Reasoning about Comprehensions with First-Order SMT Solvers

K. Rustan M. Leino
Microsoft Research
Redmond, WA, USA
leino@microsoft.com

Rosemary Monahan
National University of Ireland
Maynooth, Co. Kildare, Ireland
rosemary.monahan@nuim.ie

ABSTRACT
This paper presents a technique for translating common comprehension expressions (sum, count, product, min, and max) into verification conditions that can be tackled by two off-the-shelf first-order SMT solvers. Since a first-order SMT solver does not directly support the bound variables that occur in comprehension expressions, the challenge is to provide a sound axiomatisation that is strong enough to prove interesting programs and, furthermore, that can be used automatically by the 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.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—Assertion checkers, Class invariants, Correctness proofs, Formal methods, Programming by contract; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—Assertions, Invariants, Logics of programs, Mechanical verification, Pre- and post-conditions, Specification techniques

General Terms
Verification, Languages

Keywords
SMT Solvers, Matching Triggers, Quantifiers, Spec#

1. INTRODUCTION
We consider the automatic verification of programs that use comprehension expressions. A comprehension expression, sometimes called a generalised quantifier or an aggregate expression, prescribes a family of expressions that are to be combined using some operator. For example, each of

```
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;
```

Figure 1: Spec# method to sum the elements a[i], a[i+1], . . . , 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.

the two sum comprehensions in Fig. 1 prescribes a family of array elements that are combined using addition, thus summing various elements of array a. Other familiar comprehensions include product, min, and max, which apply the operator of the same name to elements of the family, and count, which adds 1 for every element that is in the family.

We present a technique for translating common comprehension expressions into verification conditions that can be tackled by a first-order Satisfiability Modulo Theories (SMT) solver. This translation is relevant for the verification of programs that are written in languages that support comprehensions of this form. Since a first-order SMT solver does not directly support the bound variables that occur in comprehension expressions, the challenge is to provide a sound axiomatisation that is strong enough to prove interesting programs and, furthermore, that can be used automatically by the SMT solver. To our knowledge, this has not previously been done for off-the-shelf SMT solvers.

The way today’s leading SMT solvers deal with universally quantified expressions in their input is to heuristically instantiate the quantifiers during the proof search. This process can be steered by giving matching triggers in the SMT-solver input [7]. A key to using such an SMT solver effectively—and a central difficulty faced by those who generate verification conditions for SMT solvers (see, e.g.,[10, 1])—lies in the design of appropriate matching triggers. In this paper, we show and discuss the matching triggers we use. Since the literature is scarce on practical guidance in the design of matching triggers, we hope our description pro-
provides illumination also in other applications of SMT solvers.

We have designed and implemented our technique in the Spec# program verifier [2]. The result is an environment where, using a choice of Simplify [7] or Z3 [6] 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 [8].

2. PRELIMINARIES

2.1 Notation

In Fig. 1, we show a method SegSum, which sums the elements in a segment of an array. We use the syntax of Spec# [3] (which is similar to that of several other object-oriented languages, including C#, Java, C++, and Python), since we later include Spec# programs that we have compiled and verified using our encoding technique. Using the sum comprehension

\[
\text{sum} \{ \text{int } k \text{ in } (i : j); \ a[k] \} \tag{1}
\]

where \( k \) in \((i : j)\) denotes the range \( i \leq k < j \).

SegSum’s postcondition expresses the summation of the \( j-i \) array elements starting with \( a[i] \). Similarly, the loop invariant says that \( s \) is the sum of the first \( n-i \) of these elements.

The general form of comprehension expressions that we consider in this paper is

\[
Q \{ \text{int } k \text{ in } (L : H); \ F; \ T \} \tag{2}
\]

where \( Q \) is \text{sum}, \text{count}, \text{product}, \text{min}, \text{or} \text{max}; \( k \) is a bound variable of type integer; the integer expressions \( L \) and \( H \) in the half-open interval \((L : H)\) say that each value of \( k \) satisfies \( L \leq k < H \); the boolean expression \( F \) is a filter that further restricts the values of \( k \) under consideration (if omitted, \( F \) defaults to \text{true}); and the integer (or for \text{count}, boolean) expression \( T \) is the term of the comprehension. The bound variable \( k \) can occur free in \( F \) and \( T \), but not in \( L \) or \( H \). The comprehension expression prescribes the family of expressions \( T \), for every \( k \) in the range \((L : H)\) that satisfies \( F \). The value of the comprehension is obtained by applying the (commutative and associative) operator associated with \( Q \) (for example, \(+\) for \text{sum}) to the expressions in this family.

As an example of a filter expression, comprehension (1) can also be expressed as:

\[
\text{sum} \{ \text{int } k \text{ in } (0 : \text{a.Length}); \ i \leq k \ \&\& \ k < j; \ a[k] \} \tag{3}
\]

2.2 Proofs

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

\[
\text{empty range} \quad (\forall \space lo, hi \bullet hi \leq lo \Rightarrow \text{sum} \{ \text{int } k \text{ in } (lo : hi); \ a[k] \} = 0)
\]

\[
\text{induction} \quad (\forall \space lo, hi \bullet lo \leq hi \Rightarrow \text{sum} \{ \text{int } k \text{ in } (lo : hi + 1); \ a[k] \} = \text{sum} \{ \text{int } k \text{ in } (lo : hi); \ a[k] \} + a[hi])
\]

We need to include these and other properties as axioms in the program verifier, in order for it to be able to verify programs that use comprehensions.

2.3 Using Spec#

In languages such as 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 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.

The Spec# program verifier works by translating compiled Spec# programs into the intermediate verification language BoogiePL, from which it then generates verification conditions for various SMT solvers [2]. BoogiePL is a simple first-order language that includes mathematical functions, arithmetic, and logical quantifiers, as well as syntax to indicate matching triggers. For the purposes of this paper, there are no significant differences between SMT-solver input and BoogiePL, so we will render our formulas in BoogiePL syntax.

Being a first-order language, BoogiePL (like the input of SMT solvers) has no direct support for comprehensions or binders, so the translation from Spec# into BoogiePL must instead use some suitable encoding that includes the mathematical properties of the comprehensions that we wish to support. Such an encoding will necessarily be incomplete, but we hope to achieve an encoding that is good enough in practice.

3. ENCODING COMPREHENSIONS AS FIRST-ORDER EXPRESSIONS

The key strategy in our translation is the introduction and axiomatisation of 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 Fig. 1 as a running example. The sum comprehension (1) has bound variable \( k \) with range \((i : j)\), (implicit) filter \text{true}, and term \( a[k] \).

The BoogiePL translations of these expressions are \( i, j, \text{true} \), and \( \text{ArrayGet}($\text{Heap}[a, s\text{elements}], k) \) respectively. (To understand this translation, think of every array as being an object with one instance field, \s\text{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 \text{ArrayGet}.)

3.1 Comprehension Functions

We consider the most general parameterisation of the expressions \( F \) and \( T \) in the comprehension (2), extracting what we call the template of the comprehension. The template is a triple whose first component is \( Q \) 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 (1) is

\[
(\text{sum}, \Box, \text{ArrayGet}(\Box, k))
\]

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 general form of a
function
with the understanding that Template as
produces a function. We shall refer to it as a instantiation.
stantiations are irrelevant to the verification goal, one can
universal quantifier with anything at all, but since most in
expressions that determines how the SMT solver instanti-
role in our encoding.
Comprehension functions. We present further details in [9].
function sum#0(lo: int, hi: int, a0: bool, a1: Elems)
returns (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” in the template. Intuitive-
vly, for a comprehension template (4), the comprehension
function has the following meaning:
\[ Q\#n(lo, hi, aa) = \]
\[ Q_{k\in(lo : hi)} \text{such that } Filter_{aa,k} \text{Term}[aa, k] \]
where aa corresponds to as many arguments as there are
□’s in the template.
For example, comprehension function sum#0 above has the
meaning:
\[ sum#0(lo, hi, a0, a1) = \]
\[ \sum_{k\in(lo : hi)} \text{such that } a_0 ArrayGet(a1, k) \]
Using comprehension function sum#0, the sum compre-
hension (1) 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 subexpress-
sions that were abstracted over in the template find them-
selves as arguments in the translation of a particular sum
comprehension.
As another example, the sum comprehension (3) with a
filter has the following template:
\[ (\text{sum, } \square \leq k \land k < \square, ArrayGet(\square, k)) \]
Thus, if both it and the sum comprehension (1) were present
in the same program, they would give rise to two different
comprehension functions. We present further details in [9].

3.2 Matching Triggers
For each comprehension function, our translation also gen-
erates a number of axioms. How the SMT solver instantiates
the quantifiers in these axioms is a crucial point in obtaining
the desired effect. In several SMT solvers, including Simplify
and Z3, this instantiation can be customised by supplying
matching triggers for the quantifiers [7]. Paying close atten-
tion to these triggers is a vital activity in designing program
verifiers that work with SMT solvers, so they play a central
role in our encoding.
A matching trigger of a universal quantifier is a set of
expressions that determines how the SMT solver instanti-
ates the quantifier. Logically, it is correct to instantiate a
universal quantifier with anything at all, but since most in-
stantiations are irrelevant to the verification goal, one can
hope for a more fruitful search by limiting which instantia-
tions 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 [12]) 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 Skolemises 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 en-
closes triggers in curly braces, the trigger
\[ (\forall x: \text{int, } y: \text{int } \cdot \{ 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: \text{int } \cdot \{ 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 deci-
sion 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 men-
tion + 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: \text{int } \cdot \{ h(x) \} \ h(x) < h(k(x)) \) \]
matches any argument of h, but when the quantifier is in-
stantiated, 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.

3.3 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, \texttt{sum\#n} and \texttt{s\#n} where their parameters \texttt{lo} and \texttt{hi} refer to the bounds of the comprehensions range. A value satisfying the filter on \texttt{aa} and from that range is generated from the heap. We axiomatise these function symbols to be synonyms of each other:

\[
(\forall \texttt{lo}: \texttt{int}, \texttt{hi}: \texttt{int}, \texttt{aa}: \mathcal{T} \bullet \{\texttt{s\#n(lo, hi, aa)}\} \\
\texttt{sum\#n(lo, hi, aa)} = \texttt{s\#n(lo, hi, aa)}
\]

Each sum comprehension in the Spec# program turns into a term that uses \texttt{sum\#n}, as we showed earlier in this section. For all axioms below, we use \texttt{s\#n} in all quantifier bodies, but we sometimes use \texttt{sum\#n} instead of \texttt{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 \texttt{s\#n}, instantiations will not give rise to any new \texttt{sum\#n} terms. Note that the \texttt{synonym} axiom above uses \texttt{sum\#n} in its trigger (shown in square braces), not \texttt{s\#n}; thus, for each \texttt{sum\#n} term in the input, the SMT solver will generate an equivalent \texttt{s\#n} term, but not vice versa.

We provide a \texttt{unit} axiom, which we render as follows:

\[
(\forall \texttt{lo}: \texttt{int}, \texttt{hi}: \texttt{int}, \texttt{aa}: \mathcal{T} \bullet \{\texttt{s\#n(lo, hi, aa)}\} \\
(\forall k: \texttt{int} \bullet k < h & \land \text{Filter}[\texttt{aa}, k]) \\
\Rightarrow \text{Term}[\texttt{aa}, k] = 0 \Rightarrow \texttt{s\#n(lo, hi, aa)} = 0
\]

where \texttt{Filter}[\texttt{aa}, k] and \text{Term}[\texttt{aa}, k] stand for the filter and term expressions of the comprehensions (as described in Section 3.1). Note that the \texttt{empty range} property in the Section 2.2 is a special case of the \texttt{unit} axiom. The trigger says for the outer quantifier to be instantiated for every occurrence of \texttt{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 \texttt{sum\#n} expression in the input to one instantiation of each induction axiom, which we achieve by mentioning \texttt{sum\#n}, not \texttt{s\#n}, in the triggers. We provide four induction axioms altogether.

The \texttt{induction below} axioms relate \texttt{s\#n(lo, hi, aa)} and \texttt{s\#n(lo + 1, hi, aa)}:

\[
(\forall \texttt{lo}: \texttt{int}, \texttt{hi}: \texttt{int}, \texttt{aa}: \mathcal{T} \bullet \{\texttt{s\#n(lo, hi, aa)}\} \\
\texttt{lo} < \texttt{hi} \land \text{Filter}[\texttt{aa}, \texttt{lo}] \\
\Rightarrow \texttt{s\#n(lo, hi, aa)} = \texttt{s\#n(lo + 1, hi, aa + Term[aa, lo])}
\]

and the \texttt{induction above} axioms relate \texttt{s\#n(lo, hi, aa)} and \texttt{s\#n(lo, hi − 1, aa)}:

\[
(\forall \texttt{lo}: \texttt{int}, \texttt{hi}: \texttt{int}, \texttt{aa}: \mathcal{T} \bullet \{\texttt{s\#n(lo, hi, aa)}\} \\
\texttt{lo} < \texttt{hi} \land \text{Filter}[\texttt{aa}, \texttt{hi} - 1] \\
\Rightarrow \texttt{s\#n(lo, hi, aa)} = \texttt{s\#n(lo, hi - 1, aa + Term[aa, hi - 1])}
\]

Another way to avoid matching loops would be to use \texttt{s\#n(lo + 1, hi, aa)} as the trigger for the \texttt{induction below} axioms and \texttt{s\#n(lo, hi − 1, aa)} as the trigger for the \texttt{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 \texttt{split range} axiom:

\[
(\forall \texttt{lo}: \texttt{int}, \texttt{mid}: \texttt{int}, \texttt{hi}: \texttt{int}, \texttt{aa}: \mathcal{T} \bullet \{\text{\texttt{sum\#n(lo, mid, aa)}}, \text{\texttt{sum\#n(mid, hi, aa)}\}} \\
\texttt{\texttt{sum\#n(lo, mid, aa)} + \texttt{sum\#n(mid, hi, aa)}} \\
\texttt{lo} \leq \texttt{mid} \land \texttt{mid} \leq \texttt{hi} \\
\Rightarrow \texttt{s\#n(lo, mid, aa)} = \texttt{s\#n(mid, hi, aa)}
\]

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

\[
\{\text{\texttt{sum\#n(lo, hi, aa)}, \text{\texttt{sum\#n(mid, hi, aa)}\}}
\]

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 5, including this trigger slowed down the verifications by as much as a factor of 35 with both Simplify and Z3). This slowdown is due to the number of instantiations that are made. Third, the triggers use \texttt{sum\#n}, despite the fact that using \texttt{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 \texttt{s\#n} is not increased). However, using \texttt{s\#n} had a bad impact on performance (by as much as a factor of 10 for our examples).

We also generate a \texttt{same terms} axiom:

\[
(\forall \texttt{lo}: \texttt{int}, \texttt{hi}: \texttt{int}, \texttt{aa}: \mathcal{T}, \texttt{bb}: \mathcal{T} \bullet \{\text{\texttt{sum\#n(lo, hi, aa)}, \text{\texttt{sum\#n(lo, hi, bb)}\}} \\
(\forall k: \texttt{int} \bullet \texttt{lo} \leq k \land k < \texttt{hi} \Rightarrow \\
(\text{\texttt{Filter}[aa, k]} = \text{\texttt{Filter}[bb, k]} \land \\
(\text{\texttt{Filter}[aa, k]} \Rightarrow \text{\texttt{Term}[aa, k]} = \text{\texttt{Term}[bb, k]})) \\
\Rightarrow \text{\texttt{s\#n(lo, hi, aa)} = \text{\texttt{s\#n(lo, hi, bb)}}
\]

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 (\texttt{lo} : \texttt{hi}) and, whenever the filters hold for a \texttt{k} in that range, the terms for \texttt{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

\[
\{\text{\texttt{s\#n(lo, hi, aa)}, \text{\texttt{s\#n(lo, hi, bb)}\}}
\]

without running the risk of matching loops, since instantiating the quantifier would not give rise to any \texttt{s\#n} terms that are not already required by this trigger. However, the trigger with two \texttt{s\#n} terms gave rise to unacceptable performance, so we chose to use \texttt{sum\#n} in one of the terms. We also tried specifying both terms in the trigger with \texttt{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 min and max comprehensions, we generate one more axiom, the distribution (of plus over min/max) axiom (here shown for min, using functions min#\#n and m#\#n):

\[(\forall k : \text{int} \bullet \text{lo} \leq k \land k < \text{hi} \Rightarrow
(\text{Filter}[\text{aa}, k] = \text{Filter}[\text{bb}, k]) \land
(\text{Term}[\text{aa}, k] + D = \text{Term}[\text{bb}, k])
\land
(\exists k : \text{int} \bullet \text{lo} \leq k \land k < \text{hi} \land \text{Filter}[\text{aa}, k]
\land \text{Term}[\text{aa}, k] + D = \text{Term}[\text{bb}, k])
\Rightarrow m#\#n(lo, hi, aa) + D = m#\#n(lo, hi, bb)\]

Several remarks are in order. First, for nonempty ranges, this axiom generalises 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 conjunction

\[\text{Term}[\text{aa}, k] + D = \text{Term}[\text{bb}, k]\]

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

\[\text{lo} < \text{hi} \land \text{Filter}[\text{aa}, k]\]

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.

3.4 Adequacy of the Axiomatisation

We make a few remarks about the adequacy of our axiomatisation. First, notice that all axioms concern just one 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

\[\text{lo}\]

...
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 ct == (10)
    count(int i in (0 : m), int j in (0 : n); f[i] == g[j]);
  invariant m <= f.Length && n <= g.Length &&
  (m == f.Length || forall(int j in (0 : n); g[j] < f[m]))
  & &
  (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 { ct++; m++; n++; }
  }
  return ct;
}

Figure 3: A solution to the Coincidence Count problem, written in Spec#. 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.

far or what is yet to be computed (cf. Fig. 2). The verification of the corresponding four programs [9] 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 numbers in Fig. 8.

4.2 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 included in the book of Dijkstra and Feijen [8]. A solution to this problem is shown in Fig. 3, while a less efficient version (called CoincidenceCount0) is presented in [9].

To get a better sense of how our axioms are used to prove this program, we now show the main ideas in the proof of the loop body. Please note: for this presentation, we will use Spec# notation; the proofs at the level of the BoogiePL encoding or the verification condition it gives rise to would be similar, just bigger. We will show a proof sketch for the maintenance of invariant (10), which is a shorthand for the nested comprehension

count(int i in (0 : m),
count(int j in (0 : n); f[j] == g[j]));

In Figs. 4, 5, and 6, we consider the three respective branches of the if statement in the body of the loop. We show that (10), suitably modified by the assignments in each if branch, follows from loop invariants and the enclosing guards. The nested comprehensions make this verification interesting: while induction can be readily applied to the outer comprehension, one first has to apply the same term axiom to the outer comprehension in order to apply induction to the inner comprehension. The main issue in the automatic verification of this program is triggering the instantiation of the various axioms.

count(int i in (0 : m);
  count(int j in (0 : n); f[i] == g[j]);
\begin{itemize}
  \item \text{by induction above axiom of sum } \end{itemize}

The program in Fig. 3 can also be verified using an alternative loop that focuses on what is left to compute, see Fig. 7. Dijkstra and Feijen, who consider the derivation of
4.3 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 \( 0 \leq i \leq j \leq a.Length \). This problem is also considered by Dijkstra and Feijen [8] (and a Spec# version called MinSegmentSum is presented in [9]).

The specification of the problem uses a sum comprehension nested inside two minimum comprehensions.

The main issue in the automatic verification of this program 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 verification also requires the fragile distribution axiom, which means our verifier is unable to prove the program using Z3.

5. EVALUATION

<table>
<thead>
<tr>
<th>Program</th>
<th>Ref</th>
<th>Simplify</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum0</td>
<td>[9]</td>
<td>0.219</td>
<td>0.172</td>
</tr>
<tr>
<td>Sum1</td>
<td>[9]</td>
<td>0.063</td>
<td>0.016</td>
</tr>
<tr>
<td>Sum2</td>
<td>Fig. 2</td>
<td>0.047</td>
<td>0.016</td>
</tr>
<tr>
<td>Sum3</td>
<td>[9]</td>
<td>0.110</td>
<td>0.016</td>
</tr>
<tr>
<td>Factorial</td>
<td></td>
<td>0.172</td>
<td></td>
</tr>
<tr>
<td>CoincidenceCount0</td>
<td>[9]</td>
<td>6.017</td>
<td>1.815</td>
</tr>
<tr>
<td>CoincidenceCount1</td>
<td>Fig. 3</td>
<td>18.970</td>
<td></td>
</tr>
<tr>
<td>CoincidenceCount2</td>
<td>Fig. 7</td>
<td>12.907</td>
<td>1.16</td>
</tr>
</tbody>
</table>

Figure 8: Performance measurements (in seconds) of program verifications taken on a Core 2 Duo laptop, running at 2.33GHz with a 4MB L2 cache. The version of the Spec# static program verifier used is the July 2007 release.

Many of the difficulties met during our program verifications (which we did while developing the axioms and triggers we have described) were in trying to diagnosis 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 statements?

The overall performance of the verifier is acceptable. Figure 8 shows the times required to verify a number of programs using two SMT solvers: Simplify and Z3. In most cases, the Z3 solver verifies the programs slightly faster than Simplify. However, Simplify succeeds in verifying all of our examples whereas Z3 does not. Factorial cannot be verified by Z3 as multiplications by non-constants are, in the current Z3 version, essentially ignored. Simplify is willing to treat such multiplications as uninterpreted functions and hence it can verify the program.

The verification of MinSegmentSum requires the distribution of + over the min comprehension, and since the trigger of our distribution axiom mentions +, Z3 cannot verify the program. We do not fully understand why Z3 cannot verify CoincidenceCount1. 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.

6. RELATED WORK

Paulson and Meng [11] present work on translating Isabelle/HOL [13] 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 predicates and Isabelle types to first-order logic terms so that type information present in Isabelle/HOL is not lost during the translation. Comprehensions are a special form of higher-order functions, but our translation of them into first-order logic does not require carrying type information.

Perfect Developer [5, 4], 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.

7. CONCLUSIONS AND FUTURE WORK

We have designed and implemented support for summation-like comprehensions in the Spec# program verifier. Our technique extracts templates from the comprehensions used in the program to be verified, introduces functions for these templates, and generates axioms about the functions. As a byproduct of describing our design, our paper also contributes something of a tutorial on using matching triggers, which is a crucial point in encoding verification problems for SMT solvers.

Our axiomatisation is of a modest size, and our approach can automatically verify some challenging textbook examples. Yet, several challenges lie ahead. One challenge is to do a better job of explaining error messages to users. Another is to overcome the trigger fragility problem of axioms like distribution. Yet another challenge lies in taking on more programs that use comprehensions, like Minimum Distance or Maximal Monotone Subsequence [8]. Such programs may rely on mathematical properties that are specific to the problem, so a challenge is to find a balance between
the set of axioms built into the tool and an easy-to-use, sound mechanism by which users can extend that axioms set.

Another direction for future work, which in fact was the original motivation behind our work described in this paper, is to employ more tool support in the teaching of program correctness. Because of the immediate feedback they provide, tools can help cement some of the program-correctness concepts in the minds of students. Many textbooks on program correctness use as their illustrative examples simple programs that solve problems in familiar mathematical domains, such as arithmetic, which includes the summation comprehension. Our technique and implementation take us a step closer to providing program-correctness tools for students. With more such steps, we hope to develop tool support for a larger repertoire of programs used in educating students.

8. ACKNOWLEDGEMENTS

We thank the participants of the IFIP WG 2.3 meeting in Sydney, January 2007 and the referees and participants at the FTfJP 2007 workshop for serving as a springboard for the initial ideas.

9. REFERENCES


