechanisms incrementally reduce the mathematical annotation burden for the software professional. It is not yet clear how effective the invariant-inference approach will be under a verified software paradigm for component-based software; when automated verification does not succeed, it will be critical for a human to understand these invariants in order to repair the code, the annotation or both. This means that inferred invariants should be not only technically correct but also comprehensible to the software professional, who will ultimately be responsible for at least reading and likely for modifying formal mathematical descriptions of software behavior. Some human input into writing invariants and other assertions therefore seems unavoidable.

This paper describes a modest advance down the special-syntax path: providing language constructs to reduce the annotation burden. It focuses on relationships between abstract invariant properties of individual variables that hold during an entire code segment and loop invariants within that segment. We observe that two kinds of properties must be included in a loop invariant to verify software. The first kind arise from the desire to treat a loop as a single statement in straight-line code for verification and reasoning purposes. These document the behavior of the loop by stating what it does not change; they are intimately tied to the loop and are local to it. The second kind arise from the need to maintain continuity of abstract invariants on variable values. These properties are often incidental to a particular loop yet are critical pieces of the loop invariant. For example, when using memoization to avoid re-computation of a function with a Java Map, one abstract invariant on the Map’s value is that if a key is defined then the value associated with that key is the function applied to the key. This information must be in the loop invariant for any loop involving the Map, because this property is true before the loop is encountered, is maintained by the loop, and might be intended to persist after the loop has terminated. This restricted set of Map values is known a priori by the software developer independently of any loops, and it can and should be documented. If the documentation is formal, its connection to the code can be verified. In other words, this documentation not only records the software developer’s reasoning but—in a verified software paradigm—also can be used to check that the reasoning is correct.

The contribution of this paper is a programming language construct, restrictions, that can be used to document abstract invariant properties of individual variables over segments of imperative code without introducing new programmatic types. This construct allows for reuse of these invariants. Moreover, it implicitly provides guidance to the
verifier by “factoring” potentially complicated verification conditions (VCs) into conceptually simpler VCs. The over-abundance of assumptions in VCs has been reported [9, 10] as a problem for back-end provers.

Restrictions are presented in the context of the RESOLVE programming language, a research language designed for verifiability. Specifically, RESOLVE has clean semantics, and provides syntactic slots for contracts and mathematical annotations of various kinds. However, the restriction construct should be adaptable to other programming languages with little change, so long as the specification language can express and ensure frame properties, such as JML [7] or Dafny [11].

The paper is structured as follows. Section 2 presents a simple motivating example (in C++ rather than RESOLVE). Section 3 includes a summary of the features and syntax of RESOLVE needed to explain restrictions. An introduction to restrictions in Section 4 features an in-depth example using sorting. Related work is discussed in Section 5, with conclusions in Section 6.

2. MOTIVATING EXAMPLE

Consider code that computes \( x^p \) where \( x \) is a double and \( p \) is a positive integer; see Figure 1(a). It computes \( x^p \) by first computing \( x^{2^k} \) where \( k \) is the largest natural number that satisfies \( 2^k \leq p \) and then making a recursive call to finish the job. In this particular implementation, \( q \) always equals \( 2^k \) where \( k' \) is some non-negative integer, and this property holds both as a loop invariant and, more generally, as an invariant on \( q \) throughout the code. We argue that this invariant can and should be documented.

One method a software professional can use to document the invariant on \( q \) is to add extra assertions in the code. At every line where the invariant holds, she asserts the invariant. Frame properties allow one to limit the number of such statements needed, by using them only after a modification to a variable under consideration. This documents the invariant on \( q \), but it is rather clumsy and the annotation burden is high. Restrictions (Section 4) are a construct to document the claims for this code more clearly and to reduce the annotation burden. Figure 1(b) shows what the code might look like in this situation. The loop invariant is simplified and the invariant on \( q \) is explicit.

3. RESOLVE OVERVIEW

As mentioned in the introduction, RESOLVE is an imperative and component-based research language designed for verifiability, performance and understandability [8]. The language has reference semantics with the following critical qualification that is enforced as a consequence of the language primitives: no aliasing of references across component boundaries is possible, i.e., there is no inter-component aliasing. Practically speaking, aliasing even within a single component implementation is rarely used except in implementations of a few low-level library components. This language restriction, a frame property, provides the illusion or effect of value semantics for any client usage (i.e., code written by users) of any component. Each component has a mathematical model of its behavior described in a contract, as illustrated in Figure 2 by a Queue contract. The mathematical model is explicit in the type declaration. Each operation has a formal description of behavior in terms of the mathematical model via standard requires and ensures clauses. The control return type is used within if/while conditions. One more restriction simplifies verifiability: all program function operations must behave as mathematical functions and must restore their arguments. This is superficially similar to the restriction that functions be “pure” in JML [7] or Dafny [11]. The difference is that RESOLVE program functions still may not be used in specifications because RESOLVE rigorously separates and distinguishes mathematical entities (including definitions of mathematical functions) from programming entities (including program operations, i.e., program functions, that happen to have functional behavior).

Behavioral extensions to abstract components, such as the Concatenate extension in Figure 2, are specified via con-
Figure 2: QueueTemplate contract and Concatenate extension

contracts and ordinarily implemented by layering on other components' contracts. Another extension of a Queue type is the operation Sort. Sorting has been studied by the computer science community since the field's inception; in the past few years there has been significant work on inferring loop invariants [2, 3, 4, 5, 6] among other work on verification of sorting algorithms. For the purposes of demonstrating the utility of restrictions, sorting therefore serves as an appropriate standard benchmark that naturally involves variables with abstract invariants beyond any of the generic abstract data type (ADT) invariants of its variables.

3.1 Sort Specification

Since the QueueTemplate component is generic, i.e., parameterized by a type Item, the contract of a Sort operation should also be generic. Figure 3 shows the requisite restriction on the ordering relation ARE_IN_ORDER to be used in sorting, namely that ARE_IN_ORDER is a total pre-order.

Figure 3 shows the mathematical definitions used to specify sorting, given ARE_IN_ORDER. OCCURS_COUNT is a mathematical function that returns the number of times a given Item appears in a string; it is used to construct the other mathematical definitions. This allows the contract to be specific about the value of the outgoing Queue: not only are the values of items in the outgoing Queue the same as in the incoming queue, but the number of times each appears is the same. IS_PRECEDING is a binary predicate and holds on two strings if and only if every item in the first string is related by ARE_IN_ORDER to every item in the second string; intuitively, every item in the first string is "no larger" than every item in the second. IS_NON_DECREASING is a unary predicate that is true if and only if every consecutive pair of Items in the string are related by ARE_IN_ORDER. Finally, IS_PERMUTATION is a binary predicate on strings that is true if and only if the number of occurrences of every Item is the same in the first string and in the second.

contract Sort (type Item)

definition ARE_IN_ORDER (x: Item, y: Item): boolean

  satisfies
  for all z: Item
    (ARE_IN_ORDER (x, y) or ARE_IN_ORDER (y, x)) and
    (if ARE_IN_ORDER (x, y) and ARE_IN_ORDER (y, z) then ARE_IN_ORDER (x, z)))

end Sort

Figure 3: Sort extension to QueueTemplate

The contract specification of a Sort operation is given in Figure 3. The Sort operation takes a Queue and returns with the property that the outgoing string q is a permuta-
tion of the incoming string and the outgoing string is non-decreasing with respect to ARE_IN_ORDER.

3.2 Quicksort Implementation

We present an implementation of the Sort operation using quicksort in Figure 4. Our implementation partitions a non-empty incoming queue into two queues (q and qBig) and a partitioning element (partitionElement) with the property that every item in q is in order with partitionElement and partitionElement is in order with every item in qBig. Each of the smaller queues is sorted recursively and q, partitionElement, and qBig are all concatenated to obtain the final, sorted queue. Besides the loop invariant and other mathematical annotations, this code is similar to code in most other languages. The loop invariant documents the insight of the algorithm, namely the ordering relationship among the variables partitionElement, q and qBig, as expressed formally via IS_PRECEDING and IS_PERMUTATION. The programmer's justification for termination is given by the decreases clause (i.e., progress metric).

A local operation Partition is used to split a queue according to the quicksort algorithm. The :=: operator is the "swap" operator [12, 13]. This exchanges the values of its two arguments, and is a key aspect of avoiding aliasing while preserving efficiency.

4. INTRODUCTION TO THE SYNTAX AND SEMANTICS OF RESTRICTIONS: SORTING EXAMPLE

First we examine the issues involved in defining of restrictions (the new programming language construct), and then the issues in the usage of restrictions in client code. Code presented in this section is analogous to the code in section 3.1 except it uses restrictions.

4.1 Restrictions

We create three restrictions for this example, one for each different abstract invariant maintained by specific uses of Queue in quicksort. The first invariant is that a Queue is sorted, i.e., it is an OrderedQueue. The other invariants relate the Items in a Queue to another Item. These invariants arise during the Partition implementation and simply relate qSmall to p and qBig to p by ARE_IN_ORDER; more specifically every item in qSmall is in order with p and p is in order with every item in qBig. For each operation that is called on any Queue that satisfies one of these properties, the programmer reasons that the abstract invariant is not broken by the operation call. The proof boils down to the question: if the operation is executed, does the new value of the variable still satisfy the restriction? With this intuition in mind, we show the contracts for the restrictions corresponding to these ideas in Figure 5.

A restriction is declared relative to one or more existing contracts, e.g., QueueTemplate. Operations of the underlying contract may be given additional requires and ensures. The restriction is given by a predicate where parameters are of the specified types. Since functions may not "break" the invariant—they cannot change the abstract value of any argument—they are always available to be used with a program type in any restriction.

realization QuickSort {
    function AreInOrder (restores i: Item, restores j: Item): control
        ensures ARE_IN_ORDER (i, j)
    )
    implements Sort for QueueTemplate
uses Concatenate for QueueTemplate

local procedure Partition (updates qSmall: Queue, replaces qBig: Queue, restores p: Item)
    ensures IS_PERMUTATION (qSmall + qBig, #qSmall)
    and IS_PRECEDING(<qSmall, <p>)
    and IS_PRECEDING(<p>, qBig)
variable tmp: Queue
loop
    updates qSmall, qBig, tmp
    maintains IS_PERMUTATION (qSmall + qBig + tmp, #qSmall + #qBig + #tmp)
    and IS_PRECEDING(tmp, <p>)
    and IS_PRECEDING(<p>, qBig)
    decreases |qSmall|
    while not IsEmpty (qSmall) do
        variable x: Item
        Dequeue (qSmall, x)
        if AreInOrder (x, p) then
            Enqueue (tmp, x)
        else
            Enqueue (qBig, x)
        end if
    end loop
    qSmall :=: tmp
end Partition

procedure Sort (updates q: Queue)
    decreases |q|
    if not IsEmpty (q) then
        variable partitionElement: Item
        variable qBig: Queue
        Dequeue (q, partitionElement)
        Partition (q, qBig, partitionElement)
        Sort (q)
        Sort (qBig)
        Enqueue (q, partitionElement)
        Concatenate (q, qBig)
    end if
end Sort
end QuickSort

Figure 4: Quicksort implementation of Sort extension to QueueTemplate
contract OrderedQueueTemplate
restricts QueueTemplate

restriction OrderedQueue(q : Queue)
  is (IS_NON_DECREASING(q))

procedure Enqueue(q : Queue, x : Item)
  under restriction OrderedQueue(q)
  also requires IS_PRECEDING (q.<<x>)

procedure Dequeue(q : Queue, x : Item)
  under restriction OrderedQueue(q)
  also ensures IS_PRECEDING (<x>, q)
end OrderedQueueTemplate

contract SmallValueQueueTemplate
restricts QueueTemplate

restriction SmallValueQueue(q : Queue, max : Item)
  is (IS_NON_DECREASING(q, <max>))

procedure Enqueue(updates q : Queue, clears x : Item)
  under restriction SmallValueQueue(q, max)
  also requires ARE_IN_ORDER(x, max)

procedure Dequeue(updates q : Queue, replaces x : Item)
  under restriction SmallValueQueue(q, max)
  also ensures ARE_IN_ORDER(x, max)
end SmallValueQueueTemplate

contract LargeValueQueueTemplate
restricts QueueTemplate

restriction LargeValueQueue(q : Queue, min : Item)
  is (IS_PRECEDING (q, <min>))

procedure Enqueue(updates q : Queue, clears x : Item)
  under restriction LargeValueQueue(q, min)
  also requires ARE_IN_ORDER(min, x)

procedure Dequeue(updates q : Queue, replaces x : Item)
  under restriction LargeValueQueue(q, min)
  also ensures ARE_IN_ORDER(min, x)
end LargeValueQueueTemplate

Figure 5: OrderedQueue, SmallValueQueue and LargeValueQueue restrictions

Conceptually, the also requires clauses are conjoined with the original requires clauses for the operation. These are used by the programmer to ensure that both the restriction is maintained by the operation, and to document conditions under which it is safe to call the operation while still maintaining the invariant. The also ensures clauses strengthen the previous postconditions. In the OrderedQueue contract, Dequeue’s also ensures clause gives information about how the dequeued item relates to items that remain in the OrderedQueue.

Since programmers may need some help in making sure that their reasoning process is correct, the compiler should generate VCs corresponding to the correctness of the restriction contract. The contract’s correctness condition is that if an operation is invoked in a state satisfying the variable restrictions and the requires clause, and the operation completes successfully, then the restriction is still satisfied by the updated variables; any also ensures clauses must also be satisfied. More concretely, each operation’s invocation can be assumed to occur in a state in which the restriction, the original requires clause, and the also requires clause hold. By a process similar to datatype induction, these VCs are generated just once for the contract. (Notice that this construction leaves the initialization of restrictions to a client-side activity and is discussed in Section 4.2.) The general form of the generated VCs, where s is a variable of the mathematical model of the restriction, args is the list of arguments to the operation, and ′ indicates a fresh variable, is given by:

\[
\text{restriction}(s') \land \text{requires}_{\text{original}}(s', \text{args}') \land \\
\text{requires}_{\text{also}}(s', \text{args'}) \land \\
\text{ensures}_{\text{original}}(s', \text{args}', s, \text{args}) \\
\implies \text{restriction}(s) \land \text{ensures}_{\text{also}}(s', \text{args}', s, \text{args})
\]

4.2 Client Usage of Restrictions

The updated Sort contract, shown in Figure 6, is almost the same as the original contract. The difference is that the Queue formal parameter q is restricted to satisfy the restriction OrderedQueue when the operation returns. The formal parameter q must be of type Queue; when Sort returns, q conforms to the restriction OrderedQueue (checked as a proof obligation). We can omit the IS_NON_DECREASING clause, since it is subsumed by the restriction.

contract Sort

procedure Sort(updates q : Queue)
  establishes restriction OrderedQueue(q)
  ensures IS_PERMUTATION (q, #q)
end Sort

Figure 6: Sort extension to QueueTemplate using restrictions

This reduces the mathematical annotation burden on the programmer. The restriction annotation in the formal parameters need not be checked by the static type system. It is equivalent to having the type restriction in the ensures clause for that variable. We examine this issue in more depth in the discussion of the Sort operation.

Figure 7 shows an additional operation Concatenate defined on Queues that is used by OrderedQueues. The also requires restriction slot is used in this contract to indicate
that two variables, \( q_1 \) and \( q_2 \) satisfy the \textit{OrderedQueue} restriction.

\begin{verbatim}
contract OrderedQueueConcatenate
restricts

procedure Concatenate(updates q1: Queue, clears q2: Queue)
under restriction
OrderedQueue(q1) and OrderedQueue(q2)
also requires
IS_PRECEDING(q1, q2)
end OrderedQueueConcatenate
\end{verbatim}

Figure 7: OrderedQueueConcatenate Restriction

The \textit{Partition} operation uses the \textit{SmallValueQueue} and \textit{LargeValueQueue} restrictions. The code for performing the partition operation is given in Figure 8. In the contract of \textit{Partition}, the \textit{ensures} clause and loop invariant are simplified by the use of the \textit{establishes restriction} annotation. Otherwise, the code is similar to the original version in Section 3.1.

Recall that in the contracts of restrictions, there were no VCs generated for initialization; that piece is left to the clients or users of the restriction. So, when a variable of a particular type, say \textit{Queue}, has a new restriction, say a \textit{LargeValueQueue}, a VC is generated to make sure that variable satisfies that restriction. For example, \textbf{confirm restriction} \textit{LargeValueQueue}(\textit{qBig}, \textit{p}) generates a VC whose goal is \textit{IS_PRECEDING}(\textit{q}, \textit{qBig}) and whose assumptions are those facts known at that point in the code, \textit{e.g.}, resulting from path conditions, loop invariants, and contracts of other operations called. We note that not only is the specification of \textit{Partition} simpler, but the loop invariant has been significantly simplified as well.

Figure 8 also shows the \textit{Sort} implementation using restrictions and the modified \textit{Partition} local operation. Except for the \textbf{confirm restriction} annotation, the code is exactly the same as the original version. The loop invariant is simplified as two conjuncts may be removed as the restrictions implicitly ensures the loop invariant.

Finally, to finish an earlier discussion about the implementation of the \textbf{expects restriction} or \textbf{establishes restriction} annotation in an operation parameter, one can implement the annotation by automatically translating it into a \textbf{requires} or \textbf{ensures} clause, respectively, in the operation contract. On every client use of the operation, the verification system adds a \textbf{confirm restriction} annotation after the call to reassert the restriction, generating one additional (simple) VC. This process can be invisible to the user, but simplifies the information needed for restrictions by avoiding carrying it across operation boundaries.

\begin{verbatim}
realization QuickSort {
...
uses OrderedQueueConcatenate for QueueTemplate

local procedure Partition (updates qSmall: Queue, replaces qBig: Queue, restores p: Item)
ensures IS_PERMUTATION (qSmall \# qBig, \#qSmall)
establishes restriction
LargeValueQueue(qBig, p) and SmallValueQueue(qSmall, p)

variable tmp: Queue
call Clear (qBig)
call Enqueue (qBig, x)
end Partition

procedure Sort (updates q: Queue)
decreases \#q

variable qtmp: Queue
qtmp :=: q
call confirm restriction OrderedQueue(q)
if not IsEmpty (qtmp) then
variable partitionElement: Item
variable qBig: Queue
Dequeue (qtmp, partitionElement)
Partition (qtmp, qBig, partitionElement)
Sort (qtmp)
Sort (qBig)
Enqueue (qtmp, partitionElement)
Concatenate (qtmp, qBig)
q :=: qtmp
end if
end Sort
end QuickSort
\end{verbatim}

Figure 8: Quicksort implementation of Sort using restrictions

\begin{verbatim}
\begin{align*}
\text{rules (L), theory-specific knowledge (M) or local mathematical definitions (D) are needed to prove a VC. VCs that use fewer assumptions or require less mathematical knowledge are considered less difficult. The metrics are summarized in Figure 9.}
\end{verbatim}

The code presented in Section 3 without restrictions was compared to the code in this section with restrictions. The original quicksort implementation’s most difficult VC was categorized as MH\(_5\), while the restrictions version has VCs of difficulty at most MH\(_3\). The MH\(_5\) VC is particularly difficult; it arises from proving the second conjunct in the
What is needed in the proof

<table>
<thead>
<tr>
<th>Label</th>
<th>What is needed in the proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Rules of mathematical logic</td>
</tr>
<tr>
<td>Hₙ</td>
<td>At most n hypotheses from the VC (n &gt; 0)</td>
</tr>
<tr>
<td>M</td>
<td>Knowledge of mathematical theories used in the specifications</td>
</tr>
<tr>
<td>D</td>
<td>Knowledge of programmer-supplied definitions based on mathematical theories above</td>
</tr>
</tbody>
</table>

(a) VC classification

Figure 9: VC classification and diagram of category relationships (adapted from [14])

ensures clause of Sort:

\[
\begin{align*}
& \text{is\_initial}(\text{partitionElement}_r) \\
& \land IS\_PERMUTATION(q_r, qBig, q_l) \\
& \land IS\_PRECEDING(q_l, (\text{partitionElement}_4)) \\
& \land IS\_PRECEDING((\text{partitionElement}_4), qBig) \\
& \land IS\_PERMUTATION(q_r, q_l) \\
& \land IS\_NON\_DECREASING(q_l) \\
& \land IS\_PERMUTATION(qBig, qBig) \\
& \land IS\_NON\_DECREASING(qBig) \\
& \land is\_initial(\text{partitionElement}_4) \\
& \land (\text{partitionElement}_4) * q_l \neq \Lambda \\
& \rightarrow IS\_NON\_DECREASING(q_l+ (\text{partitionElement}_4) * qBig)
\end{align*}
\]

The proof requires hypotheses 3 through 8, and is fairly involved; mathematical lemmas are needed, for instance, to conclude that hypotheses 3, 5 and 6 imply IS\_PRECEDING(q_l, (\text{partitionElement}_4)). The corresponding VCs from restrictions are easier. The direct analog of the above VC, in particular, is in category LH₁, i.e., the goal is one of the hypotheses. The proof of a VC arising from the call to Concatenate in the body of Sort is the most difficult: it is in the category MH₃.

Another VC in MH₃ arises from the also requires clause for Concatenate:

\[
\begin{align*}
& IS\_PRECEDING(q\_original, q\_original) \\
& \land IS\_NON\_DECREASING(q\_original) \\
& \land IS\_NON\_DECREASING(q\_original) \\
& \rightarrow IS\_NON\_DECREASING(q\_original * q\_original)
\end{align*}
\]

The one-time, reusable proof of this VC is also in MH₃. However, it is a relatively easy proof to discharge; it is an algebraic lemma of string theory. This is the essence of the proof of the original VC. Proving these VCs with Isabelle [15] using a version of RESOLVE’s string theory in an automatic mode [10] confirms that the MH₃ VC is hard to prove—Isabelle does not prove it automatically. The two MH₃ VCs are proved automatically. For this example, restrictions are able to simplify the code annotations and reduce the maximum difficulty of VCs generated from the resulting code.

We expect this empirical result to generalize: restrictions have the effect of adding “way-points” in the proofs of the VCs from code using restrictions. These way-points are created from input from the programmer; the requires and ensures clause are both modified to preserve the requisite invariant, thus ensuring that the way-point is useful for the justification of correctness. Moreover, the VCs generated from the declaration of a restriction should be syntactically simple with few assumptions and highly targeted—excellent candidates for proofs from general, reusable theorems.

We also expect that many restrictions will be reusable. For example, OrderedQueue may be used for any sorting algorithm implementation or client. Restrictions presented in this paper could be generalized to be usable in selection problems, e.g., via a predicate parameter to SmallValue-Queue. Even if we assume that restrictions turn out to not be reusable, there is still value in using them; restrictions document the reasoning behind why a particular block of code is correct, and, as such, aid readability by humans.

5. RELATED WORK

The idea of restrictions is similar to a core idea expressed in predicate subtypes, dependent types, refinement types and contract types [16, 17] (PDRCT). Depending on the exact setup of PDRCT used, proof obligations may be generated (such as Type Correctness Conditions (TCCs) in PVS) when converting from a type to a predicate subtype. In other setups the type checking system can infer many of the requisite properties. These ideas have been applied both to mathematical and programmatic domains. In any case, a restriction is different in that it entails modifying pre/post-conditions of operations to maintain the user-supplied invariant. Moreover, no new executable code need be emitted as a result of a restriction, which documents invariants and simplifies proofs of resulting VCs rather than defining a new type; a restriction is not a new type. However, the type inference and other algorithms used in contract and refinement types are largely absent; these could be added in the future using some of the existing work to alleviate some of the annotation burden on programmers.

The Jahob system [9] uses annotated Java source code as its source language. The annotation language has support for a proof language, with essentially full first-order prover functionality. There are first order proof commands, such as applying modus ponens, along with commands to perform local proofs. Invariants can be expressed as well. While Jahob’s proof language is powerful, the proof commands are not natural for a software professional. Rather than learning a proof system, software professionals using restrictions think in terms of contracts and component invariants, concepts that are used in the normal course of programming.
Behavioral subtyping [18] uses a set of rules to ensure that a subtype can always be used in place of a supertype without violating a behavioral property of the client program. Contractually, the preconditions of any subtype operation may not be strengthened, postconditions may not be weakened, and invariants must be preserved. Restrictions impose different requirements: in particular, preconditions may be strengthened. The goal of restrictions is not to allow for substitution, but rather to indicate that during specific code segments (i.e., not necessarily for the entire lifetimes of variables) stronger abstract invariants hold for specific variables.

Object invariants [19] are defined over the concrete representation of the object; they denote consistency or other properties that relate specific fields or ownership of a particular field or object. Restrictions instead are over the abstract state of the objects, their cover story as represented in mathematics, rather than over any particular representation of the object’s abstract state space. This feature ensures that restrictions can be used with any correct implementation of their underlying type, making them more reusable.

6. CONCLUSION

We have presented a programming language construct, restrictions, that helps address a limitation in current verification languages, namely the clumsiness of formally documenting client code, especially with loops. This construct, when applied to code similar to that shown in Section 4, provides a mechanism to separate out two uses of loop invariants, namely an abstraction of the behavior of a loop and a mechanism to maintain abstract invariants on variables. This approach not only can simplify VCs generated in client code, but also can result in reasoning reuse. This reuse happens both when restrictions are reused across clients, and even when there are multiple calls to a single restriction operation by a particular client.

7. ACKNOWLEDGMENTS

The authors are grateful for the constructive feedback from Bruce Adcock, Derek Bronish, Paolo Bucci, Harvey M. Friedman, Wayne Heym, Bill Ogden, Aditi Tagore, and Diego Zaccai. This material is based upon work supported by the National Science Foundation under Grants No. DMS-0701260, CCF-0811737, and ECCS-0931669. The authors would also like to thank the anonymous reviewers for their helpful suggestions. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES