# Verifying the reflective visitor pattern

Ben Horsfall
Department of Informatics
University of Sussex
b.g.horsfall@sussex.ac.uk

Nathaniel Charlton
Department of Informatics
University of Sussex
billiejoecharlton@gmail.com

Bernhard Reus
Department of Informatics
University of Sussex
bernhard@sussex.ac.uk

## ABSTRACT
Computational reflection allows a program to inspect and manipulate the structure or behaviour of itself at runtime. Often this means that it is possible to create more generic or adaptable programs in an elegant way. However, there is little support for specification and automatic verification of reflective programs. We address this problem by implementing, specifying, and verifying a reflective library using a Hoare-logic for a simple language with stored procedures. The latter is important since reflective metadata is modelled on the heap, thus method objects will be realised as stored procedures. We verify memory safety as well as functional correctness of an instance of the reflective visitor pattern, including the reflective library. The entire verification is carried out in our (semi-)automatic verification tool Crowfoot.

## 1. INTRODUCTION
The visitor design pattern [8] provides a general method for traversing a data structure (possibly a collection of several different types of object), and performing operations on each element. The advantage of the visitor pattern is that it provides a generic tree traversal that can be instantiated with different algorithms (see for example [12]). However, a disadvantage of the standard pattern is that if we change the data structure itself, for instance by adding a new type of node to a tree structure, we have to extend the visitor interface and thus every class which implements it. This is contrary to the *Open/closed* principle of object-oriented programming. Additionally, there is indirection with the *accept* methods, which is tedious to program, reduces readability, and demonstrates that the pattern is *intrusive* because the traversal operation can't just be added on – it requires these accept methods to be inserted into the structure.

We can overcome these disadvantages by using a reflective version of the visitor pattern, inspired by [12, 3]. Essentially, one creates a generic visit method that uses reflection to decide which concrete visit should be invoked, based on inspecting the runtime type of the node being visited. Another benefit of a reflective version is that we can have a default behaviour for unknown objects.

If we want to verify an instance of the reflective visitor pattern, however, we run into the problem that verification tools such as [2, 6, 10] do not support reflection, or do so weakly [11]. Therefore this paper investigates how to specify and verify programs that use reflection, using an instance of the reflective visitor as a guiding example. This involves developing specifications for the necessary reflective operations.

Rather than analysing programs in a language with built-in reflective operations, we informally translate into a language without reflective features and *implement* the reflective operations in terms of more primitive operations and data structures. Thus we need not consider reflection at the level of programming language semantics and avoid having to devise and implement proof rules for reflective program statements. In particular we use the programming language analysed by our verification tool Crowfoot [4, 1] which allows us to semi-automatically verify the program. Crowfoot's logic provides a natural way to express the relationship between the behaviour of methods and the metadata which describes them. The full verified code of the example (including the reflective library) is available on [1].

## 2. THE REFLECTIVE VISITOR
In this section, we present an example instance of the reflective visitor. Our example structure is a binary tree, composed of Leaf and Parent objects and the goal of our visitor is to count the number of leaves. For comparison, in Fig. 1 we first show the example using the standard visitor pattern [8]. The reflective version is given in Fig. 2. Note that Visitor becomes an abstract class, with a visit method that is inherited by CountVisitor. Also note the removal of the Node interface, and the accept methods.

The code of the generic visit, which can be used for *any type of object*, is presented using the Crowfoot language (see Section 3.1) in Fig. 3. The important reflective features are the ability to introspect the metadata of a class to discover the name of the class and what methods it includes. The first two call commands get us the name of the object's class. The third call appends this name to the constant *VISIT*, containing the string "visit". Using the result of this concatenation, we then look for a method with this name in the current class and if found it is invoked. We will return

*FTfJP'12,* June 12, 2012, Beijing, China

```
interface Visitor {
  visitParent(Parent p);
  visitLeaf(Leaf l);
}
class CountVisitor implements Visitor {
  int val = 0;
  visitParent(Parent p) {
    p.l.accept(this);
    p.r.accept(this);
  }
  visitLeaf(Leaf l) { val++; }
}
interface Node {
  accept(Visitor v);
}
class Parent implements Node {
  Node l,r;
  accept(Visitor v) { v.visitParent(this); }
}
class Leaf implements Node {
  int val;
  accept(Visitor v) { v.visitLeaf(this); }
}
```

**Figure 1: Instance of the standard non-reflective visitor pattern**

to this visit method later and endow it with an appropriate specification that we can show it satisfies.

Although the visitor pattern is necessarily class-based object oriented and the language of our verifier is a procedural language, every effort has been made to present the example in a Java-like language style, e.g. C_method() represents `C.method()`.

## 3. THE LANGUAGE IN USE

### 3.1 Programming language

Our program is written in an imperative heap manipulating language with pointer arithmetic and recursive procedures. The details of the language can be found in [4]. The example in Fig. 3 demonstrates parts of the language. Allocation of a new heap cell at address $a$ is achieved with $a :=$ new 0 (using initial contents 0), and deallocation with dispose $a$. Square brackets are used to dereference addresses, so the statement $[a] := x$ stores the value of $x$ onto the heap at address $a$, whilst $x := [a]$ reads the contents at address $a$ into $x$. Fixed procedures can be called with the call command. Note that procedures don't have a return value, so we use the convention of having the last argument to be the address of a cell where the result can be stored (named *res* in our example). Procedures can be stored on the heap and then invoked via the eval command. This feature, often referred to as *higher-order store*, is needed to store the classes' methods in our visitor example (see the metadata representation given in Section 4.1) since loaded classes and their metadata are stored on the heap. Hence the reflective method invoke operation is implemented with eval.

```
abstract class Visitor {
  visit(Object o) { ... }
}
class CountVisitor extends Visitor {
  int val = 0;
  visitParent(Parent p) { visit(p.l); visit(p.r); }
  visitLeaf(Leaf l) { val++; }
}
class Parent { Object l,r; }     class Leaf { int val; }
```

**Figure 2: Our reflective visitor instance**

```
proc CountVisitor_visit(this, obj) {
  locals res, obj_class, obj_className, this_class, methodName,
    method;
  res := new 0;
  call Object_getClass(obj, res);
  obj_class := [res];
  call Class_getName(obj_class, res);
  obj_className := [res];
  call append(VISIT, obj_className, res);
  methodName := [res];
  call Object_getClass(this, res);
  this_class := [res];
  call Class_getMethod(this_class, methodName, res);
  method := [res];
  call Method_invoke(method, this, obj);
  dispose res
}
```

**Figure 3: A visitor using reflective features, presented in the Crowfoot language.**

Recall that, as stated in Sec. 1, the language does not contain built-in reflective operations. They will be implemented using heap data structures.

### 3.2 Assertion language

The assertion language (described in detail in [4]) is separation logic [14], crucially extended with *nested Hoare triples* for describing heaps that can contain code (see [15, 16]). The assertion $a \mapsto \forall r.\ \{r \mapsto \_\} \cdot (r)\, \{r \mapsto \_\}$, for instance, states that the heap cell at address $a$ contains code which satisfies the given specification (i.e. a heap cell at the address of the only argument is required and maintained). Whereas all the expressions in the programming language denote integers, our assertion language includes sets (set variables are prefixed by the %-sign) and tuples.

Another important feature of our assertion language is the support for user-defined predicates (declared with the keyword recdef). These are prefixed by the \$-sign, for instance $\$\mathrm{P}(a, b; \%S)$, and we separate integer arguments from set arguments with a semi-colon. The language given in [4] has been enriched to allow sets-of-sets and a projection map, $\mathrm{proj}_n$ that maps sets of tuples to sets by lifting the standard projection map $\pi_n$ to sets. *Abstract* predicates without a definition are permitted.

With no built in support for strings, we need to devise a method for modelling them and the *append* operation needed for the reflective visitor. We simply assume that there exists some encoding from string to integer, and abstract away from it by using two abstract predicates: firstly \$AppendString($s, t, st$) states that $st$ is the result of appending $t$ onto $s$, and secondly \$AppendStringAll ($s$; %$t$, %$st$) states that set %$st$ results from appending all elements of %$t$ onto common prefix $s$.

To support the abstract predicates, it is necessary to provide some lemmas to more precisely describe the abstract encoding. Three examples of such lemmas are given in Fig. 4. These are provided to our verification tool in the form of abstract procedures, which can be used on-demand by adding call commands at the necessary points in the program.

```
// appending the same strings yields the same result
proc abstract ghost_lemma_append()
  ∀ a, b, c, d.
  pre : $AppendString(a, b, c) ⋆ $AppendString(a, b, d);
  post : c = d;

// appending distinct strings yields distinct results
proc abstract ghost_lemma_append_distinct()
  ∀x, a, b, c, d.
  pre : $AppendString(x, a, b) ⋆ $AppendString(x, c, d) ⋆  a ≠ c
  post : b ≠ d;

// core property of $AppendStringAll
proc abstract ghost_unfold_append()
  ∀a, %B, b, %C.
  pre : $AppendStringAll(a; %B, %C) ⋆ b ∈ %B;
  post : ∃c. $AppendString(a, b, c) ⋆ (c, b) ∈ %C;
```

**Figure 4: Lemmas for our encoding of strings as integers**

Further, in the visitor example we use the following string constants: *VISIT*, *LEAF*, *PARENT*, *COUNTVISITOR*. We assume that they are all different by means of further abstract lemmas that we omit (see full code).

## 4. REFLECTIVE LIBRARY

### 4.1 Metadata representation

To specify and implement a reflective library we first need to discuss how to represent metadata. In line with common reflective libraries, metadata is kept on the heap where it can be manipulated by the program. All predicates used to describe the metadata are given in Fig. 5.

Since our example is an instance of the reflective visitor pattern implemented in a Java-like language, we must represent the object-oriented aspect. To avoid burdening ourselves too much, we keep this simple and only implement the parts of the system we actually need (for instance, it is not possible to reflect on the fields of a class). We also simplify by not having the metadata represented by objects and thus avoid the need for meta-classes.

**Metadata container** At the top level, we have a predicate \$Meta which encompasses all the class/method metadata (in a list). Any procedure that wants to perform reflection must carry this predicate in its specification. We define a global constant *META* pointing to the start of the class list.

**Classes** The \$ClassLseg predicate is used to represent the list of classes currently loaded into the system (more precisely we define list *segments* in a standard way). For each class the metadata contains the class name (*strName*) and its list of methods (*ms*). Note the inclusion of the constraint $strName \notin \text{proj}_2(\%rest)$, saying that there can never be more than one class with the same name.

**Methods** Contained within each class is a list of methods (described again as list segments). This contains the method's code that satisfies a specification abbreviated by $\mathfrak{GenSpec}$, its name (*mName*), and a list of types of its parameters (starting at address *types*). Predicate \$ParamTypes ($a$, *len*; %*types*) uses the standard linked-list structure, however the ordering of elements is important here. Therefore each element of %*types* is a pair[1], where the first part identifies the position in the list. For simplicity we actually use the length of the list to identify the position, so effectively the numbering is reversed because the parameter list described by \$ParamTypes is defined inductively on the length *len*.

The definition of \$MethodLseg shows how Crowfoot's assertion language allows us to express relationships between the behaviour of each method, and the metadata describing it: *targetClass* and *argClass* found in the metadata also appear in the specification $\mathfrak{GenSpec}$ of the method code.

The method's behavioural specification, $\mathfrak{GenSpec}$, has been designed to be sufficiently generic to cater for a large set of possible methods by delegating the behaviour to concrete definitions of the \$Pre and \$Post predicates. However, we have fixed the number of arguments at two[2]. The first is the *target* of the method (the visitor object), and the second, *arg*, is the single argument (the node object being visited).

**Objects** We define a predicate \$Object in Fig. 6, instances of which represent the heap space taken up by objects (not including referenced objects' heap space). This includes the dynamic type of the object and its fields, where we separate primitive integer fields from the fields containing other objects (represented by the respective sets %*intFs* and %*objFs*). The shape of the fields is described by \$ObjFields which requires a concrete definition reflecting the classes present in the program being verified. Each field has an identifier $(1, 2 \ldots)$, and for object-fields their class is recorded in addition to their value. Our example uses three different classes, hence its definition has three disjuncts.

**Program specific predicates** The predicate instances of \$Pre and \$Post as well as all visitor specific predicate definitions can be found in Fig. 7. Because the visitor proceeds by visiting the current node and then being directed through any children, all the children need to appear in the specification. Predicate \$Tree(*root*, *type*; %$T$) takes care of this, describing a (non-empty) binary tree whose root object

[1]We wouldn't need to encode it this way, as a pair, if our assertion language supported lists and not just sets.
[2]This is purely to simplify the example, and there should be no problem in having an arbitrary number of arguments by wrapping them up into a list.

$$\begin{array}{l}
\textsf{recdef}\ \$\mathrm{Meta}(;\%cs) := \\
\quad \exists list.\ META \mapsto list \star \$\mathrm{ClassLseg}(list, 0; \%cs);
\end{array}$$

$$\begin{array}{l}
\textsf{recdef}\ \$\mathrm{ClassLseg}(a, z; \%cs) := a = z \star \%cs = \emptyset \\
\vee \quad \exists next, strName, ms, \%ms, \%rest. \\
\qquad a \mapsto strName, ms, next \\
\qquad \star\ strName \notin \mathrm{proj}_2(\%rest) \star \$\mathrm{MethodLseg}(ms, 0; \%ms) \\
\qquad \star\ \$\mathrm{ClassLseg}(next, z; \%rest) \\
\qquad \star\ \%cs = \{(a, strName, \%ms)\} \cup \%rest;
\end{array}$$

$$\begin{array}{l}
\textsf{recdef}\ \$\mathrm{MethodLseg}(a, z; \%methods) := a = z \star \%methods = \emptyset \\
\vee \quad \exists\ next, mName, types, targetClass, argClass, \%types, \%rest. \\
\qquad a \mapsto \mathfrak{GenSpec}(targetClass, argClass), mName, types, next \\
\qquad \star\ \ \$\mathrm{ParamTypes}(types, 2; \%types) \\
\qquad \star\ \ \%types = \{(1, targetClass)\} \cup \{(2, argClass)\} \\
\qquad \star\ \ mName \notin \mathrm{proj}_2(\%rest) \star \$\mathrm{MethodLseg}(next, z; \%rest) \\
\qquad \star\ \ \%methods = \{(a, mName, \%types)\} \cup \%rest;
\end{array}$$

$$\begin{array}{l}
\textsf{recdef}\ \$\mathrm{ParamTypes}(a, len; \%types) := \\
\quad a = 0 \star \%types = \emptyset \star len = 0 \\
\vee \quad \exists\ paramType, next, \%rest. \\
\qquad a \mapsto paramType, next \star \$\mathrm{ParamTypes}(next, len - 1; \%rest) \\
\qquad \star\ \ \%types = \{(len, paramType)\} \cup \%rest \star len \neq 0;
\end{array}$$

$$\begin{array}{l}
\mathfrak{GenSpec}(targetClass, argClass) \triangleq \\
\quad \forall\ target, arg, \%cs, \%r, \%tObjFs, \%tIntFsPre. \\
\qquad \left\{ \begin{array}{l} \$\mathrm{Pre}(target, targetClass, arg, argClass; \\ \qquad\qquad \%tIntFsPre, \%tObjFs, \%cs, \%r) \end{array} \right\} \\
\qquad\qquad\qquad \cdot(target, arg) \\
\qquad \left\{ \begin{array}{l} \$\mathrm{Post}(target, targetClass, arg, argClass; \\ \qquad\qquad \%tIntFsPre, \%tObjFs, \%cs, \%r) \end{array} \right\}
\end{array}$$

**Figure 5: Predicates describing metadata**

is *root*:*type*, and whose node pointers are contained in set $\%T$, paired with their types. Note that the usage of $\star$ in \$Tree enforces that disjoint heap cells are used by the objects to actually establish a tree structure. The objects can still contain extra fields that point back into the tree.

Both the pre- and postcondition contain the spatial information regarding the *target* object, the tree with root *arg* and the metadata, which is important because the methods we are reflectively invoking may need to use reflection themselves. The rest of the pre-condition is pure. It first ensures that the classes of all the objects in the tree are defined in the metadata, including the target object which is the visitor and secondly that, for each of these types, there exists a visit method in the target's class (see \$MethodsExist).

The post-condition is almost identical to the pre-condition. The difference is the extra \$Fun predicate occurrence, and the different set of integer variables held in the visitor object. This reflects the state change that the visitor goes through when visiting elements. More concretely, the pred-

$$\begin{array}{l}
\textsf{recdef}\ \$\mathrm{Object}(ptr, type; \%intFields, \%objFields) := \\
\quad ptr \mapsto type \star \$\mathrm{ObjFields}(ptr, type; \%intFields, \%objFields);
\end{array}$$

$$\begin{array}{l}
\textsf{recdef}\ \$\mathrm{ObjFields}(ptr, type; \%intFields, \%objFields) := \\
\qquad \exists\ val.\ type = LEAF \star ptr + 1 \mapsto val \\
\qquad\quad \star\ \ \%intFields = \{(1, val)\} \star \%objFields = \emptyset \\
\quad \vee \quad \exists\ l, l\_type, r, r\_type.\ type = PARENT \star ptr + 1 \mapsto l, r \\
\qquad\quad \star\ \ \%intFields = \emptyset \\
\qquad\quad \star\ \%objFields = \{(1, l, l\_type)\} \cup \{(2, r, r\_type)\} \\
\quad \vee \quad \exists\ val.\ type = COUNTVISITOR \star ptr + 1 \mapsto val \\
\qquad\quad \star\ \ \%intFields = \{(1, val)\} \star \%objFields = \emptyset;
\end{array}$$

**Figure 6: Predicates describing objects**

icate \$Fun$(;\%T, \%pre, \%post)$ specifies the "action" of the visiting method, i.e. that, if starting in a visitor state with integer fields as listed in $\%pre$, visiting tree structure $\%T$ will result in integer fields as listed in $\%post$. Abstracting the precise behaviour in predicate \$Fun means that the generic specification is applicable to a large set of visitors that each perform different operations. The particular instance of \$Fun for our CountVisitor requires one integer variable that will be incremented by the size of the tree $\%T$. Auxiliary predicate \$TreeSize$(n; \%T)$ states that tree $\%T$ has $n$ Leaf nodes.

Note that pre- and post-condition refer to the *same* tree \$Tree$(arg, argClass; \%T)$. This does limit the behaviour of the visitor, for example it wouldn't be possible to have a visitor which deletes any elements from the structure. However, it is only the *structure* that is exposed by the $\%T$ argument, so the content of each node may change.

It is worth pointing out that the predicates \$Pre/ \$Post and \$Meta are necessarily mutually recursively defined due to the nature of *higher-order store.*

## 4.2 Specification

The specifications of the procedures in our reflection library are given in Fig. 8. It is important to stress here that the specification of the reflective library uses a generic \$Object and generic \$Pre and \$Post predicates in the metadata representation \$Meta so we can clearly separate the specification of the reflective library from any concrete example using it. We briefly explain the specifications below.

Note that for the reflective operations we need here, the implementation will only ever read the metadata and not alter it. This is ensured to some extent in the specifications by the outer quantified $\%cs$ set that is the same in both the pre- and post-conditions. Recalling the structure of this set from Fig. 5, one can see that it includes the addresses of the list cells[3] as well as the content of the cells, therefore the specifications ensure that the *content* of the the metadata is not allowed to change. However, it would be possible for the *ordering* to change because the *next* pointers aren't included in $\%cs$. This weakness can obviously be overcome by

[3]This is necessary for the way our verification tool 'splits' list segments during verification.

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{Tree}(root, type; \%T) := \\
\quad \exists\ \%intFs, \%objFs, \%L, \%R, l, lCls, r, rCls. \\
\quad type = PARENT \star \$\mathrm{Object}(root, PARENT; \%intFs, \%objFs) \\
\quad \star\ \ \%T = \{(root, PARENT)\} \cup \%L \cup \%R \\
\quad \star\ \ (1, l, lCls) \in \%objFs \star \$\mathrm{Tree}(l, lCls; \%L) \star (l, lCls) \in \%L \\
\quad \star\ \ (2, r, rCls) \in \%objFs \star \$\mathrm{Tree}(r, rCls; \%R) \star (r, rCls) \in \%R \\
\quad \star\ \ \{lCls\} \cup \{rCls\} \subseteq \{LEAF\} \cup \{PARENT\} \\
\vee \\
\quad \exists\ \%intFs, \%objFs. \\
\quad type = LEAF \star \$\mathrm{Object}(root, LEAF; \%intFs, \%objFs) \\
\quad \star\ \ \%T = \{(root, LEAF)\};
\end{array}
$$

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{Pre}(target, targetClass, arg, argClass; \\
\qquad\qquad\qquad \%tIntFsPre, \%tObjFs, \%cs, \%T) := \\
\ \exists\ targetClassId, \%methodNames, \%vMs. \\
\quad \$\mathrm{Object}(target, targetClass; \%tIntFsPre, \%tObjFs) \\
\quad \star\ \ \$\mathrm{Tree}(arg, argClass; \%T) \\
\quad \star\ \ \$\mathrm{Meta}(; \%cs) \star \mathrm{proj}_2(\%T) \subseteq \mathrm{proj}_2(\%cs) \\
\quad \star\ \ (targetClassId, targetClass, \%vMs) \in \%cs \\
\quad \star\ \ \$\mathrm{AppendStringAll}(VISIT; \mathrm{proj}_2(\%T), \%methodNames) \\
\quad \star\ \ \$\mathrm{MethodsExist}(; \%methodNames, \%vMs);
\end{array}
$$

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{Post}(target, targetClass, arg, argClass; \\
\qquad\qquad\qquad \%tIntFsPre, \%tObjFs, \%cs, \%T) := \\
\ \exists\ \%tIntFsPost. \\
\quad \$\mathrm{Object}(target, targetClass; \%tIntFsPost, \%tObjFs) \\
\quad \star\ \ \$\mathrm{Tree}(arg, argClass; \%T) \\
\quad \star\ \ \$\mathrm{Meta}(; \%cs) \\
\quad \star\ \ \$\mathrm{Fun}(; \%T, \%tIntFsPre, \%tIntFsPost);
\end{array}
$$

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{MethodsExist}(; \%nameXtype, \%ms) := \%nameXtype = \emptyset \\
\vee\ \ \exists\ \%rest, a, name, type. \\
\quad \%nameXtype = \{(name, type)\} \cup \%rest \\
\quad \star\ \ (a, name, \{(1, COUNTVISITOR)\} \cup \{(2, type)\}) \in \%ms \\
\quad \star\ \ \$\mathrm{MethodsExist}(; \%rest, \%ms);
\end{array}
$$

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{Fun}(; \%T, \%pre, \%post) := \\
\ \exists\ m, n. \\
\quad \$\mathrm{TreeSize}(n, \%T) \star \%pre = \{(1, m)\} \star \%post = \{(1, m+n)\};
\end{array}
$$

$$
\begin{array}{l}
\mathsf{recdef}\ \$\mathrm{TreeSize}(n; \%T) := \\
\ n = 0\ \star\ \ \%T = \emptyset \\
\vee\ \ \exists o, \%rest.\ \%T = \{(o, LEAF)\} \cup \%rest \star (o, LEAF) \notin \%rest \\
\quad \star\ \ PARENT \notin \mathrm{proj}_2(\%rest) \star \$\mathrm{TreeSize}(n-1; \%rest) \\
\vee\ \ \exists o, \%rest.\ \%T = \{(o, PARENT)\} \cup \%rest \\
\quad \star\ \ (o, PARENT) \notin \%rest \star \$\mathrm{TreeSize}(n; \%rest);
\end{array}
$$

**Figure 7: Predicates instantiated for visitor example**

including the extra pointers in the tuples, although it is less concise. Alternatively one could require that $\$\mathrm{Meta}(\%cs)$ is immutable. This approach, along with a discussion of the typical loss of precision or conciseness when describing elements that do not change, is discussed and developed further

$$
\begin{array}{l}
\mathsf{proc}\ Object\_getClass(obj, res) \\
\ \forall\ cName, \%intFs, \%objFs, \%cs. \\
\ \mathsf{pre} : \$\mathrm{Object}(obj, cName; \%intFs, \%objFs) \star res \mapsto \_ \\
\qquad \star\ \ \$\mathrm{Meta}(; \%cs) \star cName \in \mathrm{proj}_2(\%cs); \\
\ \mathsf{post} : \exists cPtr, \%cMs. \\
\qquad \$\mathrm{Object}(obj, cName; \%intFs, \%objFs) \star res \mapsto cPtr \\
\qquad \star\ \ \$\mathrm{Meta}(; \%cs) \star (cPtr, cName, \%cMs) \in \%cs;
\end{array}
$$

$$
\begin{array}{l}
\mathsf{proc}\ Class\_getName(cPtr, res) \\
\ \forall\ \%cs, cNm, \%cMs. \\
\ \mathsf{pre} : \$\mathrm{Meta}(; \%cs) \star (cPtr, cNm, \%cMs) \in \%cs \star res \mapsto \_; \\
\ \mathsf{post} : \$\mathrm{Meta}(; \%cs) \star (cPtr, cNm, \%cMs) \in \%cs \star res \mapsto cNm;
\end{array}
$$

$$
\begin{array}{l}
\mathsf{proc}\ Class\_getMethod(cPtr, mName, res) \\
\ \forall\ cName, \%cMs, \%cs. \\
\ \mathsf{pre} : \$\mathrm{Meta}(; \%cs) \star (cPtr, cName, \%cMs) \in \%cs \star res \mapsto \_; \\
\ \mathsf{post} : \exists mPtr, \%ps.\ \$\mathrm{Meta}(; \%cs) \star (cPtr, cName, \%cMs) \in \%cs \\
\qquad \star\ \ res \mapsto mPtr \star (mPtr, mName, \%ps) \in \%cMs \\
\quad \vee\ \ \$\mathrm{Meta}(; \%cs) \star (cPtr, cName, \%cMs) \in \%cs \\
\qquad \star\ \ res \mapsto 0 \star mName \notin \mathrm{proj}_2(\%cMs);
\end{array}
$$

$$
\begin{array}{l}
\mathsf{proc}\ Method\_invoke(method, target, arg) \\
\ \forall\ targetClass, argClass, \%intFsPre, \%objFs, \%cs, \%rest, name, \\
\quad \%param\_types, targetTypeId, \%targetClsMs. \\
\ \mathsf{pre} : \$\mathrm{Pre}(target, targetClass, arg, argClass; \\
\qquad\qquad\qquad \%intFsPre, \%objFs, \%cs, \%rest) \\
\quad \star\ \ (targetTypeId, targetClass, \%targetClsMs) \in \%cs \\
\quad \star\ \ (method, name, \%param\_types) \in \%targetClsMs \\
\quad \star\ \ \%param\_types = \{(1, targetClass)\} \cup \{(2, argClass)\}; \\
\ \mathsf{post} : \$\mathrm{Post}(target, targetClass, arg, argClass; \\
\qquad\qquad\qquad \%intFsPre, \%objFs, \%cs, \%rest);
\end{array}
$$

**Figure 8: Specification of reflective library**

in [5].

**Object_getClass** The specification states that the first argument is an object, and the second argument is the address of a heap cell, where the result will be stored. Additionally, we require that the metadata is available, and that the object's class appears within the metadata. The result of calling this procedure is that we get the class's ID *cPtr* (a pointer in the metadata) stored in the result cell together with the *assertion* that this is the case.

**Class_getName** This procedure simply looks up the given class identifier (key) in the metadata, and stores the contained name (string encoded as integer) in the result cell.

**Class_getMethod** This procedure *tries* to find the method with the given name (string as integer), contained within the given class. If successful, the address of its metadata is stored in the result cell. Otherwise, 0 is stored and we know that there is no method of that name in the given class.

**Method_invoke** In essence this is just a wrapper around an eval command. Therefore, its specification is almost like the

```
proc $Class\_getMethod(cPtr, mName, res)$ {
    locals $methodList$;
    // Class's method list is the second record of metadata
    $methodList := [cPtr + 1]$;
    // Search the methods
    call $searchMethodList(methodList, mName, res)$;
}
```

**Figure 9: Code for reflective library's getMethod**

generic specification 𝔊𝔢𝔫𝔖𝔭𝔢𝔠 we saw earlier. Additionally the call protocol must be ensured. This is done by checking in the precondition that the given object arguments are of the classes expected by the method being invoked. The precondition also ensures that the method exists, and the current state includes all the (well-defined w.r.t. classes) objects that might be needed.

## 4.3 Implementation

The implementation uses the metadata on the heap in the obvious manner, reading the desired records and traversing the linked lists as necessary. The implementation of the reflective library is thus not particularly interesting (the interesting part is the modelling of metadata) hence we show only the code of library method Class_getMethod in Fig. 9. We have omitted the implementation of the auxiliary procedure searchMethodList(*methodList*, *mName*, *res*) that searches for the method *mName* in list *methodList* and writes into cell *res* a pointer to the matching element, or 0 otherwise. This procedure as well as all other details can be found in the full version online.

## 5. REFLECTIVE VISITOR EXAMPLE

Let us return to the concrete reflective visitor from Fig. 2 (Section 2) that counts the number of leaves in a tree made from Parent and Leaf objects. We discuss the implementation and specification. We need not say much about the verification as this has been carried out formally with our semi-automatic verification tool Crowfoot[4] [4].

## 5.1 Implementation

The code for the reflective visitor uses the reflective library from the previous section. Fig. 3 shows the implementation of the generic visit method. It demonstrates how the reflective features are used to invoke the relevant concrete visit method, based on the type of the object it receives. These concrete visit methods are shown in Fig. 10 and are a simple translation from the Java-style version in Fig. 2.

## 5.2 Specification

In the tradition of Smallfoot [2] we would like to prove memory safety. In a closed world setting, one would know in advance what the classes are going to be and write the specification accordingly. A more generally applicable version of the specification may assume that the methods the reflective visitor might call are actually loaded at runtime, whatever

[4]This tool was developed for reasoning about code pointers in general.

```
proc $CountVisitor\_visitLeaf(this, l)$ {
    locals $currentVal$;
    $currentVal := [this + 1]$;
    $[this + 1] := currentVal + 1$;
}

proc $CountVisitor\_visitParent(this, p)$ {
    locals $left, right$;
    // Visit left subtree;
    $left := [p + 1]$;    call $CountVisitor\_visit(this, left)$;
    // Visit right subtree
    $right := [p + 2]$;    call $CountVisitor\_visit(this, right)$;
}
```

**Figure 10: Implementations of Leaf/Parent node visitors, translated into Crowfoot language.**

they are. Such a specification does not need to be changed if more classes are added to the underlying tree type which is very much in the spirit of the reflective visitor the code of which does not need to change either. On top of this we can also prove the functional specification for the concrete CountVisitor, i.e. that it computes the number of leaves of the tree it visits. In the following we discuss and refer to this functionally full specification. We do all this in a style that is re-usable for other visitor examples, as explained in Sec. 5.3.

When the metadata for visitor classes is generated, the code for the visitor methods will appear in \$MethodLseg. Therefore, *all of the specifications of visitor methods must entail the generic specification* 𝔊𝔢𝔫𝔖𝔭𝔢𝔠 *seen in the* \$MethodLseg *predicate definition.* Note that we don't expect methods that aren't reflectively invoked to appear in \$MethodLseg.

The simplest method in our visitor is visitLeaf which only needs the current visitor object, and the leaf node being visited will be the single object within the tree structure. Its specification can be found in Fig. 11. Note that it does not need to mention the metadata. One can, however, show that it entails the generic specification 𝔊𝔢𝔫𝔖𝔭𝔢𝔠 using the frame rule to add on the metadata. The specifications for visitParent and the generic visit are almost identical to the generic specification 𝔊𝔢𝔫𝔖𝔭𝔢𝔠 (they use different instances of *targetClass* and *argClass*).

## 5.3 Reusability

As alluded to earlier, the arrangement of the predicates is such that it should be possible to reuse the model for different examples. Firstly if one would like to verify code for a different visitor, say one that counts *all* nodes and not just leaves, it will often be enough to just change the definition of \$Fun (and its dependent \$TreeSize) to reflect the new *effect of the visitor.* Another case of visitor reuse might add an *effect on the tree structure* during the traversal (perhaps removing some nodes). In this case, it would be necessary to modify the \$Post definition to account for the structural change. This could be achieved by allowing the \$Tree occurrence to refer to an existential $\%newT$, and define and in-

```
proc $CountVisitor\_visitLeaf(this, l)$
 $\forall$ $\%intFsPre, \%objFs, \%T.$
  pre : $\$\mathrm{Object}(this, COUNTVISITOR; \%intFsPre, \%objFs)$
   $\star$ $\$\mathrm{Tree}(l, LEAF; \%T);$
  post : $\exists$ $\%intFsPost.$
   $\$\mathrm{Object}(this, COUNTVISITOR; \%intFsPost, \%objFs)$
   $\star$ $\$\mathrm{Tree}(l, LEAF; \%T) \star \$\mathrm{Fun}(\%T, \%intFsPre, \%intFsPost);$

proc $CountVisitor\_visitParent(this, p)$
 $\forall$ $\%intFsPre, \%objFs, \%cs, \%T.$
  pre : $\$\mathrm{Pre}(this, COUNTVISITOR, p, PARENT;$
                 $\%intFsPre, \%objFs, \%cs, \%T);$
  post : $\$\mathrm{Post}(this, COUNTVISITOR, p, PARENT;$
                 $\%intFsPre, \%objFs, \%cs, \%T);$

proc $CountVisitor\_visit(this, obj)$
 $\forall$ $\%intFsPre, \%objFs, \%cs, \%T, type.$
  pre : $\$\mathrm{Pre}(this, COUNTVISITOR, obj, type;$
                 $\%intFsPre, \%objFs, \%cs, \%T);$
  post : $\$\mathrm{Post}(this, COUNTVISITOR, obj, type;$
                 $\%intFsPre, \%objFs, \%cs, \%T);$
```

**Figure 11: Specifications of visit methods**

clude a new predicate that describes the relation of $\%newT$ and the original $\%T$. Finally, if a program wants to call Method_invoke, but not in the spirit of the visitor pattern, then \$Pre and \$Post have to be given completely different definitions.

At each of these levels, it is possible to make use of a case analysis on the class arguments (*argClass* and *targetClass*) in the \$Fun[5], \$Pre and \$Post predicates. This would mean that the same program can use Method_invoke in different ways, according to the signature (class types) of the method invoked. By structuring the proof in a way that mimics the abstract predicate families of Parkinson et al. [13] (which, after all, are predicates with an extra class argument), we can account for class types that are not known in advance.

## 5.4 Verification

*We have carried out the entire verification described in this paper (library and visitor example) in our prover Crowfoot* [1, 4] which is an automatic verifier that needs the occasional hint from the user. These hints usually relate to dealing with the folding/unfolding of predicate instances, but in other cases it is necessary to apply lemmas that go beyond the capability of the SMT solver. Like in VeriFast [10] we can use procedures to encode lemmas. Since our verifier is (semi-)automatic, however, it is often enough to have code bodies containing precisely a skip statement, in which case our lemmas are verified automatically.

[5]For simplicity of our example, \$Fun doesn't take a class argument in our definition, though it would be trivial to add this.

```
proc $Class\_getMethod(cPtr, mName, res)$ {
  locals $methodList;$
  ghost "unfold $\$\mathrm{Meta}(;?)$";
  // Use cPtr to expose particular element in list
  ghost "split $\$\mathrm{ClassLseg}$ $list1$ $(cPtr, cName, \%cMs)$";
  // Class's method list is the second record of metadata
  $methodList := [cPtr + 1];$
  // Search the methods
  call $searchMethodList(methodList, mName, res);$
  ghost "fold $\$\mathrm{ClassLseg}(cPtr, 0; ?)$";
  ghost "join $\$\mathrm{ClassLseg}$ $list1$";
  ghost "fold $\$\mathrm{Meta}(;?)$"
}
```

**Figure 12: Annotated version of getMethod**

To demonstrate the use of the hints for unfolding/folding predicates, Fig. 12 contains the code for Class_getMethod as seen in Sec. 4.3, but this time with the additional annotations that are necessary for verification. These annotations are provided as ghost statements, which are unfolding (or folding) instances of predicates as defined in Fig. 5. Unfolding of the instance of predicate \$Meta in Fig. 12 allows the dereferencing of the address *cPtr* that is otherwise hidden deep inside \$Meta. Using the question mark in place of arguments will cause the prover to attempt to guess the correct variables to use. A special case of unfolding is the split, which will split a list segment at a certain position into three parts: a prefix segment, the "sought" element, and a suffix segment. In Fig. 12 the corresponding ghost statement splits a \$ClassLseg list segment *list1* which is a fresh name for the existential variable *list* that was produced from unfolding the \$Meta predicate. The additional argument ($cPtr, cName, \%cMs$) describes the "sought" element, i.e. the position where the list is split.

## 6. RELATED WORK

The Java Modeling Language (JML) is a popular specification language for Java [11] supported by several state-of-the-art static checkers and verifiers [7]. As part of the JML project many of the key Java APIs have been annotated with JML specifications including the reflection library on which the program in this paper is based. However, these JML specifications are generally weaker than ours. For the Object.getClass method, the specification is equivalent to what we have. However for Class.getName, the specification does not appear to relate the returned string to the class's name. The Class.getMethod specification simply says there are no side-effects, without asserting anything about the return value. For Method.invoke, [11] suffers from a couple of weaknesses. Firstly, the precondition merely requires that the target object is not null (for non-static methods). Secondly, the post-condition only really applies to primitive types, saying that if the method returns a primitive, then an object in the equivalent wrapper class is actually returned. We, however, ensure that the invoke method *will not* fail, because we assert that the method exists and the argument types are correct. We can do this because we have available the loaded classes in the current state.

## 7. CONCLUSION AND FUTURE WORK

We have devised, implemented and specified a Java-like reflective library, along with an instance of a reflective visitor pattern. This has been successfully verified with our tool. The process has highlighted some of the weaknesses of our prover, demonstrating areas for improvement in the instantiation of quantified variables, cases where our entailment rules needed extending, and limitations of sending our complex assertions to the SMT-solvers. Some of these limitations have been addressed by sacrificing automation (details have been omitted for lack of space), and it is future work to try to restore the automation where possible. As well as enhancing our own prover, something to consider is whether we can implement a similar logic in another prover, for instance one that has built in support for object-oriented programming.

In contrast to Java, we presently have to manually set up the metadata for our program, in a programmatic way. Ideally we would like to automate this generation of metadata. Such an extension should generally be straight-forward, turning a parsed program into a concrete instance of the $Meta predicate, however there is an issue of strings to take into account because our language only has integers. Presently we use constants to represent strings, and manually associate a constant with a method when setting up the metadata. If we want the process to be automated, we would need to add strings to the Crowfoot language such that the names of methods can be directly converted and used in the logic. This will naturally require an implementation of the encoding of strings so that they can be passed to the SMT-solver that is charged with deciding pure entailments.

Another step to make our approach more closely relate to Java's reflection, would be to represent metadata in object form (i.e. have Class and Method meta-objects). Java also offers more reflective features which could be supported in future, perhaps the most obvious being introspection on fields. This would allow us to verify the Walkabout from [12].

Finally, future work could investigate how to systematically translate reflective Java programs into programs written in a language like the one in Sec. 3.1, where the reflective operations are implemented in terms of ordinary statements and data structures. Such a translation could perhaps be proved correct using a bisimulation argument as used in [9].

## 8. ACKNOWLEDGMENTS

We acknowledge the support of EPSRC grant *"From Reasoning Principles for Function Pointers To Logics for Self-Configuring Programs"* (EP/G003173/1), and would also like to thank the anonymous referees for their suggestions, in particular for spotting a possible simplification of $Pre.

## 9. REFERENCES

[1] The Crowfoot website, 2011. `www.sussex.ac.uk/informatics/crowfoot`.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[3] Jeremy Blosser. Java tip 98: Reflect on the visitor design pattern. *JavaWorld*, 2000. `http://www.javaworld.com/javaworld/javatips/jw-javatip98.html`.

[4] Nathaniel Charlton, Ben Horsfall, and Bernhard Reus. Crowfoot: A verifier for higher-order store programs. In *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2012.

[5] Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. In *OOPSLA*, pages 359–374, 2011.

[6] Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.

[7] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[8] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[9] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Proceedings of the International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, January 2012.

[10] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, pages 41–55, 2011.

[11] Gary Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel Zimmerman, and Werner Dietl. JML reference manual (draft v1.235). Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, July 2008.

[12] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC*, pages 9–15, 1998.

[13] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86, 2008.

[14] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[15] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare triples and frame rules for higher-order store. In *CSL*, pages 440–454, 2009.

[16] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare triples and frame rule for higher-order store. *Logical Methods in Computer Science*, 7(3), September 2011.