Automatic verification of textbook programs that use comprehensions

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, 30 June 2007.

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 two first-order SMT solvers. 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. However, 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

```
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.

have implemented the technique in the Spec# [1] program verifier [0]. Using a choice of Simplify [5] or Z3 [4] as the underlying SMT solver, we are able to verify the partial correctness of several challenging programming examples from the Dijkstra and Feijen book *A Method of Programming* [6].

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, if they exist.

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

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

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 these 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 the 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 $(L \mathinner{..} H)$, 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 the $SegSum$ 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 included 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:

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

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 axiomatise 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 modelled 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 parameterisation 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 do not mention the bound variable. For example, the template of comprehension (0) is

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

Each "$\square$" 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 general form of a template as

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

with the understanding that $\mathit{Filter}[\![\square ... \square, k]\!]$ and $\mathit{Term}[\![\square ... \square, k]\!]$ stand for expressions that can mention k and some number of $\square$'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" $\square$ in the template. Intuitively, for a comprehension template (2), the comprehension function has the following meaning:

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

where aa corresponds to as many arguments as there are $\square$'s in the template.

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:

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

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 $sum\#0$ above and $sum\#1$:

$$\mathbf{function}\ sum\#1(lo\colon \mathbf{int}, hi\colon \mathbf{int}, a0\colon \mathbf{int}, a1\colon \mathbf{int}, a2\colon Elements)\ \mathbf{returns}\ (\mathbf{int});$$

Function $sum\#1$ then has the intuitive meaning

$$sum\#1(lo, hi, a0, a1, a2) \;=\; \sum_{k \in (lo:hi) \text{ such that } a0 \leqslant k \wedge k < a1} ArrayGet(a2, k)$$

and comprehension (1) translates into BoogiePL as

$$sum\#1(0, \$ArrayLength(a), i, j, \$Heap[a, \$elements])$$

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 and Z3 SMT solvers, it is crucial to indicate appropriate *matching triggers* for the quantifiers [5]. A matching trigger of a universal quantifier is a set of expressions that determines 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 [8]) 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 $\{f(x),\ y\}$.

Typically, the terms mentioned in triggers also appear in the body of the quantifier, but this is not a requirement. For example, $\{h(x,y)\}$ is a legal trigger for the quantifier above.

Since matching is done in the e-graph in Simplify and Z3, 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*. The interpreted functions of interest in this paper are $+$ and $-$. Simplify enters given expressions that mention $+$ and $-$ into the e-graph (as well as passing them onto the arithmetic theory, which interprets the symbols), which means they are available in matching, but with no regard to their interpretation. In Z3, the interpreted symbols $+$ and $-$ are not entered into the e-graph, so triggers that mention $+$ and $-$ will never give rise to any matches.

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.

For every comprehension template, our encoding introduces not one, but two function symbols, $sum\#n$ and $s\#n$. We axiomatize these to be synonyms of each other:

$$\begin{array}{l}(\forall\, lo\colon \mathbf{int}, hi\colon \mathbf{int}, aa\colon T \bullet\ \{sum\#n(lo,hi,aa)\} \\ \qquad sum\#n(lo,hi,aa) = s\#n(lo,hi,aa)\ )\end{array}$$

Each sum comprehension in the Spec# program turns into a term that uses $sum\#n$, as we showed earlier in this section. For all axioms below, we use $s\#n$ in all quantifier bodies, but we sometimes use $sum\#n$ instead of $s\#n$ in quantifier triggers. The effect of this encoding is that we can limit certain instantiations to avoid matching loops: since the bodies of axioms only mention $s\#n$, instantiations will not give rise to any new $sum\#n$ terms. Note that the **synonym** axiom above uses $sum\#n$ in its trigger (shown in curly braces), not $s\#n$; thus, for each $sum\#n$ term in the input, the SMT solver will generate an equivalent $s\#n$ term, but not vice versa.

We provide a **unit** axiom, which we render as follows:

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

where $Filter[\![aa,k]\!]$ (and $Term[\![aa,k]\!]$ below) stands for the filter (and term, respectively) expression in the template for the $\mathbf{sum}$ comprehension. 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 $s\#n$ in the e-graph. The inner quantifier appears in a negative position in the axiom, so we need not worry about triggers for it.

It is important to be able to reason inductively about comprehensions, but induction axioms are susceptible to matching loops. To avoid matching loops, we limit each $sum\#n$ expression in the input to one instantiation of each induction axiom, which we achieve by mentioning $sum\#n$, not $s\#n$, in the triggers. We provide four induction axioms altogether. The **induction below** axioms relate $s\#n(lo,hi,aa)$ and $s\#n(lo+1,hi,aa)$:

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

and the **induction above** axioms relate $s\#n(lo,hi,aa)$ and $s\#n(lo,hi-1,aa)$:

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

Another way to avoid matching loops would be to use $\{s\#n(lo+1,hi,aa)\}$ as the trigger for the **induction below** axioms and $\{s\#n(lo,hi-1,aa)\}$ as the trigger for the **induction above** axioms; however, these triggers are fragile, because they mention the interpreted symbols $+$ and $-$, so they are of limited use with Simplify and of no use with Z3. Our synonym encoding, on the other hand, works with both Simplify and Z3.

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

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

Several remarks about the triggers are in order. First, each trigger mentions two terms, because there is no single term that covers all bound variables. Second, we give two triggers; a match of either one gives rise to an instantiation of the quantifier. From the point of view of symmetry, the possible trigger

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

is conspicuously absent. We omitted this trigger, because it had a dramatically adverse impact on performance (for the larger examples we report on in Section 4, including this trigger slowed down the verifications by as much as a factor of 35 with both Simplify and Z3). Third, the triggers use $sum\#n$, despite the fact that using $s\#n$ would not lead to any matching loop here (repeated instantiations will eventually lead to quiescence, because the set of terms used among the first two arguments to $s\#n$ is not increased). However, using $s\#n$ had a bad impact on performance (by as much as a factor of 10 for our examples).

We also generate a **same terms** axiom:

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

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. The inner quantifier appears in a negative position in the axiom, so we need not worry about a trigger for it. For the outer quantifier, we could have chosen the trigger

$$\{s\#n(lo, hi, aa), s\#n(lo, hi, bb)\}$$

without running the risk of matching loops, since instantiating the quantifier would not give rise to any $s\#n$ terms that are not already required by this trigger. However, the trigger with two $s\#n$ terms gave rise to unacceptable performance, so we chose to use $sum\#n$ in one of the terms. We also tried specifying both terms in the trigger with $sum\#n$, but that was too restrictive for our example programs, which sometimes need this axiom to be applied to terms generated by the inductive axioms.

Finally, exclusively for $\mathbf{min}$ and $\mathbf{max}$ comprehensions, we generate one more axiom, the **distribution (of plus over min/max)** axiom (here shown for $\mathbf{min}$, using functions $min\#n$ and $m\#n$):

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

Several remarks are in order. First, for nonempty ranges, this axiom generalizes the **same terms** axiom (with 0 for D). Second, the nested universal quantifier appears in a negative position, so we need not worry about a trigger for it, but the trigger for the existential quantifier matters. What makes a good trigger for it depends on the comprehension template. Therefore, we specify no trigger, which puts us at the mercy of the SMT solver's heuristics to select a trigger from the body of the quantifier. Third, given the nested universal quantifier, the conjunct

$$Term[\![aa, k]\!] + D = Term[\![bb, k]\!]$$

in the body of the existential quantifier follows from the other conjuncts. However, we include it to give the SMT solver's heuristics a better chance of finding some trigger. Fourth, in the case where $Filter[\![aa, k]\!]$ does not actually depend on k (which happens in the common case where the comprehension uses no filter at all), we replace the existential quantifier by

$$lo < hi \wedge Filter[\![aa, k]\!]$$

Fifth, the trigger of the outer quantifier is problematic. It mentions $+$ and is therefore fragile. For our examples, this fragility does not cause a problem for Simplify, but it renders the axiom useless for Z3.

2.3 Adequacy of the Axiomatisation

We make a few remarks about the adequacy of our axiomatisation.

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. Although some verifications could benefit from axioms that relate different comprehension functions, this was not necessary for any of the textbook examples that we looked at. This is because their loop invariants and postconditions are written in the same style. We recommend that when students write specifications, this similarity between loop invariants and postconditions is maintained.

Second, our use of $sum\#n$ instead of $s\#n$ in some triggers limits the number of quantifier instantiations. However, the instantiations are adequate for the examples we tried. Also, using Simplify as the SMT solver, we have not experienced any problems with the fragile trigger of the **distribution** axiom. The lack of the **distribution** axiom for Z3 means that it cannot verify examples like Minimal Segment Sum.

Third, 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 above** axioms, and all larger ranges can be addressed by decomposing them into smaller ranges with the **split range** axiom. For example, it is not necessary to include the **induction below** axiom that enlarges the range at the lower end, as the effect of that axiom can be achieved by first reasoning about the ranges $(lo : lo + 1)$ and $(lo + 1 : hi)$ and then using the **split range** axiom.

However, triggers are an issue. Omitting the **induction below** axiom from our axiomatisation prevents the verification of programs like $Sum2$ from Fig. 2. This program could be verified using the **induction above** and **split range** axioms as just described, but the needed axiom applications are not triggered automatically. In cases like this, it is possible, as an advanced feature, to introduce expressions in the Spec# source code that will trigger the instantiation of axioms. For example, adding the assert statement $\mathbf{assert}\ a[n] == \mathbf{sum}\{\mathbf{int}\ k\ \mathbf{in}\ (n : n + 1);\ a[k]\};$ before modifying s would be enough to make $Sum2$ verify even without the **induction below** axiom. Simply mentioning sum comprehension over the range $(n : n+1)$ acts as a prover directive causing the appropriate axiom to be instantiated. However, this is not a solution that we recommend, since adding such prover directives puts a much higher burden on the specifier.

3 Some More Difficult Examples

We now report on our experience of using Spec# to verify some more challenging examples, including some programming problems described in a textbook by Dijkstra and Feijen [6]. We begin with an example that illustrates the use of alternative loop invariants.

```
 public static int $Sum0$(int[] $a$)
   ensures result == sum{int $i$ in $(0 : a.Length)$; $a[i]$};
 {
   int $s = 0$;
   for (int $n = 0$; $n < a.Length$; $n$++)
     invariant $n \leqslant a.Length$ && $s$ == sum{int $i$ in $(0 : n)$; $a[i]$};
   {
     $s$ += $a[n]$;
   }
   return $s$;
 }
public static int $Sum1$(int[] $a$)
  ensures result == sum{int $i$ in $(0 : a.Length)$; $a[i]$};
{
  int $s = 0$;
  for (int $n = 0$; $n < a.Length$; $n$++)
    invariant $n \leqslant a.Length$ &&
      $s$ + sum{int $i$ in $(n : a.Length)$; $a[i]$} == sum{int $i$ in $(0 : a.Length)$; $a[i]$};
  {
    $s$ += $a[n]$;
  }
  return $s$;
}
```

Fig. 1. Two programs that sum an array's elements starting from its first element. The programs are identical, except that they use different loop invariants. Whereas, $Sum0$ uses a loop invariant that focuses on what has been summed so far, $Sum1$ uses a loop invariant that focuses on what is yet to be summed. (The interval analysis performed by the Spec# program verifier infers the invariant $0 \leqslant n$ automatically.) Our **induction above** axiom allows the verification of both programs.

3.0 Variations of summing

There are two main ways that a loop can iterate over a number of items to compute a property expressed by a comprehension, namely forward and backward. And for each of these ways, there are two main ways to write the associated loop invariant, either describing what has been computed so far or what is yet to be computed. Figure 1 and 2 show these four variations for summing the elements of an array.

The verification of these four programs collectively make use of both the **induction below** and **induction above** axioms, and trigger these with different terms. Our verifier verifies all of these programs, in a fraction of a second, as seen in the performance figures in Fig. 7.

3.1 Coincidence count

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. This problem is

```
 public static int Sum2(int[] a)
    ensures result == sum{int i in (0 : a.Length); a[i]};
 {
    int s = 0;
    for (int n = a.Length; 0 ⩽ --n; )
       invariant 0 ⩽ n && n ⩽ a.Length &&
          s == sum{int i in (n : a.Length); a[i]};
    {
       s += a[n];
    }
    return s;
 }
public static int Sum3(int[] a)
   ensures result == sum{int i in (0 : a.Length); a[i]};
{
   int s = 0;
   for (int n = a.Length; 0 ⩽ --n; )
      invariant 0 ⩽ n && n ⩽ a.Length &&
         s + sum{int i in (0 : n); a[i]} == sum{int i in (0 : a.Length); a[i]};
   {
      s += a[n];
   }
   return s;
}
```

Fig. 2. Two programs that sum an array's elements starting from its last element. The programs are identical, except that they use different loop invariants. Whereas, $Sum2$ uses a loop invariant that focuses on what has been summed so far, $Sum3$ uses a loop invariant that focuses on what is yet to be summed. Our **induction below** axiom allows the verification of both programs.

included in the book of Dijkstra and Feijen [6]. The specification of the problem uses a sum comprehension nested inside two minimum comprehensions.

We show one solution to this problem in Fig. 3. Although this a solution that some students might write, it is not the nicest solution to the problem. First, it iterates until both arrays have been exhausted, despite the fact that all coincidences have been found by the time that one array has been exhausted. Second, this has an effect on the guards of the if statement in the program, which need to consider the possibility of either array having been exhausted. The program is correct, however, and it passes our verifier.

A nicer solution to the problem is shown in Fig. 4. This solution is more efficient as it stops iterating when either array has been exhausted. This, in turn, makes the if statement guards less complicated. The main issue is triggering the instantiation of the **split range** axiom. The inclusion of the second trigger for the **split range** axiom gets used here, and the program is automatically verified.

The programs in Fig. 3 and 4 have different loop and if guards, but have identical loop invariants. The loop invariant about ct focuses on what has been computed so far. The program in Fig. 4 can also be verified using an alternative loop that focuses

```
public static int $CoincidenceCount0$(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 $m \leqslant f.Length$ && $n \leqslant g.Length$;
    invariant $ct$ == count{int $i$ in $(0 : m)$, int $j$ in $(0 : n)$; $f[i] == g[j]$};
    invariant $m == f.Length$ || forall{int $j$ in $(0 : n)$; $g[j] < f[m]$};
    invariant $n == g.Length$ || forall{int $i$ in $(0 : m)$; $f[i] < g[n]$};
  {
    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.

on what is left to compute, see Fig. 5. Dijkstra and Feijen, who consider the derivation of program from its specification, comment that this alternative invariant "leads more inevitably" [6] to this solution.

3.2 Minimal Segment Sum

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. 6. The main verification problems are due to the nesting of comprehensions in the program invariant. In particular, the verification of the invariants requires the **induction** axioms to be applied to both the inner and outer comprehensions, using a combination of the **induction** and **same terms** axioms. The verification also requires the fragile **distribution** axiom, which means our verifier is unable to prove the program using Z3.

```
public static int $CoincidenceCount1$(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 $m \leqslant f.Length$ && $n \leqslant g.Length$;
      invariant $ct$ == count{int $i$ in $(0 : m)$, int $j$ in $(0 : n)$; $f[i] == g[j]$};
      invariant $m == f.Length$ || forall{int $j$ in $(0 : n)$; $g[j] < f[m]$};
      invariant $n == g.Length$ || forall{int $i$ in $(0 : m)$; $f[i] < g[n]$};
   {
      if ($f[m] < g[n]$) {
         $m$++;
      } else if ($g[n] < f[m]$) {
         $n$++;
      } else {   // $g[n] == f[m]$
         $ct$++; $m$++; $n$++;
      }
   }
   return $ct$;
}
```

Fig. 4. A more efficient solution to the Coincidence Count problem that terminates the loop as soon as one array is exhausted. The program and its loop invariants are identical to the one in Fig. 3, except for the loop and if guards.

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 assist the verifier by adding **assert** or **assume** statements?

Debugging by adding prover-directive assertions at the Spec# level requires an understanding of the verification process and the methodology employed by the program verifier. Adding assertions merely to ensure that the correct axiom is triggered might make the proof appear mysterious to the student. This should be avoided by carefully guiding students to examples that the verifier can prove automatically.

The overall performance of the enhanced system programming system is acceptable. Table 7 shows the times required to verify a number of programs using two first-order SMT solvers, Simplify and Z3. As we would expect, the performance decreases as the number of comprehensions and the complexity of the invariants increase. In most cases, the Z3 solver verifies the programs slightly faster than Simplify. However, Simplify succeeds in verifying all of our examples where Z3 does not. Factorial cannot be verified by Z3 as multiplications by non-constants are, at the moment, essentially ig-

```
invariant $m \leqslant f.Length$ && $n \leqslant g.Length$;
invariant
      $ct + \mathbf{count}\{\mathbf{int}\ i\ \mathbf{in}\ (m : f.Length),\ \mathbf{int}\ j\ \mathbf{in}\ (n : g.Length);\ f[i] == g[j]\}$
      $== \mathbf{count}\{\mathbf{int}\ i\ \mathbf{in}\ (0 : f.Length),\ \mathbf{int}\ j\ \mathbf{in}\ (0 : g.Length);\ f[i] == g[j]\};$
```

Fig. 5. The alternative invariant for the Coincidence Count problem. This invariant can be used in lieu of the one in Fig. 4 to yield the program *CoincidenceCount2*.

```
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 $n \leqslant a.Length$;
    invariant x == min{int j in (0 .. n); min{int i in (0 .. j);
                          sum{int k in (i : j); a[k] }}};
    invariant y == min{int i in (0 .. n); sum{int k in (i : n); a[k] }};
  {
    y += a[n];
    if ($0 \leqslant y$) { y = 0; } else if (y < x) { x = y; }
  }
  return x;
}
```

Fig. 6. Spec# specification and solution of the Minimal Segment Sum problem.

nored. Simplify is willing to treat such multiplications as uninterpreted functions and hence it can verify the solution. Z3 cannot verify *MinSegmentSum* because the distribution of $+$ over the **min** comprehension is required, and our **distribution** axiom that states this property uses a trigger that contains a "$+$", which is not allowed in Z3.

We do not fully understand why Z3 cannot verify *CoincidenceCount1* in Fig. 4. If we remove the first of the two triggers for the **split range** axiom for the outer **count** comprehension, then Z3 verifies the program in less than 2 seconds. The problem therefore seems related to the first of these triggers setting off a chain of instantiations that prevent Z3 from completing the verification.

5 Related Work

Paulson and Meng [7] present work on translating Isabelle/HOL [9] to first-order logic. Their motivation is to improve the automation of interactive provers by integrating them with automatic provers which are usually based on first-order logic. Much of their work focuses on translating Isabelle's axiomatic type classes to first-order logic

Program	**Simplify**	**Z3**
Sum0	0.219	0.172
Sum1	0.063	0.016
Sum2	0.047	0.016
Sum3	0.110	0.016
Factorial	0.172	
MinSegmentSum	16.063	
CoincidenceCount0	6.017	1.815
CoincidenceCount1	18.970	
CoincidenceCount2	12.907	1.16

Fig. 7. Performance measurements (measured in seconds) of program verifications.

predicates and Isabelle types to first-order logic terms so that type information present in Isabelle/HOL is not lost during the translation.

Our work also translates higher-order functions to first-order logic but the comprehensions that we support do not require any type information to be carried in our encoding. This is due to all comprehensions supported by the Spec# program verifier having the same form:

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

Perfect Developer [3, 2], an automatic specification and verification environment, uses a custom theorem prover to provide 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 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 programs that use filtered subsequences and reverse summations). 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 the Spec# program verifier, using the SMT solvers Simplify and Z3. This implementation takes us a step closer to providing tool support for students learning program verification. Our axiomatisation 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 and abstraction invariants. Our aim is to provide a learning environment that focuses on writing good program specifications rather than burdening the user with seemingly formidable program verifications.

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. We also thank the anonymous referees, for their thoughtful and helpful comments.

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. Leonardo de Moura and Nikolaj Björner. Efficient e-matching for SMT solvers. In *Proceedings CADE 2007*, July 2007. To appear.
5. 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.
6. Edsger W. Dijkstra and W. H. J. Feijen. *A method of programming*. Addison-Wesley, July 1988.
7. Jia Meng and Lawrence C. Paulson. Translating higher-order problems to first-order clauses. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *ESCoR 2006: Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 70–80. http://ceur-ws.org, 2006.
8. Charles Gregory Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC, June 1981. The author's PhD thesis.
9. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.