Formal Verification of `MIX` Programs

Jean-Christophe Filliâtre
CNRS
LRI, Univ Paris-Sud, Orsay F-91405
INRIA Futurs, ProVal, Orsay F-91893

Abstract

We introduce a methodology to formally verify `MIX` programs. It consists in annotating a `MIX` program with logical annotations and then to turn it into a set of purely sequential programs on which classical techniques can be applied. Contrary to other approaches of verification of unstructured programs, we do not impose the location of annotations but only the existence of at least one invariant on each cycle in the control flow graph. A prototype has been implemented and used to verify several programs from *The Art of Computer Programming*.

1 Introduction

`MIX` is a machine introduced by D. Knuth in *The Art of Computer Programming* [5] and equipped with an assembly language. If it looks outdated compared to today's computers, it is nonetheless a language with a clearly exposed semantics and the vehicle of many algorithms described in this set of books. It is thus quite natural to consider proving such programs in a formal way. Moreover, the techniques involved in the verification of assembly-like programs have direct applications in domains such as *Proof Carrying Code* [8] or safety of low-level C programs.

Our approach to formally verifying `MIX` programs is built on previous works on the verification of C and Java programs [3]. The general idea is to specify a program using logical annotations inserted in its source code, and then to translate the program into an intermediate language suitable for Hoare logic [4]. The output is a set of *verification conditions*, that is a set of logical formulae whose validity implies the correctness of the program. Discharging the verification conditions can be done using existing theorem provers, either automatic or interactive.

The formal specification of structured programs is naturally done using pre- and postconditions for functions and loop invariants. Assembly programs, on the contrary, do not favor any particular program point for logical assertions. Recent work has identified the entry points of natural loops as obvious candidates for such assertions [1], but our first experiments show that it can be a constraint. This is why we choose an approach where assertions can be freely inserted at any program point. We only impose that any cycle in the control flow graph contains at least one assertion. Then it is possible to turn a `MIX` program into a set of purely sequential programs without jumps (either conditional or unconditional). In a second step, these programs are interpreted in a model where it is possible to apply the usual techniques of Hoare logic.

Section 2 introduces our methodology. Section 3 describes a prototype and our first experiments. We conclude with possible future work.

```
X        EQU   1000
         ORIG  3000
MAXIMUM  STJ   EXIT
                          <-- Pre
INIT     ENT3  0,1
         JMP   CHANGEM
LOOP     CMPA  X,3
         JGE   *+3
CHANGEM  ENT2  0,3
         LDA   X,3
         DEC3  1
                          <-- Inv
         J3P   LOOP
                          <-- Post
EXIT     JUMP  *
```

Figure 1: Program M (TAOCP, vol. 1, page 145).

2 Methodology

We first show how to annotate a `MIX` program using logical assertions (section 2.1). Then we give an algorithm to turn this program into a set of purely sequential programs (section 2.2). Finally we explain how to get verification conditions from these sequential programs (section 2.3) and we prove the soundness of our method (section 2.4).

2.1 Specification

Let us consider the first program illustrating `MIX`, namely program M finding the maximum of the elements of an array [5, page 145]. This program is given Figure 1. It assumes that the size of the array is in register I_1, that the array contains at least one element, and that the elements are stored in memory at addresses $\mathtt{X}+1, \ldots, \mathtt{X}+I_1$. When the program return, accumulator A contains the maximum element and register I_2 an index where it appears. The first step consists in specifying this behavior by inserting logical annotations in program M. There are three such annotations:

- a *precondition* indicating the assumption at the program entry point, namely

$$\mathit{Pre} \equiv I_1 \geq 1$$

  inserted at program point `MAXIMUM`;

- a *postcondition* indicating the expected property at the end of execution, namely the annotation

$$\mathit{Post} \equiv 1 \leq I_2 \leq I_1 \wedge A = \mathtt{X}[I_2] \wedge \forall i,\, 1 \leq i \leq I_1 \Rightarrow A \geq \mathtt{X}[i]$$

  inserted a program point `EXIT`;

- and an *invariant* indicating the property maintained by the program, namely the annotation

$$\mathit{Inv} \equiv 0 \leq I_3 \leq I_1 \wedge 1 \leq I_2 \leq I_1 \wedge A = \mathtt{X}[I_2] \wedge \forall i,\, I_3 < i \leq I_1 \Rightarrow A \geq \mathtt{X}[i]$$

  inserted right before instruction `J3P`.

These three annotations are indicated on Figure 1. The meaning of an annotation is clear: each time execution reaches an annotation, it must be verified.

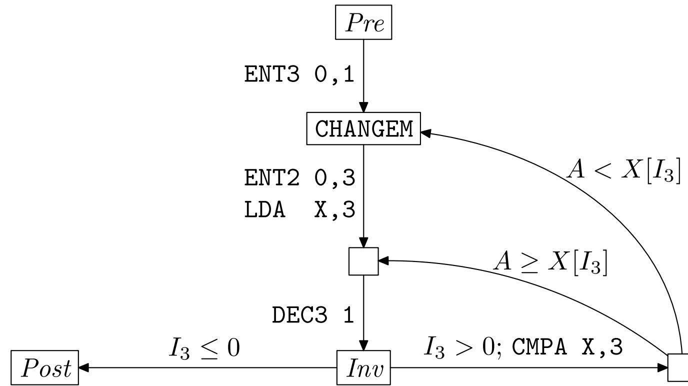

Figure 2: Control flow graph for program M.

traverse(n) =
  if n is currently visited then fail (we found a cycle with no invariant)
  if n has not yet been visited then
    for each transition $n \xrightarrow{s} m$
      call traverse(m)
      if m is an invariant I then
        associate to n the code s; `assert` I
      else
        associate to n the code s; s' for each code s' associated to m
    if n is an invariant J the prefix each code of m by `assume` J

Figure 3: Sequentialization algorithm.

2.2 Sequentialization

The next step is to turn the `MIX` program into a set of purely sequential programs which do not contain jumps anymore, and whose correctness imply the correctness of the initial program. To do so, we first build the control flow graph, where nodes correspond to program points and transitions to sequences of instructions and test results. Such a control flow graph for program M is given Figure 2.

The key idea is to impose the presence of (at least) one annotation on each cycle in the control flow graph. Annotations are considered as *invariants* here: they must be verified when reached initially and maintained by any path used to come back. (In the remainder of this section we will refer to annotations as *invariants* instead of *assertions* to make it clearer.) Our sequentialization algorithm simply performs a depth-first traversal of the graph from the entry point, associating to each node a set of purely sequential programs. Pseudo-code for this algorithm is given Figure 3.

The result of sequentialization is the set of programs associated to the entry point and to each invariant encountered during the traversal. The result for program M is the set of four codes, given Figure 4. Keyword `assume` introduces an assumption and keyword `assert` a property to be verified. These programs are naturally interpreted as follows: $\texttt{seq}_1$ is the initial validity of the invariant *Inv*; $\texttt{seq}_2$ and $\texttt{seq}_3$ express the preservation of *Inv* on the two possible paths; finally $\texttt{seq}_4$ expresses the validity of the postcondition.

It is important to notice that this algorithm is not related to `MIX` but could be used with any assembly-like language.

```
seq1 ≡
  assume Pre
  ENT3 0,1
  ENT2 0,3
  LDA X,3
  DEC3 1
  assert Inv
```

```
seq2 ≡
  assume Inv
  assume I3 > 0
  CMPA X,3
  assume CMP ≥ 0
  DEC3 1
  assert Inv
```

```
seq3 ≡
  assume Inv
  assume I3 > 0
  CMPA X,3
  assume CMP < 0
  ENT2 0,3
  LDA X,3
  DEC3 1
  assert Inv
```

```
seq4 ≡
  assume Inv
  assume I3 ≤ 0
  assert Post
```

Figure 4: Sequentializing program M.

2.3 Generating the Verification Conditions

The last step consists in generating verification conditions from the sequential programs *i.e.* logical formulae whose validity implies the correctness of the initial program. For this purpose, we can use traditional Hoare logic [4], or more conveniently a calculus of weakest preconditions [2], provided some logical model of `MIX` programs:

- registers $A, X, I_1, \dots I_6$ are modeled as global integer variables;
- the memory is modeled as a global array of integers;
- flags E, G and L are interpreted as the sign of a unique global variable CMP.

Furthermore, we make the following assumptions:

- direct use of register J is not supported (only the co-routine pattern is recognized and handled separately);
- input-output instructions are not modeled: they can be used in programs but the user has to state assumptions about data which is read.

2.4 Soundness

In this section we prove the soundness of our method. Let us write $\mathcal{S} = \{seq_1, \dots, seq_k\}$ the set of purely sequential programs resulting from the sequentialization. Each seq_i has the following shape

$$\begin{array}{l}\texttt{assume } P_i \\ s_i \\ \texttt{assert } Q_i\end{array}$$

where P_i and Q_i are user invariants and s_i is a purely sequential program which does not contain any invariant (but with possible `assume` declarations corresponding to test results).

Let us assume an operational semantics for purely sequential `MIX` programs, as a set of states and a transition relation between states written $S_1 \xrightarrow{s} S_2$ which means "execution of program s in state S_1 leads to state S_2". We note $S \models I$ if invariant I holds in state S. The correctness of each program seq_i means that in any state S_1 such that $S_1 \models P_i$ and for any state S_2 such that $S_1 \xrightarrow{s_i} S_2$ we have $S_2 \models Q_i$. Note that we are only considering partial correctness.

Soundness can be stated as follows:

Theorem 1 (soundness) *Let S be a state satisfying the invariant I at entry point,* i.e. $S \models I$, *and let us consider an execution reaching a program point with an invariant J in state S'. Then $S' \models J$.*

Proof. Let us consider all the intermediate states where the execution reaches an invariant. By definition of the sequentialization algorithm, there is a finite number of steps between two such states (otherwise we would have a cycle in the control flow graph without any invariant). Therefore the execution looks like

$$S = S_0 \xrightarrow{s_1} S_1 \xrightarrow{s_2} S_2 \ldots \xrightarrow{s_n} S_n = S'$$

where each state S_i is associated to an invariant I_i, with $I_0 = I$ and $I_n = J$. Each triple (I_{i-1}, s_i, I_i) exactly corresponds to one sequential program in $\mathcal{S}$, since invariant I_{i-1} can be reached in the control flow graph from the entry point, and since there is path from I_{i-1} to I_i in this graph which does not cross any other invariant. Then proving $S_i \models I_i$ is a straightforward induction, since $S_0 \models I_0$ by assumption and since the correctness of the sequential programs implies that if $S_{i-1} \models I_{i-1}$ and $S_{i-1} \xrightarrow{s_i} S_i$ then $S_i \models I_i$. □

3 Implementation

3.1 Prototype

We implemented our method as a prototype tool, called demixify, taking annotated MIX programs as input and generating verification conditions in the native syntax of several theorem provers. Our tool uses the back-end of the Why platform [3], which provides an intermediate language dedicated to program verification. Thus our prototype is simply a front-end which parses annotated MIX programs, performs the sequentialization and then prints the resulting programs in the syntax of the Why tool. Then we rely on the Why tool to compute the verification conditions and to dispatch them to interactive provers (such as Coq, PVS, Isabelle/HOL, etc.) or automatic provers (Simplify, Ergo, Yices, etc.).

In practice, demixify is run on a file `file.mix` to produce a Why file `file.why`. Then it is possible to use tools from the Why platform on this file (to compute the verification conditions, display them, launch provers, display the results, etc.). Inside `file.mix` annotations are inserted between brackets. One pair of brackets { P } stands for an assertion to be verified at the corresponding program point and double brackets {{ P }} stands for an invariant. A quotation mechanism allows to insert pure Why code inside the input file, using the syntax {{{ *why code* }}}. This is typically used to insert a prelude containing parameters for the program or logical declarations for its specification.

Within annotations, the current values of registers is referred to using the names A, I_1, I_2, etc. The current value at address p in memory is denoted by $mem[p]$.

3.2 Case Studies

Up to now, we only verified very simple MIX programs, such as program M above (which is verified fully automatically). We briefly describe some of these case studies in this section.

3.2.1 Sequential Searching

We consider here three implementations of sequential searching from Section 6.1 of *The Art of Computer Programming* [7, page 397]. The goal is to check whether a given value stored at address K appears in an array of N values stored at addresses KEY + 1, . . . , KEY + N.

First, we introduce KEY, K and N as parameters, using the quotation for Why code:

```
{{{ logic KEY,K,N:int }}}
```

The first implementation (Program S page 397) simply scans the array from index 1 to index N. It can be annotated as follows:

```
start:  {{ N ≥ 1 }}
        lda K
        ent1 1-N
2H:     {{  1 ≤ N + I1 ≤ N ∧ A = mem[K] ∧
            ∀i, 1 ≤ i < N + I1 ⇒ mem[K] ≠ mem[KEY + i]  }}
        cmpa KEY+N,1
        je success
        inc1 1
        j1np 2B
failure:{ ∀i, 1 ≤ i ≤ N ⇒ mem[K] ≠ mem[KEY + i] }
        hlt
success:{ mem[K] = mem[KEY + N + I1] }
```

The postcondition is split into two assertions at labels `failure` and `success`, respectively. The loop invariant is here located at the loop entry point (label `2H`). It may seem unnecessary to add $A = mem[\texttt{K}]$ in the loop invariant, since A and `K` are not modified in the loop body, but there is currently no mechanism to get such invariant for free. When demixify is run on this program, it generates 10 verification conditions (when splitting conjunctions), all discharged automatically using an automatic theorem prover such as Simplify or Ergo.

The next implementation (Program Q page 397) improves on the previous one by setting a sentinel at address $\texttt{KEY} + \texttt{N} + 1$, namely the value to be searched for. The annotated code is as follows:

```
start:  {{ N ≥ 1 }}
        lda K
        sta KEY+N+1
        ent1 -N
        inc1 1
        {{  1 ≤ N + I1 ≤ N + 1 ∧ A = mem[K] ∧ mem[KEY + N + 1] = A ∧
            ∀i, 1 ≤ i < N + I1 ⇒ mem[K] ≠ mem[KEY + i]  }}
        cmpa KEY+N,1
        jne *-2
        j1np success
failure:{ ∀i, 1 ≤ i ≤ N ⇒ mem[K] ≠ mem[KEY + i] }
        hlt
success:{ mem[K] = mem[KEY + N + I1] }
```

The specification is exactly the same, apart from the addition of a specification for the sentinel in the loop invariant, namely $mem[\texttt{KEY} + \texttt{N} + 1] = A$. Note that the loop invariant is placed one instruction *after* the loop entry point (which is located here right *before* instruction `inc1 1`). As we already did with program M, we notice that it is convenient to have freedom in invariant locations. For this second implementation, 12 verification conditions are generated and all are automatically discharged.

The last improvement consists in unrolling the loop once, in order to save one increment (Program Q' page 398). The specification is exactly the same as for program Q:

```
start:  {{ N ≥ 1 }}
        lda K
```

```
        sta KEY+N+1
        ent1 -1-N
3H:     inc1 2
```

$\{\{\ \ 1 \le \mathtt{N} + I_1 \le \mathtt{N} + 1 \land A = mem[\mathtt{K}] \land mem[\mathtt{KEY} + \mathtt{N} + 1] = A \land$
$\quad \forall i,\ 1 \le i < \mathtt{N} + I_1 \Rightarrow mem[\mathtt{K}] \ne mem[\mathtt{KEY} + i]\ \ \}\}$

```
        cmpa KEY+N,1
        je 4F
        cmpa KEY+N+1,1
        jne 3B
        inc1 1
4H:     j1np success
```

`failure:` $\{\ \forall i,\ 1 \le i \le \mathtt{N} \Rightarrow mem[\mathtt{K}] \ne mem[\mathtt{KEY} + i]\ \ \}$

```
        hlt
```

`success:` $\{\ \ mem[\mathtt{K}] = mem[\mathtt{KEY} + \mathtt{N} + I_1]\ \ \}$

Again the loop invariant is not placed at the loop entry point but immediately after the increment instruction (`inc1 2` here). Running demixify on program Q' results in 14 verification conditions. All are automatically discharged, *but one*. Not surprisingly, this is the preservation of the property

$$\forall i,\ 1 \le i < \mathtt{N} + I_1 \Rightarrow mem[\mathtt{K}] \ne mem[\mathtt{KEY} + i]$$

when I_1 is incremented by 2, *i.e.* after two successive negative tests. Thus the proof requires *two* case analyzes to distinguish between $1 \le i < \mathtt{N} + I_1$, $i = \mathtt{N} + I_1$ and $i = \mathtt{N} + I_1 + 1$. It seems to be out of reach of the heuristics involved in the automatic theorem provers we are using.

3.2.2 Selection Sort

Our last case study is a proof of *straight selection sort* [7, Program S page 140]. The purpose of this program is to sort in place the values stored at locations $\mathtt{INPUT} + 1, \ldots, \mathtt{INPUT} + \mathtt{N}$, in increasing order. We start a Why prelude with the declaration of parameters `N` and `INPUT`:

```
{{{
  logic N,INPUT:int
```

Next we introduce predicate definitions to simplify annotations. The first such predicate is the main invariant of selection sort, which states that the upper part of the array $\mathtt{INPUT} + i, \ldots, \mathtt{INPUT} + \mathtt{N}$ is already sorted and that all values in the lower part are smaller than values in the upper part:

```
  predicate Inv(a: int array, i: int) =
    sorted_array(a, INPUT+i, INPUT+N) and
    forall k,l:int. 1 <= k <= i-1 -> i <= l <= N -> a[INPUT+k] <= a[INPUT+l]
```

Here we use a predefined predicate `sorted_array` from Why's standard library. The second predicate is the permutation property. Indeed, we not only need to prove that the final array is sorted but also that it is a permutation of the initial. For this purpose we introduce an auxiliary variable `mem0` representing the initial contents of the memory and we use a predefined predicate `sub_permut` from Why's standard library:

```
  logic mem0: int farray
  predicate Perm(m: int array) = sub_permut(INPUT+1, INPUT+N, m, mem0)
}}}
```

We are now in position to annotate the MIX code for selection sort:

```
init:  {{ N ≥ 2 ∧ Perm(mem) }}
       ent1 N-1
2H:    {{ 1 ≤ I1 ≤ N − 1 ∧ Inv(mem, I1 + 2) ∧ Perm(mem) }}
       ent2 0,1
       ent3 1,1
       lda INPUT,3
8H:    {{  1 ≤ I1 ≤ N − 1 ∧ Inv(mem, I1 + 2) ∧ Perm(mem) ∧
           1 ≤ I2 < I3 ≤ I1 + 1 ∧ A = mem[INPUT + I3] ∧
           ∀k, I2 + 1 ≤ k ≤ I1 + 1 ⇒ mem[INPUT + k] ≤ mem[INPUT + I3] }}
       cmpa INPUT,2
       jge *+3
       ent3 0,2
       lda INPUT,3
       dec2 1
       j2p 8B
       ldx INPUT+1,1
       stx INPUT,3
       sta INPUT+1,1
       dec1 1
       j1p 2B
       { sorted_array(mem, INPUT + 1, INPUT + N) ∧ Perm(mem) }
```

This annotated code generates 42 verification conditions, all discharged automatically.

3.2.3 Other Case Studies

A proof of algorithm I (inversion of a permutation in place) is already engaged. We could also consider the formal proof of Knuth's algorithm for the first N prime numbers performed by L. Théry a few years ago [9]: the proof was done using Why and Coq directly and we could turn it into a proof of the original MIX code.

4 Conclusion and Future Work

We have presented a methodology for the formal verification of MIX programs. A prototype has been implemented and the first experiments are encouraging. There are many possible extensions for this work. One improvement would be the ability to prove the termination of MIX programs, by addition of *variants* on each cycle of the control flow graph, as we did for invariants. Another interesting extension would be formal reasonings regarding the complexity of MIX programs, since one of MIX's main interests is precisely to allow a detailed complexity analysis. For this purpose, we could automatically associate counters to program lines in our model of MIX programs, and make it possible for the user to refer to these counters in annotations.

Acknowledgements. I am sincerely grateful to L. Théry for his suggestion to consider the formal proof of MIX programs.

References

[1] Mike Barnett and K. Rustan M. Leino. Weakest-Precondition of Unstructured Programs. In *6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Lisbon Portugal, September 2005.

[2] Edsger W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

[3] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *19th International Conference on Computer Aided Verification*, 2007.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[5] D. E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison-Wesley, 1968.

[6] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1969.

[7] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[8] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[9] Laurent Théry. Proving Pearl: Knuth's algorithm for prime numbers. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, 2003.