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 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.

```
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 $a[i], a[i+1], \ldots, a[j-1]$. 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.

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 difficultly faced by those who generate verification conditions for SMT solvers (see, *e.g.*[10, 1]—lies in the design of appropriate matching triggers. In

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

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 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

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

where k **in** $(i:\ j)$ denotes the range $i \leqslant k < j$, *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

$$\mathcal{Q}\{\mathbf{int}\ k\ \mathbf{in}\ (L:\ H),\ F;\ T\} \tag{2}$$

where $\mathcal{Q}$ is **sum**, **count**, **product**, **min**, or **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 $L \leqslant k < H$; 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 **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 $\mathcal{Q}$ (for example, $+$ for **sum**) to the expressions in this family.

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

$$\mathbf{sum}\{\mathbf{int}\ k\ \mathbf{in}\ (0:\ a.Length),\ i \leqslant k\ \&\&\ k < j;\ a[k]\} \tag{3}$$

2.2 Proofs

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

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

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

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 *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*.)

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 $\mathcal{Q}$ 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 (1) is

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

Each "□" 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

$$(\mathcal{Q},\ \mathit{Filter}[\![\Box \ldots \Box, k]\!],\ \mathit{Term}[\![\Box \ldots \Box, k]\!]) \qquad (4)$$

with the understanding that $\mathit{Filter}[\![\Box \ldots \Box, k]\!]$ and $\mathit{Term}[\![\Box \ldots \Box, k]\!]$ stand for expressions that can mention k and some number of □'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 (1) in program *SegSum* gives rise to the following comprehension function in our translation into BoogiePL:

function $\mathit{sum}\#0(\mathit{lo}\colon \mathbf{int}, \mathit{hi}\colon \mathbf{int}, a0\colon \mathbf{bool}, a1\colon \mathit{Elems})$
  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. Intuitively, for a comprehension template (4), the comprehension function has the following meaning:

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

where aa corresponds to as many arguments as there are □'s in the template.

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

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

Using comprehension function $\mathit{sum}\#0$, the sum comprehension (1) translates into BoogiePL as

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

Notice how the filter and term of the template are part of the intuitive meaning of $\mathit{sum}\#0$, 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 (3) with a filter has the following template:

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

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 generates 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 attention 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 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 [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 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.

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

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

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:

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

where $Filter[\![aa, k]\!]$ and $Term[\![aa, k]\!]$ stand for the filter and term expressions of the comprehensions (as described in Section 3.1). Note that the **empty range** property in the Section 2.2 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 $s\#n(lo + 1, hi, aa)$:

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

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

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

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:

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

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

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

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 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 $sum\#n$, 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:

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

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 $(lo : hi)$ 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 **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 $m\#n$):

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

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

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 function: there is no axiom that relates two different comprehension functions. For example, since sum comprehension (1) has a different template than sum comprehension (3), 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 (3) that uses the filter, then the verification would not be able to establish the postcondition (which is written in the form (1)) 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 writing 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 an effective **distribution** axiom for Z3 means that it cannot verify examples like Minimal Segment Sum (Section 4.3).

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 that iterate backwards. Such a 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 **assert** $a[n] ==$ **sum{int** k **in** $(n:\ n+1);\ a[k]\}$; before modifying the variable s in Fig. 2, would be enough to make this program verify even without the **induction below** axiom. Simply mentioning the 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.

```
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;
    invariant s == sum{int i in (n : a.Length); a[i]};
  {
    s += a[n];
  }
  return s;
}
```

Figure 2: A program that sums an array's elements starting from its last element using a loop invariant that focuses on what has been summed so far. Our induction below axiom allows the verification of programs that have this form.

4. SOME MORE DIFFICULT EXAMPLES

We now report on our experience of using our encoding,

```
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 ==                                          (I0)
    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.

implemented in Spec#, to verify some more challenging examples, including some programming problems described by Dijkstra and Feijen [8]. We begin with an example that illustrates the use of alternative loop invariants.

4.1 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 (*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 *CoincidenceCount*0) 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 (I0), which is a shorthand for the

```
ct == sum{int i  in (0 : m + 1);
          count{int j  in (0 : n); f[j] == g[j]}};
 ⇐   < by induction above axiom of sum >
ct == sum{int i  in (0 : m);
          count{int j  in (0 : n); f[j] == g[j]}};
          + count{int j in (0 :  n); f[m] == g[j]}
 ⇐   < by the invariant >
0 == count{int j in (0 :  n); f[m] == g[j]}
 ⇐   < by unit axiom of the count comprehension >
forall{int j in (0 :  n); f[m] ≠ g[j]}
 ⇐   < by the invariant >
true
```

Figure 4: A sketch of the correctness argument that the if branch $f[m] < g[n]$ in program *CoincidenceCount*1 maintains invariant (I0).

```
ct == sum{int i  in (0 : m);
          count{int j  in (0 : n + 1); f[j] == g[j]}};
 ⇐   < by the invariant >
sum{int i  in (0 : m);
          count{int j  in (0 : n); f[j] == g[j]}};
== sum{int i  in (0 : m);
          count{int j  in (0 : n + 1); f[j] == g[j]}};
 ⇐   < by same term axiom of sum >
forall{int i in (0 :  m); count{int j in (0 :  n); f[i] == g[j]}
        ==  count{int j in (0 :  n + 1); f[i] == g[j]}}
 ⇐   < by induction above axiom of count >
forall{int i in (0 :  m); f[i] ≠ g[n]}
 ⇐   < by the invariant >
true
```

Figure 5: A sketch of the correctness argument that the else if branch $g[n] < f[m]$ in program *CoincidenceCount*1 maintains invariant (I0).

nested comprehension

```
ct == sum{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 (I0), 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.

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 a program from its specification, comment that this alternative invariant "leads more inevitably" [8] to this solution. This solution also verifies automatically using our encoding, using deductions similar to those described above.

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 $0 \leqslant i \leqslant j \leqslant a.Length$. This problem is also considered by Dijkstra and Feijen [8] (and a

$ct + 1 ==$ **sum**{**int** i **in** $(0 : m + 1)$;
count{**int** i **in** $(0 : n + 1); f[j] == g[j]$}};
$\Leftarrow$ < by **induction above** axiom of **sum** >
$ct + 1 ==$ **sum**{**int** i **in** $(0 : m)$;
count{**int** j **in** $(0 : n + 1); f[j] == g[j]$}}
+ **count**{**int** j **in** $(0 : n + 1)$; $f[m] == g[j]$}
$\Leftarrow$ < by **induction above** axiom of **count** >
$ct + 1 ==$ **sum**{**int** i **in** $(0 : m)$;
count{**int** j **in** $(0 : n + 1); f[j] == g[j]$}}
+ **count**{**int** j **in** $(0 : n)$; $f[m] == g[j]$} + 1
$\Leftarrow$ < by **unit** axiom of **count** >
$ct ==$ **sum**{**int** i **in** $(0 : m)$;
count{**int** j **in** $(0 : n + 1); f[j] == g[j]$}}
&& **forall**{**int** j **in** $(0 : n)$; $f[m] \neq g[j]$}
$\Leftarrow$ < by the invariant >
$ct ==$ **sum**{**int** i **in** $(0 : m)$;
count{**int** j **in** $(0 : n + 1); f[j] == g[j]$}}
$\Leftarrow$ < by the calculation in Fig. 5 >
true

Figure 6: A sketch of the correctness argument that the else branch $g[n] == f[m]$ in program *CoincidenceCount*1 maintains invariant (I0).

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

Figure 7: A different invariant for the Coincidence Count solution in Fig. 3. This invariant can be used in lieu of the one in Fig. 3 to yield the program *CoincidenceCount*2, as we refer to it in our Fig. 8 performance summary.

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

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 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** 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

Program	Ref	Simplify	Z3
Sum0	[9]	0.219	0.172
Sum1	[9]	0.063	0.016
Sum2	Fig. 2	0.047	0.016
Sum3	[9]	0.110	0.016
Factorial		0.172	
CoincidenceCount0	[9]	6.017	1.815
CoincidenceCount1	Fig. 3	18.970	
CoincidenceCount2	Fig. 7	12.907	1.16
MinSegmentSum	[9]	16.063	

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.

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 *CoincidenceCount*1. 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

[1] A. Banerjee, M. Barnett, and D. A. Naumann. Boogie meets regions: A verification experience report. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, VSTTE 2008*, volume 5295 of *Lecture Notes in Computer Science*, pages 177–191. Springer, October 2008.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. de Roever, editors, *FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J. Lanet, and T. Muntean, editors, *CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[4] G. Carter, R. Monahan, and J. M. Morris. Software refinement with Perfect Developer. In B. K. Aichernig and B. Beckert, editors, *SEFM 2005*, pages 363–373. IEEE Computer Society, September 2005.

[5] D. Crocker and J. Carlton. A high productivity tool for formally verified software development. Technical report, Escher Technologies, September 2004. `http://www.eschertech.com/papers/pdpaper.pdf`.

[6] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In F. Pfenning, editor, *CADE-21*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, July 2007.

[7] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[8] E. W. Dijkstra and W. H. J. Feijen. *A method of programming*. Addison-Wesley, July 1988.

[9] K. R. M. Leino and R. Monahan. Automatic verification of textbook programs that use comprehensions. 9th Workshop on Formal Techniques for Java-like Programs, FTfJP 2007, 2007.

[10] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI 2003*, pages 220–231. ACM Press, 2003.

[11] J. Meng and L. C. Paulson. Translating higher-order problems to first-order clauses. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *ESCoR 2006: Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 70–80. `http://ceur-ws.org`, 2006.

[12] G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC, June 1981. The author's PhD thesis.

[13] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.