Automatic verification of textbook programs

K. Rustan M. Leino [0] and Rosemary Monahan [1]

[0] Microsoft Research, Redmond, WA, USA
leino@microsoft.com
[1] National University of Ireland, Maynooth, Co.Kildare, Ireland
Rosemary.Monahan@nuim.ie

Manuscript KRML 175, 13 May 2007, DRAFT.

Abstract. Textbooks on program verification make use of simple programs in mathematical domains as illustrative examples. Mechanical verification tools can give students a quicker way to learn, because the feedback cycle can be reduced from days (waiting for hand-proofs to be graded by the teaching assistant) to seconds or minutes (waiting for the tool's output). However, the mathematical domains that are so familiar to students (for example, sum-comprehensions) are not directly supported by first-order SMT solvers.
This paper presents a technique for translating common comprehension expressions ($\mathbf{sum}$, $\mathbf{count}$, $\mathbf{product}$, $\mathbf{min}$, and $\mathbf{max}$) into verification conditions that can be tackled by a first-order SMT solver. The technique has been implemented in the Spec# program verifier. The paper also reports on the experience of using Spec# to verify several challenging programming examples drawn from a textbook by Dijkstra and Feijen.

0 Motivation

Computer science students are often introduced to program verification—thus learning about assertions, pre- and postconditions, and invariants—in a class setting where they conduct hand proofs of small programs. To let students focus on specifications and programming, the example programs often draw from the domain of familiar mathematics; for example, computing factorials or summing the elements of arrays. Just like parsers and type checkers are tools whose feedback help teach students about well-formed and well-typed programs, the feedback from verification tools can help teach students about preconditions and invariants, bridging the gap that otherwise exists between hand proofs and programming practice. But using verification tools in a class setting brings complications: the verification tool might be built on an interactive theorem prover, which puts an additional burden on the student to learn the commands and tactics of the prover, or on an automatic prover, whose theory domains might not be rich enough to include the mathematics that is so familiar to the student (like multiplication and sum-comprehensions).

In this paper, we present a technique for translating common comprehension expressions (like $\mathbf{sum}$, $\mathbf{count}$, $\mathbf{product}$, $\mathbf{min}$, and $\mathbf{max}$) into verification conditions that can be tackled by a first-order Satisfiability Modulo Theories (SMT) solver. We have implemented the technique in the Spec# [1] program verifier [0]. Using Simplify [4]

```
public static int SegSum(int[] a, int i, int j)
  requires 0 <= i && i <= j && j <= a.Length;
  ensures result == sum{int k in (i : j); a[k]};
{
  int s = 0;
  for (int n = i; n < j; n++)
    invariant i <= n && n <= j;
    invariant s == sum{int k in (i : n); a[k]};
  {
    s += a[n];
  }
  return s;
}
```

Fig. 0. Spec# method to sum the elements $a[i], a[i+1], \ldots, a[j-1]$. Here and throughout, our examples assume use of the Spec# compiler's `/nn` switch, which treats reference types as non-null reference types by default.

as the underlying SMT solver, we are able to verify several challenging programming examples from the Dijkstra and Feijen book *A Method of Programming* [5].

1 Some Textbook Examples: The Programmer's Perspective

In this section, we write some textbook examples to introduce the Spec# notation, focusing especially on comprehension expressions. In Spec#, every method has a specification outlining a contract between its callers and its implementations. The programmer writes each method and its specification together in a Spec# source file before running the verifier. The verifier is run like the compiler—either from the IDE or the command line. In either case, this involves just pushing a button, waiting, and then getting a list of compilation/verification error messages.

A sample Spec# method, $SegSum$, which sums the elements in a segment of an array, is presented in Fig. 0. Using the sum comprehension

$$\mathbf{sum}\{\mathbf{int}\ k\ \mathbf{in}\ (i : j);\ a[k]\} \tag{0}$$

where $k\ \mathbf{in}\ (i : j)$ expresses the range $i \leqslant k < j$, $SegSum$'s postcondition expresses the summation of the $j - i$ array elements starting with $a[i]$.

The general form of a comprehension in Spec# is

$$\mathcal{Q}\{K\ k\ \mathbf{in}\ E,\ F;\ T\}$$

where $\mathcal{Q}$ is **sum**, **count**, **product**, **min**, or **max** (or **forall**, **exists**, or **exists unique**, but those forms have counterparts in first-order logic, so we won't cover them in this paper), k is a bound variable of some type K, E is an enumerable expression that generates values of type K, the boolean expression F is a *filter* that further restricts the values of k under consideration (if omitted, F defaults to $true$), and the integer

(or for $\mathbf{count}$, boolean) expression T is the *term* of the comprehension. The bound variable k can occur free in F and T, but not in E.

The comprehension expression evaluates to the value obtained by applying operator associated with $\mathcal{Q}$ (for example, $+$ for $\mathbf{sum}$) to the expressions T that result for each of the prescribed values of k. To support the dynamic execution of comprehensions, Spec# insists on E being executable; currently, static verification is supported only for comprehensions where K is $\mathbf{int}$ and E is a half-open interval $(L : H)$, which means k satisfies $L \leqslant k < H$, and the closed (inclusive) interval $(L .. H)$, which means k satisfies $L \leqslant k \leqslant H$.

As an example of a filter expression, comprehension (0) can also be expressed as:

$$\mathbf{sum}\{\mathbf{int}\ k\ \mathbf{in}\ (0 : a.Length),\ i \leqslant k\ \&\&\ k < j;\ a[k]\} \tag{1}$$

To verify $SegSum$ example, it suffices to know the following mathematical properties about sum comprehensions:

empty range $(\forall\, lo, hi \bullet\ hi \leqslant lo\ \Rightarrow\ \mathbf{sum}\{\mathbf{int}\ k\ \mathbf{in}\ (lo : hi);\ a[k]\} = 0\ )$
induction $(\forall\, lo, hi \bullet\ lo \leqslant hi\ \Rightarrow$
$\mathbf{sum}\{\mathbf{int}\ k\ \mathbf{in}\ (lo : hi + 1);\ a[k]\}\ =\ \mathbf{sum}\{\mathbf{int}\ k\ \mathbf{in}\ (lo : hi);\ a[k]\} + a[hi]\ )$

As we explain in the next section, we have built in these and other properties as axioms in the program verifier.

Another classical example used to introduce students to program verification is the calculation of factorials. A method that calculates $n!$ can be specified as:

$\mathbf{requires}\ 0 \leqslant n;$
$\mathbf{ensures}\ \mathbf{result} == \mathbf{product}\{\mathbf{int}\ k\ \mathbf{in}\ (1\ ..\ n);\ k\};$

This specification lends itself to the obvious iterative implementation.

2 Encoding Comprehensions as First-Order Expressions

The Spec# static program verifier, named Boogie, translates compiled Spec# programs into the intermediate verification language BoogiePL, from which it then generates verification conditions for various SMT solvers [0]. The BoogiePL language includes functions and axioms, and its expressions include logical quantifiers and arithmetic. BoogiePL does not include direct support for any other comprehensions or binders, so the translation from Spec# into BoogiePL must instead use some suitable encoding. Such an encoding will necessarily be incomplete, but we hope to achieve an encoding that is good enough for use in practice.

The key idea in our translation is to introduce and axiomatize one BoogiePL function for each different *comprehension template* occurring in the Spec# program. In our explanation of what that means, we use the $SegSum$ example from the previous section as a running example. The sum comprehension (0) has bound variable k with range $(i : j)$, (implicit) filter $true$, and term $a[k]$. The BoogiePL translations of these expressions are i, j, $true$, and

$$ArrayGet(\$Heap[a, \$elements], k)$$

respectively. (To understand this translation, think of every array as being an object with one instance field, $\$elements$, whose value is a sequence of element values. The sequence is retrieved from the heap, which is modeled as a two-dimensional array indexed by object identities and field names, and the element value is then retrieved using the function $ArrayGet$.)

2.0 Comprehension Functions

Comprehensions supported by the Spec# program verifier have the form

$$\mathcal{Q}\{\mathbf{int}\ k\ \mathbf{in}\ (L : H),\ F;\ T\}$$

We consider the most general parameterization of the expressions F and T, extracting what we call the *template* of the comprehension. The template is a triple whose first component is $\mathcal{Q}$ and whose other two components are obtained by abstracting over the (largest) subexpressions of the filter and term that don't mention the bound variable. For example, the template of comprehension (0) is

$$(\mathbf{sum},\ \Box,\ ArrayGet(\Box, k))$$

Each "$\Box$" indicates a place where we have abstracted over a subexpression. Here and throughout this section, we assume the bound variable has some canonical name, and we'll simply use k. Note that the range expressions L and H are not part of the template. We write the the general form of a template as

$$(\mathcal{Q},\ Filter[\![\Box\Box, k]\!],\ Term[\![\Box\Box, k]\!]) \tag{2}$$

with the understanding that $Filter[\![\Box\Box, k]\!]$ and $Term[\![\Box\Box, k]\!]$ stand for expressions that can mention k and some number of $\Box$'s.

For each comprehension template, our translation introduces a function. We shall refer to it as a *comprehension function* and give it a name like $\mathcal{Q}\#n$ where n is some unique sequence number. For example, sum comprehension (0) in program $SegSum$ gives rise to the following comprehension function in our translation into BoogiePL:

$$\mathbf{function}\ sum\#0(lo\colon \mathbf{int}, hi\colon \mathbf{int}, a0\colon \mathbf{bool}, a1\colon Elements)\ \mathbf{returns}\ (\mathbf{int});$$

The comprehension function takes as arguments the range (expressed as the end points of a half-open interval), as well as one argument for each "hole" $\Box$ in the template. Intuitively, for a comprehension template (2), the comprehension function has the following meaning:

$$\mathcal{Q}\#n(lo, hi, aa) \ = \ \mathcal{Q}_{k\in(lo:hi)\ \text{such that}\ Filter[\![aa,k]\!]}\ Term[\![aa, k]\!]$$

For example, comprehension function $sum\#0$ above has the meaning:

$$sum\#0(lo, hi, a0, a1) \ = \ \sum_{k\in(lo:hi)\ \text{such that}\ a0} ArrayGet(a1, k)$$

Using comprehension function $sum\#0$, the sum comprehension (0) translates into BoogiePL as

$$sum\#0(i, j, true, \$Heap[a, \$elements])$$

Notice how the filter and term of the template are part of the intuitive meaning of $sum\#0$, and how the subexpressions that were abstracted over in the template find themselves as arguments in the translation of a particular sum comprehension.

As another example, the sum comprehension (1) with a filter has the following template:

$$(\mathbf{sum},\ \Box \leqslant k \ \wedge\ k \leqslant \Box,\ ArrayGet(\Box, k))$$

Thus, if both it and the comprehension in Fig. 0 were present in the same program, they would give rise to two different comprehension functions, like $sum\#0$ and $sum\#1$.

2.1 Matching Triggers

For each comprehension function, our translation also generates a number of axioms. To obtain the desired effect of these axioms in the Simplify SMT solver, it is crucial to indicate appropriate *matching triggers* for the quantifiers [4]. A matching trigger of a universal quantifier is set of expressions that determine how the SMT solver instantiates the quantifier. Logically, it is correct to instantiate a universal quantifier with anything at all, but since most instantiations are irrelevant to the verification goal, one can hope for a more fruitful search by limiting which instantiations the SMT solver is allowed to consider. When the SMT solver's search heuristics determine that it is time to look at quantifiers, the solver's ground terms (typically stored in an *e-graph* data structure that tracks equivalence classes of terms [6]) are compared against the triggers of the active quantifiers. Ground terms that match the triggers are used to instantiate the quantifiers.

Note that a universal quantifier that appears in a negative position in an axiom is really an existential quantifier. The SMT solver always Skolemizes existential quantifiers, so we need not worry about triggers for them.

Let us give some simple examples that demonstrate how triggers are employed. Using BoogiePL syntax, which encloses triggers in curly braces, the quantifier

$$(\,\forall\, x\colon \mathbf{int}, y\colon \mathbf{int} \bullet\ \{g(x,y)\}\ f(x) < y\ \Rightarrow\ g(x,y) = 100\,)$$

says that it is to be instantiated with terms x and y that appear in the e-graph as arguments to the function g. In order to be discriminating, a trigger must mention all bound variables and cannot mention a bound variable by itself. For example, $\{f(x)\}$ is not a legal trigger for the quantifier above, because it doesn't limit the terms that can be used to instantiate y, and likewise for $\{f(x),\ y\}$.

Since matching is done in the e-graph in Simplify, the congruence closure of all known terms is taken into consideration. Stated differently, matching is done within the theory of uninterpreted function symbols and equality (EUF). But other theories are not taken into consideration. For example,

$$(\,\forall\, x\colon \mathbf{int} \bullet\ \{g(x+1)\}\ h(x) = g(x+1)\,)$$

would not match against either the term $g(2+y-1)$ or the term $g(1+y)$, because the equalities of $2+y-1$ and $y+1$, and of $1+y$ and $y+1$, are facts known to the decision procedure for the theory of linear arithmetic but may never be propagated

into the e-graph. In this way, using interpreted functions like $+$ in a trigger makes the trigger *fragile*.

Some triggers are not limiting enough. For example,

$$(\forall\, x\colon \mathbf{int} \bullet \{h(x)\}\ \ h(x) < h(k(x))\ )$$

matches any argument of h, but when the quantifier is instantiated, the instantiation produces a term with another argument of h. Hence, if $h(X)$ occurs in the e-graph, then this quantifier will be instantiated with X, $k(X)$, $k(k(X))$, ..., causing a *matching loop*. A more limiting trigger for this quantifier is $\{h(k(x))\}$, which does not cause a matching loop.

2.2 Axioms

Back to our comprehensions. We show our axioms for sum comprehensions; the others are similar. The first axiom we provide is the **unit** axiom, which we render as follows:

$$\begin{array}{l}(\forall\, lo\colon \mathbf{int}, hi\colon \mathbf{int}, aa\colon T \bullet \{sum\#n(lo, hi, aa)\}\\ \quad (\forall\, k \bullet\ lo \leqslant k \wedge k < hi\ \Rightarrow\ \neg Filter[\![aa, k]\!]\ )\\ \quad \Rightarrow\ sum\#n(lo, hi, aa) = 0\ )\end{array}$$

where $Filter[\![aa, k]\!]$ (and $Term[\![aa, k]\!]$ below) stands for the filter (and term, respectively) expression in the template for $sum\#n$. Note that the **empty range** property in the previous section is a special case of the **unit** axiom. The trigger says for the outer quantifier to be instantiated for every occurrence of $sum\#n$ in the e-graph. The inner quantifier appears in a negative position in the axiom, so we need not worry about triggers for it.

The next axiom is the **induction** axiom, which has two parts. The first part is:

$$\begin{array}{l}(\forall\, lo\colon \mathbf{int}, hi\colon \mathbf{int}, aa\colon T \bullet \{sum\#n(lo, hi + 1, aa)\}\\ \quad lo \leqslant hi \wedge Filter[\![aa, hi]\!]\\ \quad \Rightarrow\ sum\#n(lo, hi + 1, aa) = sum\#n(lo, hi, aa) + Term[\![aa, hi]\!]\ )\end{array}$$

The second part is the same, except that $Filter[\![aa, k]\!]$ is negated and "$+\, Term[\![aa, k]\!]$" is dropped. We avoid the possible trigger $\{sum\#n(lo, hi, aa)\}$, since it causes a matching loop. Instead, we use the fragile trigger that contains a "$+1$". The "$+1$" in the trigger means that the quantifier is instantiated only when the e-graph already considers a sum over the range $(lo : hi + 1)$, in which case the axiom is likely to be fruitful in the verification, but the fact that the trigger is fragile means the quantifier may not be instantiated as often as required by the verification.

The third axiom is the **split range** axiom:

$$\begin{array}{l}(\forall\, lo\colon \mathbf{int}, mid\colon \mathbf{int}, hi\colon \mathbf{int}, aa\colon T \bullet \{sum\#n(lo, mid, aa), sum\#n(mid, hi, aa)\}\\ \quad lo \leqslant mid \wedge mid \leqslant hi\\ \quad \Rightarrow\ sum\#n(lo, mid, aa) + sum\#n(mid, hi, aa) = sum\#n(lo, hi, aa)\ )\end{array}$$

For this axiom, we choose a trigger with two terms—both need to match in order for the quantifier to be instantiated. If the e-graph contains the comprehension function

with two consecutive ranges, thus matching the trigger, then the instantiation is also likely to be fruitful.

The final axiom is the **same term** axiom:

$$\begin{array}{l}(\forall\, lo\colon \mathbf{int}, hi\colon \mathbf{int}, aa\colon T, bb\colon T \bullet \{sum\#n(lo, hi, aa), sum\#n(lo, hi, bb)\} \\ \quad (\forall\, k\colon \mathbf{int} \bullet\ lo \leqslant k \wedge k < hi \Rightarrow \\ \qquad\qquad (Filter[\![aa, k]\!] \equiv Filter[\![bb, k]\!]) \wedge \\ \qquad\qquad (Filter[\![aa, k]\!] \Rightarrow Term[\![aa, k]\!] = Term[\![bb, k]\!])\ ) \\ \quad \Rightarrow\ sum\#n(lo, hi, aa) = sum\#n(lo, hi, bb)\ )\end{array}$$

This axiom is the only one that relates two comprehension-function applications with different arguments for the template "holes"; it says the two function applications are equal if the filters agree in the range $(lo : hi)$ and, whenever the filters hold for a k in that range, the terms for k are equal. We specify the trigger (the obvious one) for the outer quantifier; the inner quantifier appears in a negative position in the axiom, so we need not worry about a trigger for it.

2.3 Adequacy of the Axiomatization

We make four remarks about the adequacy of our axiomatization.

First, notice that all axioms concern just one comprehension function: there is no axiom that relates two different comprehension functions. For example, since sum comprehension (0) has a different template than sum comprehension (1), they give rise to different comprehension functions. Thus, if the sum comprehension in the loop invariant in the $SegSum$ method were changed to the form (1) that uses the filter, then the verification would not be able to establish the postcondition (which is written in the form (0)) after the loop. This was not a problem in any of the textbook examples we looked at, because their loop invariants and postconditions are written in the same style.

Second, using Simplify as the SMT solver, we have not experienced any problems with the fragile trigger of the **induction** axiom.

Third, an alternative trigger for the **split range** axiom is

$$\{sum\#n(lo, mid, aa), sum\#n(lo, hi, aa)\}$$

As we'll explain in the next section, we have found this to be useful as an additional trigger (a quantifier can be given more than one trigger, with the effect that it is instantiated if any one of the triggers yields a match). However, the two triggers actually give rise to too many instantiations, to the point where it has a noticeably bad impact on performance. Therefore, we have made the inclusion of this second trigger selectable by Boogie's the `/summationStrength` command-line switch.

Fourth, trigger issues aside, the collection of axioms we have provided seems plausibly adequate in that ranges of size 0 or 1 can be addressed by the **unit** and **induction** axioms, and all larger ranges can be addressed by decomposing them into smaller ranges with the **split range** axiom. For example, we did not include an induction axiom that enlarges the range at the lower end, as in $(lo - 1 : hi)$, but the effect of that axiom can be achieved by first reasoning about the ranges $(lo - 1 : (lo - 1) + 1)$ and $((lo - 1) + 1 : hi)$ and then using the **split range** axiom.

```
public static int ReverseSum(int[] a)
  ensures result == sum{int k in (0 : a.Length); a[k]};
{
  int s = 0;
  for (int n = a.Length; 0 < n; )
    invariant 0 ⩽ n && n ⩽ a.Length;
    invariant s == sum{int k in (n : a.Length); a[k]};
  {
    n--;
    assert a[n] == sum{int k in (n : n + 1); a[k]};
    s += a[n];
  }
  return s;
}
```

Fig. 1. Spec# method that sums array elements in reverse order.

However, triggers are an issue. If triggers do not work out automatically, it is possible, as an advanced feature, to introduce expressions in the Spec# source code that will trigger instantiations of axioms. For example, the fact that the assert statement in Fig. 1 mentions the sum comprehension over range $(n : n+1)$ is enough to trigger the appropriate axioms that make method $ReverseSum$ verify.

3 Some More Difficult Examples

We now report on our experience of using Spec# to verify some more challenging examples: namely the **Minimal Segment Sum** and the **Coincidence Count** as described by Dijkstra and Feijen [5].

The minimal segment sum of a given integer array a is the minimum of all segment sums, calculated for all segments $a[i], a[i+1], \ldots, a[j-1]$ where $0 \leqslant i \leqslant j \leqslant a.Length$. We present the problem's specification, together with its solution, in Fig. 2. The main verification problems are due to the nesting of comprehensions in the program invariant. In particular, the verification of invariant (I2) requires induction on both the inner and outer comprehensions. Axioms to support this and the distribution of quantifiers are discussed in Section 2.

The coincidence count of two given integer arrays, each of which is arranged in strict increasing order, is the number of values occurring in both arrays. We specify this problem and give a first solution of it in Fig. 3.

A more efficient solution is achieved if we do not insist on reaching the end of both arrays, *i.e.*, the loop guard is written as a conjunction: $m < f.Length \;\&\&\; n < g.Length$. We have also verified this version. Doing so requires some assert statements at the end of the method, helping the verifier understand that any remaining array elements have no effect on the coincidence count, or using the `/summationStrength` command-line switch to obtain the additional trigger for the **split range** axiom.

```
public static int MinSegmentSum(int[] a)
  ensures result == min{int j in (0 .. a.Length); min{int i in (0 .. j);
                          sum{int k in (i : j); a[k] }}};
{
  int x = 0; int y = 0;
  for (int n = 0; n < a.Length; n++)
    invariant 0 <= n && n <= a.Length;                                    (I0)
    invariant x == min{int j in (0 .. n); min{int i in (0 .. j);
                          sum{int k in (i : j); a[k] }}};                 (I1)
    invariant y == min{int i in (0 .. n); sum{int k in (i : n); a[k] }};  (I2)
  {
    y += a[n];
    if (0 <= y) { y = 0; } else if (y < x) { x = y; }
  }
  return x;
}
```

Fig. 2. Spec# specification and solution of the Minimal Segment Sum problem.

4 Evaluation

Many of the difficulties met during our program verifications were in trying to diagnose error messages. Error messages need to be made more descriptive, particularly for use in a learning environment. Much of the confusion comes from uncertainty about how to proceed when an error is found; do we rewrite the specification, correct the program or do we need to assist the verifier by adding **assert** or **assume** statements?

Debugging by adding assertions at the Spec# level requires an understanding of the verification process and the methodology employed by the program verifier.

The overall performance of the enhanced system programming system is usually acceptable. *SegSum* and *ReverseSum* each takes 0.2 seconds to verify, *MinSegmentSum* takes 92 seconds, and the first version of *CoincidenceCount* takes 2.8 seconds. Performance is not acceptable when using the `/summationStrength` command-line switch that uses the additional trigger for the **split range** axiom. This axiom, while it eliminated the need for many assertions in the coincidence count example, slowed some other verifications down considerably. For example, the minimal segment sum solution requires 449 seconds to verify with the additional trigger.

5 Related Work

To our knowledge, our encoding of comprehensions into first-order expressions is the first technique for supporting automatic program verification using an off-the-shelf SMT solver. However, using a custom theorem prover, the automatic specification and verification environment Perfect Developer [3, 2] also provides support for comprehensions like the ones we have considered here. In some ways, Perfect Developer provides more flexible support (allowing programmers to define their own operators that apply to

```
public static int CoincidenceCount(int[] f, int[] g)
  requires forall{int i in (0 : f.Length), int j in (0 : f.Length), i < j; f[i] < f[j]};
  requires forall{int i in (0 : g.Length), int j in (0 : g.Length), i < j; g[i] < g[j]};
  ensures result == count{int i in (0 : f.Length), int j in (0 : g.Length); f[i] == g[j]};
{
  int ct = 0; int m = 0; int n = 0;
  while (m < f.Length || n < g.Length)
    invariant 0 <= m && m <= f.Length;                                    (I0)
    invariant 0 <= n && n <= g.Length;                                    (I1)
    invariant ct == count{int i in (0 : m), int j in (0 : n); f[i] == g[j]}; (I2)
    invariant m == f.Length || forall{int j in (0 : n); g[j] < f[m]};     (I3)
    invariant n == g.Length || forall{int i in (0 : m); f[i] < g[n]};     (I4)
  {
    if (n == g.Length || (m < f.Length && f[m] < g[n])) {
      m++;
    } else if (m == f.Length || (n < g.Length && g[n] < f[m])) {
      n++;
    } else {  // g[n] == f[m]
      ct++; m++; n++;
    }
  }
  return ct;
}
```

Fig. 3. A first solution to the Coincidence Count problem. Giving multiple binders for a comprehension is a shorthand for nesting multiple comprehensions; for a **count** comprehension with multiple binders, the innermost comprehension remains a **count** whereas the enclosing ones are **sum** comprehensions.

sequences, sets, and multisets), whereas in other ways, we provide more flexible support (directly allowing comprehensions to apply to arbitrary terms, not just the elements of sequences, and supporting things like subsequences and reverse summation). We hope to learn how to combine the techniques of the two tools.

6 Conclusions and Future Work

We have implemented support for summation-like comprehensions in an automatic program verifier, which takes us a step closer to providing tool support for students learning program verification. Our axiomatization is of a modest size, and we have found our approach to work fairly well, even on some challenging textbook examples. However, more work is needed, especially in the area of explaining error messages to users.

To further support students in verification, we would like to develop a larger repertoire of verified textbook programs. We would like to include in it programs that use common mathematical data structures like sets, multisets, maps, and sequences, as well as common support for abstraction like model variables.

Acknowledgments We thank the participants of the IFIP WG 2.3 meeting in Sydney, January 2007, for serving as a springboard for the initial ideas.

References

0. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.
1. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
2. Gareth Carter, Rosemary Monahan, and Joseph M. Morris. Software refinement with Perfect Developer. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 363–373. IEEE Computer Society, September 2005.
3. David Crocker and Judith Carlton. A high productivity tool for formally verified software development. Technical report, Escher Technologies, September 2004. http://www.eschertech.com/papers/pdpaper.pdf.
4. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
5. Edsger W. Dijkstra and W. H. J. Feijen. *A method of programming*. Addison-Wesley, July 1988.
6. Charles Gregory Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC, June 1981. The author's PhD thesis.