Program Verification using Small Models

Nakshatra Gupta, Prajkta Kodavade, Shrawan Kumar, Abhisekh Sankaran, Akshatha Shenoy, R. Venkatesh

Tata Consultancy Services Research, Pune, India

Abstract

We explore the use of methods from mathematical logic, in particular model theory, in the verification of array programs. We consider the particular case of the min program P that computes the minimum of a given array, and the corresponding min assertion A that asserts that the value computed by the program is the minimum of the given array. We formulate the program P and assertion A in multi-sorted first order logic (FO), via sentences φ_P and φ_A respectively. The sentence φ_P is such that modulo a small set φ_{ax} of axioms, the models of φ_P are in 1-1 correspondence with (suitably abstract representations of) the executions of P on arbitrary input arrays. Likewise, the models of φ_A restricted to the class of models of $\varphi_P \wedge \varphi_{\text{ax}}$, correspond to (suitably abstracted) executions of P that satisfy the assertion A.

We now verify the min program P against the min assertion A by showing that the sentence $\alpha_{P,A} \coloneqq \varphi_{ax} \wedge \varphi_P \wedge \neg\varphi_A$ has the *small model property*. Specifically, we show that if $\alpha_{P,A}$ has a model of some finite size, then there is also such a model in which the domains interpreting the sorts have all sizes bounded by 7. The desired verification then reduces to checking whether there is a small model (with sort sizes at most 7) for $\alpha_{P,A}$; this check is performed using the SAT solver Z3. The small model property is established by showing that models of $\alpha_{P,A}$ that are large admit local reductions that are effected by "removal of iterations" in the executions of P that they correspond to.

Generalizing from our analysis above, we formulate a class of programs that we call *Monotone-loop programs without Lookback or Lookahead*, denoted MLL, and argue that for any program Q in MLL and any assertion B, if φ_Q and φ_B are resp. the logical formulations of Q and B, then $\varphi_{ax} \wedge \varphi_Q \wedge \neg\varphi_B$ has the small model property provided that certain combinatorial model-theoretic checks succeed, and that φ_B is a universal FO sentence. Consequently, the same translates to Monotone-loop programs with Bounded Lookback and Lookahead, denoted MLL+, since we argue that MLL is sufficiently expressive to subsume MLL+ upto equivalence.

1 Introduction

Proving properties of programs that process large arrays is hard. A variety of approaches have been explored including the development of a theory of arrays [1, 4, 5, 10, 13, 14, 15], static analysis techniques like smashing and partitioning [2, 3, 8, 9, 11, 12], and loop reduction techniques like pruning and shrinking [16, 17]. Techniques like pruning and shrinking show the equivalence of the original program with respect to the property to be verified, to a program with a small finite number of loop iterations, thus enabling the use

```
Min(n, a[n]):                        Assert-min(n, a[n]):
+++++++++++++                        +++++++++++++++++++
m = a[0];                            i = 0;
i = 0;                               while (i < n)
while (i < n)                        {
{                                        assert(min <= a[i]);
    if (m > a[i])                        i = i + 1;
        m = a[i];                    }
    i++;                             return 0;
}
assert(A);
```

$$\mathtt{A} := \forall j\ (j \geq 0 \wedge j < n) \rightarrow m \leq a[j]$$

Figure 1: The min program on the left and the min assertion as a program on the right. The min assertion can also be represented as an FO formula as above. (In our analysis though, we will consider a slightly different FO formulation.)

of bounded model checking to prove the property. This paper is inspired by this idea, and extends it using methods from mathematical logic. We specifically look at the concrete example of the min program and the min assertion as depicted in Figure 1. The program computes the minimum of a given array, and the assertion states that the value computed by the program is the minimum of the given array. We develop a model-theoretic approach to analyse the program against the assertion, whose overview we describe in this paper. We refer the reader to [22] for the technical details. The methods used in [22] admit extensions to various other programs that we describe subsequently in this introduction. Presently, we give a brief description of our approach below.

We verify the min program P for the min assertion A via translating P and A into sentences of multi-sorted first order logic (FO), that we denote φ_P and φ_A respectively. We use three specific sorts in our logical formulation: an "index sort", an "iteration sort" and a "value sort". The index sort is intended to model the set of indices of the input array a of P, the iteration sort to model the set of iteration numbers of the iterations of the loop as P executes on the input, and the value sort is intended to capture the values contained in the array a and the values the variable m takes during the execution of P. All of these sets have natural linear orders on them that would be needed in our formulation. We hence have binary predicates for these linear orders in our vocabulary, along with predicates for their corresponding successor relations. In addition, the vocabulary contains three binary predicates to encode the input array a, and the sequences of values taken by the variables m and i. Each of these predicates is intended to be interpreted as the graph of a function from a suitable domain sort to a suitable co-domain sort; for instance, the predicate for a would represent a function from the index sort to the value sort. These intended interpretations (or constraints on them) of the predicates in our vocabulary are captured using a finite set of axioms, the conjunction of which we denote φ_{ax}. We will be interested in only those models of the sentences φ_P and φ_A that are finite and that satisfy the axioms φ_{ax}; the other models of φ_{ax} have no correspondence with the structures that are used (or that are inherent) in the program P.

Our formulation of the sentence φ_P is such that modulo the axioms φ_{ax}, the models of φ_P are in 1-1 correspondence with suitably abstract representations of the executions of P on arbitrary input arrays (Theorem 2.2 and Remark 2.3). Here by an execution being suitably abstracted, we mean that the execution is seen as a tuple consisting of the input array a, and the sequences of values for the m and i variables at the end of the iterations of the loop as P runs on the array a (as opposed to also keeping track the values of the variables at all intermediate points in the loop body of P). Likewise, our formulation of φ_A is such that, restricted to the (finite) models of $\varphi_{ax} \wedge \varphi_P$, the models of φ_A correspond to the executions of P that satisfy the assertion A. The task of verifying P against A then translates to checking whether the implication "$\varphi_P \rightarrow \varphi_A$" holds modulo φ_{ax}; that is, whether the mentioned implication is a validity over the class of models of φ_{ax}. Checking the mentioned validity can equivalently be cast as checking the unsatisfiability of the negation of the implication, so the unsatisfiability of "$\varphi_P \wedge \neg\varphi_A$" modulo φ_{ax}, which in turn is equivalent to the (unrestricted) unsatisfiability of the sentence "$\varphi_{ax} \wedge \varphi_P \wedge \neg\varphi_A$".

It is however well-known that the satisfiability problem for FO is undecidable, over both arbitrary structures (shown by Turing [19]), and over finite structures (shown by Trakhtenbrot [20]). (Additionally, arbitrary FO-SAT is co-r.e. while finite FO-SAT is r.e.). These results hold even when the vocabulary contains just one binary predicate, further even when the binary predicate is required to satisfy the axioms of a function (further still for prefix FO formulae with only one quantifier alternation beginning with universal quantifiers) [18]. As a consequence, checking the satisfiability of "$\varphi_{ax} \wedge \varphi_Q \wedge \neg\varphi_B$" can easily become undecidable for a general program Q and assertion B. In view of the undecidability results, there has been an extensive study of decidable fragments of FO, beginning withe work of Löwenheim in 1915 [21] and culminating in the identification of all maximal prefix classes of FO that are decidable for the SAT problem [18, Chapter 1]. Examples of such classes apart from the Löwenheim class, include the Bernays-Schönfinkel-Ramsey class (also called Effectively Propositional Logic), the Ackermann class, and classes due to Gurevich, Shelah and Rabin (separately)[1].

The satisfiability of all the classes mentioned above, when they admit finite models for all of their satisfiable formulae, goes via showing the *small model property (SMP)* for the class [18]. This property states that for any sentence in the class, if it has a finite model, then there is also a model of size bounded by some computable function of the length of the sentence. The SMP ensures that the SAT problem for the class is decidable: one just needs to check all structures of sizes up to the "small model bound" for whether they are models of the given FO sentence. We adopt this approach for checking the satisfiability of our logical formulation as well; that is, we attempt to show the satisfiability of $\varphi_{ax} \wedge \varphi_P \wedge \neg\varphi_A$ via showing that it has the SMP. Unfortunately however, given the particular details of our logical formulation, none of the above mentioned classes shown to have the SMP serve to help since their syntax is quite restricted, and furthermore in general, the bounds on the small model sizes for formulae of the classes is very large: exponential or higher. Given

[1]Recently, a workshop titled "The Decision Problem in First Order Logic (DPFO 2023)" surveying the history of developments in the study of the FO-SAT was conducted at LICS '23.

our requirements of having small model bounds that would allow for a SAT solver like Z3 to run in competitive time with respect to existing tools for program verification, we develop our own method to show the SMP for the programs and assertions we consider.

The essential idea behind showing the SMP for $\alpha_{P,A} \coloneqq \varphi_{ax} \wedge \varphi_P \wedge \neg\varphi_A$ is as follows (see Figure 2 for a block diagram of the overall approach). We show that any model M of $\alpha_{P,A}$ whose iteration sort size is at least 8, admits a "local reduction" to yield another model M' of $\alpha_{P,A}$ whose iteration sort size is strictly smaller than that of M (Theorem 4.2). We describe the local reduction from the perspective of program execution. For any given execution E of the loop of P on an input array a, the reduction consists of identifying a small interval of iterations in the execution, and the corresponding subarray of the array a accessed in the interval's iterations, for which it is possible to "remove some iterations" and the corresponding array elements, and modify the resulting reduced subarray to preserve the I/O behaviour of the loop relative to the interval. That is, for instance, if $i, i+1, i+2$ is (constitute the set of numbers of the iterations in) the mentioned interval and $a[i], a[i+1], a[i+2]$ are the array elements accessed in the iterations of the interval, then we try to remove at least one iteration, say $i+1$, such that the following holds: there exists an array b_2 of size 2 obtained by modifying the subarray $(a[i], a[i+2])$ such that running the loop of P separately on b_2 *with the initial value of m as its value at the end of iteration $i-1$ of E, instead of $a[0]$*, and running the loop of P separately on the subarray $b_1 = (a[i], a[i+1], a[i+2])$ with the same initial value of m as mentioned, yield identical output values of m. If the executions of P's loop on arrays b_1 and b_2 with the mentioned initial value of m are respectively E_1 and E_2, then one can see that E_1 is a "sub-execution" of E, and that "replacing" E_1 with E_2 yields a new execution E' of P's loop on a strictly smaller (and potentially different) input array, that would have an identical eventual output value of the variable m as E has. This is because the value of m at the end of any iteration of P is completely determined by its value at the end of the previous iteration and the array element being accessed in the current iteration. Since E_2 gives the same output m value as E_1, the remainder of the iterations in E beyond those in E_1, would execute "unaware" of the local change made in substituting E_2 for E_1.

The check for whether such removal of iterations is always possible for any given execution of the loop of P, is performed by checking the executions of the loop for all arrays a and initializations of m that come from respective representative sets that are each small in number. These sets themselves depend on the length k of the interval, and would be representative in that, the mentioned check for removal of iterations would succeed for all integer arrays of length k and integer initializations of m if, and only if, they succeed for the particular arrays and initializations in the representative sets. This is given that P does *only comparisons* between elements, and the number of integer arrays of size k along with a given initialization of m, is "up to isomorphism" of linear orderings on the array values and the initialization, a fixed function of k. The function is an exponential, but then the values of k for which the removal check would be attempted would be small, and the whole check for all representative arrays and initializations, would be performed symbolically by feeding a suitable FO formula to Z3 to check for satisfiability. Once a k admitting the removal check is found, we know that any k length interval in any given execution E of P's loop admits an "I/O-preserving" strict reduction. Now given the 1-1 correspondence between the executions of the loop of P and the models of $\varphi_{ax} \wedge \varphi_P$ (Theorem 2.2), the said k would admit a local reduction in any model of $\alpha_{P,A}$ that is sufficiently large, in particular having iteration sort size > 7. Recursively performing the local reduction, we obtain a small model for $\alpha_{P,A}$ in which all sort sizes would be bounded by 7 (Theorem 4.1).

The above described method, that is worked out fully for the (min) program P and (min) assertion A (see [22]), admits an extension to various other programs, going by the fact that the technical results do not really need the specifics of the program P and assertion A, as they only utilize some structural features of P and A. We present a general class of programs and assertions that have these structures features. For the assertions, the structure consists of the logical formulation of the assertion being a universal FO sentence, that is an FO sentence in which all quantifiers that appear are universal. For programs, the structural features are captured via a syntactically defined class of programs that we call *Monotone-loop programs without Lookback or Lookahead*, or MLL in short. Intuitively, these are single loop array programs that can contain any fixed (so non-input dependent) number of read-only arrays (of length determined by the input) and individual variables, but only a single loop iterator variable that further only increases monotonically. The values of the individual variables are computed in any iteration by looking at their values in the previous iteration and the values stored in all the arrays at the indices whose values equal that of the iterator variable in the current iteration, and performing *only comparisons* between the said values. Examples of programs in MLL include variants of the min program, in particular, relativized versions of it that check for the minimum in subarrays satisfying given conditions (Figure 3), and the k-th min program that computes the k-th minimum of a given array for any given k (Figure 4 for $k = 2$ and more examples). We go further to show that MLL semantically is a much larger class, as it subsumes up to equivalence, among other programs, the class of Monotone-loop programs with *Bounded* Lookback and Lookahead, or MLL+ in short. These programs allow for comparisons involving not just the elements of the arrays at locations pointed to by the iterator variable, but more generally elements of the arrays at bounded distances from the mentioned locations. Since MLL+ programs can be compiled into equivalent MLL programs, the small model property for MLL programs (whenever it exists) admits a transfer to the corresponding MLL+ programs as well. (See Figure 5 for an example MLL+ program and its equivalent MLL program.)

Organization of the paper: In Section 2, we present the details of the logical formulation of P and A and also state precisely the correspondence result between models of $\varphi_{ax} \wedge \varphi_P$ and the executions of P. In Section 3, we present in more detail the overall approach depicted in Figure 2 that we have described above. Section 4 contains a selection of the main technical results from the full technical report for this paper in [22]. Section 5 presents MLL and MLL+, and the subsection 5.2 reports some preliminary experiments that we have performed using the implementations of our approach. We conclude in Section 6 presenting directions for future work.

2 Logical formulation of the min program and assertion

We assume the reader is familiar with standard notions and notation in the syntax and semantics of FO [23]. A vocabulary τ is a finite set of relation, function and constant symbols. In this paper, we are concerned only with *relational vocabularies* that contain only predicate and constant symbols. All predicate symbols are assumed to have positive arity. We denote by $\mathrm{FO}(\tau)$ the set of all FO formulae over τ. A sequence $x_1, \ldots, x_n$ of variables is written as $\bar{x}$. A formula φ whose free variables are among $\bar{x}$, is denoted $\varphi(\bar{x})$. A formula with no free variables is called a *sentence.* A τ-structure consists of a domain of elements, along with an interpretation for each predicate of τ as a relation of the same arity over the domain, and an interpretation for each constant of τ as an element of the domain. Given a τ-structure M and a sentence φ, we denote by $M \vDash \varphi$ that φ is true in M. All the notions mentioned have natural extension to multi-sorted FO.

We recall the min program and the min assertion as depicted in Figure 1, and present a logical formulation of it below. The vocabulary that we use is

$$\tau = \{\leq_d^{(2)}, \leq_r^{(2)}, \leq_v^{(2)}, \mathsf{s}_d^{(2)}, \mathsf{s}_r^{(2)}, \mathsf{a}^{(2)}, \mathsf{i}^{(2)}, \mathsf{m}^{(2)}, \underline{0_d}, \underline{0_r}, \underline{n_d}, \underline{n_r}\}$$

where the underlined symbols are constants and the rest are binary predicate symbols. There are 3 sorts associated with τ: the *array index sort* (D), the *iteration sort* (R) and the *value sort* (V). The *signature* of a predicate symbol of τ specifies the sorts of all arguments of the predicate; likewise the signature of a constant symbol specifies the sort of the symbol. The signatures of the symbols of τ are as below:

$$\begin{array}{lcl@{\qquad}lcl}
\mathrm{sig}(\leq_d) & = & D \times D & \mathrm{sig}(\mathsf{s}_d) & = & D \times D \\
\mathrm{sig}(\leq_r) & = & R \times R & \mathrm{sig}(\mathsf{s}_r) & = & R \times R \\
\mathrm{sig}(\leq_v) & = & V \times V & \mathrm{sig}(\mathsf{a}) & = & D \times V \\
\mathrm{sig}(\mathsf{i}) & = & R \times D & \mathrm{sig}(\mathsf{m}) & = & R \times V \\
\mathrm{sig}(\underline{0_d}) & = & D & \mathrm{sig}(\underline{0_r}) & = & R \\
\mathrm{sig}(\underline{n_d}) & = & D & \mathrm{sig}(\underline{n_r}) & = & R
\end{array}$$

Before presenting the logical formulation, let us see intuitively what the symbols are intended to be interpreted as. The $\leq_d, \leq_r$ and $\leq_v$ predicates are intended to represent linear orders on the sorts D, R and V. The linear order on sort D represents the ordering of the array indices, that on sort R represents the ordering of the iteration numbers of the loop, and that on sort V the usual ordering of numbers. The s_r and s_d predicates are intended to encode the successor relations of $\leq_r$ and $\leq_d$. The constants $\underline{0_d}$ and $\underline{n_d}$ encode the minimum and maximum elements of $\leq_d$ (observe that n denotes the length of the input array); likewise $\underline{0_r}$ and $\underline{n_r}$ encode the minimum and maximum elements of $\leq_r$. Finally, a, m and i are intended to encode functions with suitable domains and co-domains as follows: (i) a, that is used to model the array, is a function from sort D to sort V; (ii) m, that is used to model the sequence of intermediate min values computed across iterations, is a function from sort R to sort V; and (iii) i, that is used to model the location in the array that is looked at in any given iteration, is a function from sort D to sort V.

The above intended interpretations constitute the *axioms* for the mentioned predicates. All our analysis subsequently will be *modulo* these axioms, that is, the structures of interest for our formulae will be those that satisfy these axioms. We now give the details of the logical formulation of the axioms below.

Remark 2.1. We use a different notation below than in the introduction, but the correspondence is easy to see. So PAx below corresponds to φ_{ax}, and later PC corresponds to φ_P and PAs corresponds to φ_A.

We will use the following as convenient short-hands: $x <_d y, x >_d y, x \geq_d y, x <_r y, x >_r y, x \geq_r y, x <_v y, x >_v y, x \geq_v y,$. These are defined in the natural way. For e.g. $x <_d y := x \leq_d y \wedge x \neq y$ and $x \geq_d y := \neg(x \leq_d y) \vee x = y$.

Program axioms

1. Axioms for linear orders and successors: We give below the axioms $\mathsf{Ax:LO}$ and $\mathsf{Ax:succ}$ for the linear orders on all the sorts and and successors on all sorts except V. We provide these axioms $\mathsf{Ax:LO}_D$ and $\mathsf{Ax:succ}_D$ for the index sort, the axioms $\mathsf{Ax:LO}_R$ and $\mathsf{Ax:succ}_R$ for the iteration sort and the axioms $\mathsf{Ax:LO}_V$ for the value sort are similar.

$$\begin{aligned}
\mathsf{Ax:LO} &:= \mathsf{Ax:LO}_D \wedge \mathsf{Ax:LO}_V \wedge \mathsf{Ax:LO}_R \\
\mathsf{Ax:succ} &:= \mathsf{Ax:succ}_D \wedge \mathsf{Ax:succ}_R \\
\mathsf{Ax:LO}_D &:= \forall x \in D\,(x \leq_d x) \ \bigwedge \\
&\quad\ \forall x, y \in D\,(x \leq_d y \wedge y \leq_d x) \to x = y \ \bigwedge \\
&\quad\ \forall x, y, z \in D\,(x \leq_d y \wedge y \leq_d z) \to x \leq_d z \ \bigwedge \\
&\quad\ \forall x, y \in D\,(x \leq_d y \vee y \leq_d x) \\
\mathsf{Ax:succ}_D &:= \forall x, y \in D\,\big(\mathsf{s}_d(x, y) \leftrightarrow \forall z \in D\,(z \leq_d x \vee y \leq_d z)\big)
\end{aligned}$$

2. Axioms for constants: We give the axioms for the constants of signature D; those for constants of signature R are similar.

$$\begin{aligned}
\mathsf{Ax:const} &:= \mathsf{Ax:const}_D \wedge \mathsf{Ax:const}_R \\
\mathsf{Ax:const}_D &:= \forall x \in D\ (\underline{0_d} \leq_d x \wedge x \leq_d \underline{n_d})
\end{aligned}$$

3. Axioms for array predicates $\mathsf{a}, \mathsf{i}, \mathsf{m}$: These axioms $\mathsf{Ax:arr}_a, \mathsf{Ax:arr}_i, \mathsf{Ax:arr}_m$ for $\mathsf{a}, \mathsf{i}, \mathsf{m}$ resp. assert that each of these predicates is a function. We give below $\mathsf{Ax:arr}_a$; the others are similar.

$$\begin{aligned}
\mathsf{Ax:arr} &:= \mathsf{Ax:arr}_a \wedge \mathsf{Ax:arr}_m \wedge \mathsf{Ax:arr}_i \\
\mathsf{Ax:arr}_a &:= \forall x \in D\, \exists y \in V\, \mathsf{a}(x, y) \wedge \forall x \in D\, \forall y, z \in V\ \big((\mathsf{a}(x, y) \wedge \mathsf{a}(x, z)) \to y = z\big)
\end{aligned}$$

4. The axioms collectively are denoted as PAx.

$$\mathsf{PAx} := \mathsf{Ax:LO} \wedge \mathsf{Ax:succ} \wedge \mathsf{Ax:const} \wedge \mathsf{Ax:arr}$$

Program clauses

We now formulate the min program below. We observe that there are three parts to the program: the initialization, the loop body which is executed when the loop condition holds, and the termination of the program that happens when the loop condition does not hold. Each of these parts is formulated separately below.

Initialization: This sentence below corresponds to the initializations "`m = a[0];`" and "`i = 0;`" of the min program. (The $\leftrightarrow$ below can be replaced with $\leftarrow$ or $\rightarrow$ given the program axioms.)

$$\begin{aligned}
\mathsf{PI} &:= \mathsf{PI}_1 \wedge \mathsf{PI}_2\\
\mathsf{PI}_1 &:= \mathsf{i}(\underline{0_r}, \underline{0_d})\\
\mathsf{PI}_2 &:= \forall v_1 \in V\ \mathsf{a}(\underline{0_d}, v_1) \leftrightarrow \mathsf{m}(\underline{0_r}, v_1)
\end{aligned}$$

Body: The clauses C_1 and C_2 below resp. correspond to the "`if (m > a[i]) m = a[i];`" statement and the implicit "else" condition of the statement. The clause C_3 corresponds to "`i++;`". The clause C_0 expresses that if the loop condition is satisfied in an iteration, then that iteration is not the last iteration of the loop (and hence whose position on the iteration linear order $\leq_r$ is less than the maximum $\underline{n_r}$ of the linear order), where by last iteration we mean the (vacuous) iteration in which the loop condition gets violated.

$$\begin{aligned}
\mathsf{PB} &:= \forall l_1, l_2 \in R\ \forall k_1, k_2 \in D\ \forall v_1, v_2 \in V\ (C_0 \wedge C_1 \wedge C_2 \wedge C_3)\\
C_0 &:= (\mathsf{i}(l_1, k_1) \wedge k_1 <_d \underline{n_d}) \rightarrow l_1 <_r \underline{n_r}\\
C_1 &:= (\mathsf{i}(l_1, k_1) \wedge k_1 <_d \underline{n_d} \wedge \mathsf{m}(l_1, v_1) \wedge \mathsf{a}(k_1, v_2) \wedge \mathsf{s}_r(l_1, l_2) \wedge v_1 >_v v_2) \rightarrow \mathsf{m}(l_2, v_2)\\
C_2 &:= (\mathsf{i}(l_1, k_1) \wedge k_1 <_d \underline{n_d} \wedge \mathsf{m}(l_1, v_1) \wedge \mathsf{a}(k_1, v_2) \wedge \mathsf{s}_r(l_1, l_2) \wedge v_1 \leq_v v_2) \rightarrow \mathsf{m}(l_2, v_1)\\
C_3 &:= (\mathsf{i}(l_1, k_1) \wedge k_1 <_d \underline{n_d} \wedge \mathsf{s}_r(l_1, l_2) \wedge \mathsf{s}_d(k_1, k_2)) \rightarrow \mathsf{i}(l_2, k_2)
\end{aligned}$$

Termination: The formula below expresses that if in an iteration the loop condition gets violated, then that iteration is the last iteration of the loop.

$$\mathsf{PT} := \forall l \in R\ \forall k \in D\ (\mathsf{i}(l, k)\ \wedge\ \neg(k <_d \underline{n_d})) \rightarrow l = \underline{n_r}$$

Overall program clauses: Overall then, the FO sentence for the program is just the conjunction of the sentences for the initialization, the loop body and the program termination as seen above.

$$\mathsf{PC} := \mathsf{PI}\ \wedge\ \mathsf{PB}\ \wedge\ \mathsf{PT}$$

Assertion:

We now formulate the min assertion as shown in Figure 1. The assertion states that the final computed value of the variable `m` is no more than the value at any array index. This can be formalized as PAs below. The negation of PAs, which will be involved in our analysis in the subsequent sections, is given after. (The ":=" in the formula for $\neg\mathsf{PAs}$ is a bit of an abuse of notation; it should be "$\leftrightarrow$" strictly speaking, but we treat the equivalence as equality for later convenience.)

$$\begin{aligned}
\mathsf{PAs} &:= \forall k_s \in D\ \forall v_t, v_s \in V\\
&\qquad \big(\ \underline{0_d} \leq_d k_s\ \wedge\ k_s <_d \underline{n_d}\ \wedge\ \mathsf{m}(\underline{n_r}, v_t)\ \wedge\ \mathsf{a}(k_s, v_s)\ \big) \rightarrow v_t \leq_v v_s\\
\neg\mathsf{PAs} &:= \exists k_s \in D\ \exists v_t, v_s \in V\\
&\qquad \big(\ \underline{0_d} \leq_d k_s\ \wedge\ k_s <_d \underline{n_d}\ \wedge\ \mathsf{m}(\underline{n_r}, v_t)\ \wedge\ \mathsf{a}(k_s, v_s)\ \wedge\ v_s <_v v_t\ \big)
\end{aligned}$$

The following theorem now shows that the logical formulation above "captures" the min program and assertion in a precise sense. The proof is not difficult to see, hence we skip it.

Theorem 2.2. *The following are true:*

1. *Let $a[n]$ be a given array of length n. Let $m[n]$, resp. $i[n]$, be the array of intermediate values of the variable m, resp. i, computed across the iterations of the loop in the* min *program, when the loop is executed on $a[n]$ and a given initialization $m[0]$ of m. Then there exists, up to isomorphism, a unique τ-structure M^+ such that: (i) the interpretations of the predicates* a, m *and* i *contained in M^+ resp. represent the arrays $a[n], m[n]$ and $i[n]$; and (ii) $M^+ \models \mathsf{PAx} \wedge \mathsf{PI}_1 \wedge \mathsf{PB} \wedge \mathsf{PT}$. In particular, when the initialization $m[0]$ of m is $a[0]$, then $M^+ \models \mathsf{PAx} \wedge \mathsf{PC}$.*

2. *Let M be a τ-structure such that $M \models \mathsf{PAx} \wedge \mathsf{PI}_1 \wedge \mathsf{PB} \wedge \mathsf{PT}$. Let the interpretations of the predicates* a, m *and* i *contained in M resp. represent the arrays $a^+[n], m^+[n]$ and $i^+[n]$. Then $m^+[n]$, resp. $i^+[n]$, is the array of intermediate values of the variable m, resp. i, computed across the iterations of the loop in the* min *program, when the loop is executed on $a^+[n]$ and the initialization $m^+[0]$ of m. In particular, if $M \models \mathsf{PAx} \wedge \mathsf{PC}$, then it additionally holds that the initialization $m^+[0]$ of m is equal to $a^+[0]$.*

Remark 2.3. The sequence $(a[l], m[l], i[l])_{0 \leq l < n}$ can be seen as an abstract representation of the (unique) execution of the min program on the array $a[n]$. Clearly the sequence determines the said execution and vice-versa. Theorem 2.2 can then be seen to state that the execution of the min program on any given input array, determines up to isomorphism, a unique τ-structure that models $\mathsf{PAx} \wedge \mathsf{PC}$, and that encodes the mentioned abstract representation of the execution, in the interpretations of its predicates corresponding to the min program variables. And conversely, any τ-structure that models $\mathsf{PAx} \wedge \mathsf{PC}$, determines (uniquely) the abstract representation of the execution of the min program, on the particular array encoded in the structure, that is represented by the interpretation of the predicate of τ corresponding to the array variable. In short, the models of $\mathsf{PAx} \wedge \mathsf{PC}$ are (up to isomorphism) in 1-1 correspondence with (the abstract representations of) the executions of the min program for arbitrary inputs.

3 The Overall Approach

Our overall approach to showing the small model property for programs and assertions is as shown in Figure 2. We describe this briefly here in the specific case of the min program and assertion. The overall approach instantiates similarly for the more general setting we consider in Section 5. There are 4 stages to the entire analysis as explained below. (We again switch back to our notation used in the introduction, namely φ_{ax}, φ_P, and φ_A for the axioms and the logical formulations of P and A. For the min program, these correspond respectively to $\mathsf{PAx}, \mathsf{PC}$ and PAs as seen in Section 2.)

1. In Stage I, the input program P and assertion A are taken as input and converted to their logical formulations φ_P and φ_A. Along with these, the set φ_{ax} of axioms is also generated. These three formulae are passed to the next stage.

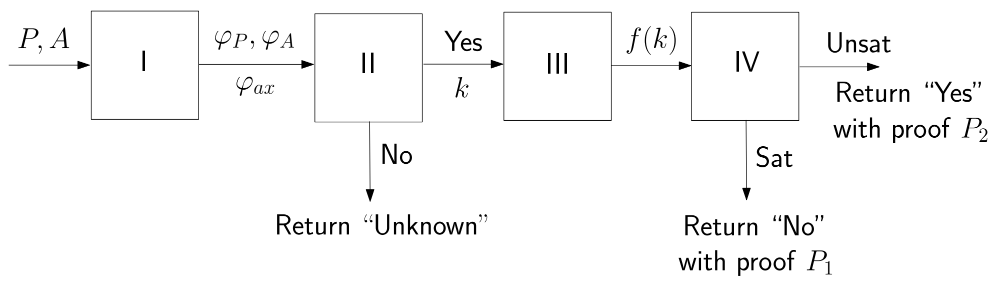

Figure 2: A block diagram of the overall approach. In Stage I, the input program P and assertion A are converted resp. to logical formulae φ_P and φ_A; also a small set φ_{ax} of axioms is generated. In Stage II, a procedure is executed that attempts to find a k that allows for an output-preserving "removal of iterations" in executions of P on arrays of size k and initializations that come from respective representative sets. If such a k isn't found within threshold time, Stage II returns "Unknown", else the found k is passed on to Stage III. In Stage III, a number $f(k)$ is calculated that serves as a bound on the sizes of small models of $\varphi_{ax} \wedge \varphi_P \wedge \neg\varphi_A$. Stage IV consists of finding a small model for the mentioned formula using Z3. Such a model if found serves to witness the violation of A by P, otherwise A provably holds under P.

2. In Stage II, for a fixed number k that is incremented starting with the value 2, we symbolically execute P on all input arrays a of length at most k and all initializations m_{in} of the variable m. For each of these executions E, we try to "remove some iterations" along with the array elements accessed in these iterations, modifying the resulting array suitably to obtain an array a' such that the execution E' of P on a' and the same initialization m_{in} of m, yields the same output m_{out} as produced earlier by E. Note that this is always possible to do for any given execution via some suitable combinatorial reasoning.

   The only issue the reader might see here is in dealing with the "for each of these executions" clause that would require considering *all possible* input arrays a and initializations m_{in} of m, which are infinite in number. It is here that we utilize the fact that the min program (and more generally any program of MLL described in Section 5) involves only *comparisons* between values stored in the variables of the program. Since the number of values that are compared in an execution E of P on a given array a of length k and a given initialization m_{in} of m, is at most $k+1$, the comparisons can be simulated on simply the set $\{0, 1, \ldots, k\}$. This is given that up to isomorphism, there is just one linear order of length $k+1$, and that the number of linear orders on $k+1$ elements that are themselves labeled using the elements of another linearly ordered set of size $k+1$, is at most $(k+1)^{(k+1)}$.[2].

   The above mentioned (output-preserving) removal of iterations from executions E can be then checked on only representative sets of arrays of size k and initializations of m, those in which the arrays and initializations take on values from the set

[2]Another way of stating this is that the number of structures that consist of a set of size $k+1$ equipped with two linear orders on the set, is up to isomorphism, at most $(k+1)^{(k+1)}$.

$\{0, \ldots, k\}$. This check in turn now can be formulated as the SAT problem for a suitably constructed FO formula θ_k. If the SAT solver (Z3) returns SAT on θ_k, then it corresponds to the iteration removal check failing for k, in which case we increment the value of k to $k+1$, and ask for the satisfiability of θ_{k+1}. This process is continued until either Z3 returns UNSAT for some value k_0 of k, or a timeout is reached. In the latter case, we return "Unknown" as the output of our overall analysis, and in the former case, the value k_0 is passed on to Stage III of our analysis. We note that k_0 admits removal of iterations for (the executions of the min program's loop on) all (of the infinitely many) arrays of size k and initializations of m.

3. Stage III of the overall approach involves a theoretical analysis that takes in the value of k output from Stage II (if there is such a value), and outputs a suitable value $f(k)$ which represents a bound on the small model sizes for the formula $\alpha_{P,A} \coloneqq \varphi_{ax} \wedge \varphi_P \wedge \neg\varphi_A$. More precisely, the analysis formulates a *reduction theorem* (Theorem 4.2) that states that if there is a model M of $\alpha_{P,A}$ whose iteration sort domain is of size greater than $f(k)$, then there is another model M' of $\alpha_{P,A}$ whose iteration sort domain is strictly smaller than that of M. Briefly the model M' is produced from M via a local reduction as explained below.

   A suitable substructure M_c of M is first chosen that is a model of $\varphi_{ax} \wedge \varphi_P$, and whose iteration sort size is k. The particular features of our logical formulation ensure that we will always be able to obtain such a substructure. Using Theorem 2.2, this substructure corresponds to the execution of P on some array b of size k and some initialization m_{in} of m. Since k admits a removal of iterations, there is another array b' of size $< k$ such that with m_{in} as the initialization of m, the program P outputs the same m value as it did on b. We re-translate b' to a structure M_c' using Theorem 2.2, and substitute it for M_c in M to obtain the structure M'. We now use the properties of our logical formulation in conjunction with classical preservation theorems from model theory to show that M' is a model of $\alpha_{P,A}$.

   As a consequence, recursively applying the above reduction theorem to M', we get a model M^* of $\alpha_{P,A}$ whose iteration sort domain is of size at most $f(k)$. It follows then that for the min program (and more generally for the programs of MLL), the index sort domain also has size $\leq f(k)$, and (consequently) the value sort domain is of size $\leq c \cdot f(k)$ for a constant c that is determined by the program (it is in particular the total number of array and individual variables of the program). Since all domains of M^* are bounded, M^* is a small model of $\alpha_{P,A}$.

4. Finally, we have Stage IV, that takes as input the value $f(k)$ output by Stage III, and checks for the satisfiability of $\alpha_{P,A}$ in small structures, that is structures whose iteration and index sort domains have sizes $\leq f(k)$ and value sort domain has size $\leq c \cdot f(k)$ for c as aforementioned. The check is again done using Z3. In the event that Z3 returns "SAT", and hence also a satisfying model M^*, we utilize Theorem 2.2 to translate M^* to an array a^* and (a suitably abstract representation of) the execution E^* of P on a^* that witnesses the violation of the assertion A. The pair (a^*, E^*) can be taken as the proof P_1 in Figure 2.

   In the event that Z3 returns "UNSAT", it is immediate that there is no model of $\alpha_{P,A}$ whose iteration sort domain has size at most $f(k)$. However, from the reduction theorem above, it follows that there is no (finite) model for $\alpha_{P,A}$. This is because, taken in contrapositive form, the reduction theorem says that for all $l \geq f(k)$, if there is no model of $\alpha_{P,A}$ whose iteration sort domain has size at most l, then there is also no model of $\alpha_{P,A}$ whose iteration sort domain has size at most $l+1$. Thus the reduction theorem provides an induction on l where the base case of the induction is the case when $l = f(k)$. We see that Z3 returning 'UNSAT" on $\alpha_{P,A}$ can then be seen as Z3 proving the base case of the mentioned induction. It follows then that for all l, there is no model of $\alpha_{P,A}$ whose iteration sort domain has size at most l; in other words, $\alpha_{P,A}$ has no finite model. As a result, in conjunction with Theorem 2.2, it holds that there is no input array a for which the finite execution of P on a leads to a violation of assertion A, establishing that A always holds under P.

The proof P_2 referred to in Figure 2 can then be returned as a pair consisting of the unsat core of $\alpha_{P,A}$ (over small models) along with the proof of the reduction theorem in contrapositive form, written as say a program in Coq. (This is of course just a theoretical proposal at this stage.)

4 Technical results

We recall the min logical formulation from Section 2. For a τ-structure M, let D^M, R^M and V^M be resp. the domains of M for sorts D, R and V. Define $|M|_D, |M|_R, |M|_V$ to be the sizes of the sets D^M, R^M and V^M. The size of M, denoted $|M|$, is simply $|M|_R + |M|_D + |M|_V$. Our main result stated below shows the small model property for the logical formulation of the min program and assertion.

Theorem 4.1 (Small models for $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$[3]). *If there is a model for* $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$*, then there is a model* M^* *of* $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$ *such that* $|M^*|_R, |M^*|_D, |M^*|_V \leq 7$.

Thus to check if P satisfies assertion A, we just need to check for the satisfiability of $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$ in structures whose domains sizes are bounded by 7 as discussed in the previous section. The proof of the small model theorem above goes via showing the following two results, the first a *reduction theorem* for $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$, and the second an *equal-sorts lemma* for $\mathsf{PAx} \wedge \mathsf{PC}$. The reduction theorem is the technical core of Theorem 4.1.

Theorem 4.2 (Reduction of large models of $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$). *Let* M *be a model of* $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$. *If* $|M|_R > 7$*, then there exists a model* M' *of* $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$ *such that* $|M'|_R < |M|_R$.

Lemma 4.3 (Equality of sizes of the iteration and index sorts in models of $\mathsf{PAx} \wedge \mathsf{PC}$). *In any model* M *of* $\mathsf{PAx} \wedge \mathsf{PC}$*, it holds that* $|M|_D = |M|_R$.

Using the reduction theorem and equal-sorts lemma, we can prove Theorem 4.1 as below.

[3]The upper bound of 7 can be lowered further, but that would need a custom handling of some corner cases.

Proof of Theorem 4.1. Let M be a model of $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$. If $|M|_R \leq 7$, then we take $M^* = M$; else by recursively applying Theorem 4.2 to M, we eventually obtain a model M^* of $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$ such that $|M^*|_R \leq 7$. Since in either case M^* models $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$, we get by Lemma 4.3 that $|M^*|_D = |M^*|_R \leq 7$. Since:

- a and m are interpreted as (the graphs of) functions in M^* (as $\mathsf{Ax{:}arr}$ is a conjunct of PAx), and
- range(m^{M^+}) $\subseteq$ range(a^{M^+}) (as PB is a conjunct of PC), where range(m^{M^+}) and range(a^{M^+}) resp. denote the ranges of the functions interpreting m and a,

it follows that there exists $V_1 \subseteq V^{M^*}$ s.t.:

$$|V_1| \leq 7; \quad \text{and} \quad \text{range}(\mathsf{m}^{M^+}), \text{range}(\mathsf{a}^{M^+}) \subseteq V_1$$

. Then w.l.o.g. V^{M^*} can be taken to be V_1 itself, whereby M^* is indeed as desired. □

4.1 Regrouping the program clauses

To simplify our analysis to be able to prove Theorem 4.2 and Lemma 4.3, and simultaneously to cast our proof in a form that can generalize to other programs, we regroup the individual components of the program clauses. Towards this, we first observe that there are two kinds of variables in the input program – those that take on as values the indices of arrays, and those that constitute the rest. The variables of the first kind are i, n, and those of the second are a, m. Correspondingly, there are two kinds of elements in τ – those involved in the "loop infrastructure", and those involved in the "actual computations" in the loop. The elements of the former kind constitute the set $\tau_{\mathsf{LI}} = \{\leq_d, \leq_r, \mathsf{s}_r, \mathsf{s}_d, \underline{0_d}, \underline{0_r}, \underline{n_d}, \underline{n_r}, \mathsf{i}\}$, and those of the latter constitute $\tau_{\mathsf{AC}} = \{\leq_d, \leq_v, \mathsf{s}_r, \mathsf{i}, \mathsf{a}, \mathsf{m}, \underline{0_d}, \underline{0_r}, \underline{n_d}\}$. So $\tau = \tau_{\mathsf{LI}} \cup \tau_{\mathsf{AC}}$. Our regrouping of the program clauses is based on gathering together all components that exclusively deal with the loop infrastructure in one group that we denote LI, and gathering together the components that deal with the actual computations in the loop in another group that we denote AC. The program axioms and the assert clause are in groups of their own resp. again called PAx and $\neg\mathsf{PAs}$.

$$\begin{aligned}
\mathsf{PAx} &:= \mathsf{Ax{:}LO} \wedge \mathsf{Ax{:}succ} \wedge \mathsf{Ax{:}const} \wedge \mathsf{Ax{:}arr}\\[1em]
\mathsf{LI} &:= \mathsf{LI{:}Init} \wedge \mathsf{LI{:}Body} \wedge \mathsf{LI{:}Term}\\
\mathsf{LI{:}Init} &:= \mathsf{i}(\underline{0_r}, \underline{0_d})\\
\mathsf{LI{:}Body} &:= \forall l_1, l_2 \in R\ \forall k_1, k_2 \in D\ (\mathsf{LIB{:}1} \wedge \mathsf{LIB{:}2})\ \ \text{where}\\
\mathsf{LIB{:}1} &:= (\mathsf{i}(l_1, k_1) \wedge k_1 <_d \underline{n_d}) \to l_1 <_r \underline{n_r}\\
\mathsf{LIB{:}2} &:= (\mathsf{i}(l_1, k_1) \wedge k_1 <_d \underline{n_d} \wedge \mathsf{s}_r(l_1, l_2) \wedge \mathsf{s}_d(k_1, k_2)) \to \mathsf{i}(l_2, k_2)\\
\mathsf{LI{:}Term} &:= \forall l \in R\ \forall k \in D\ (\mathsf{i}(l, k)\ \wedge\ \neg(k <_d \underline{n_d})) \to l = \underline{n_r}\\[1em]
\mathsf{AC} &:= \mathsf{AC{:}Init} \wedge \mathsf{AC{:}Body}\\
\mathsf{AC{:}Init} &:= \forall v_1 \in V\ \mathsf{a}(\underline{0_d}, v_1) \to \mathsf{m}(\underline{0_r}, v_1)\\
\mathsf{AC{:}Body} &:= \forall l_1, l_2 \in R\ \forall k_1, k_2 \in D\ \forall v_1, v_2 \in V\ (\mathsf{ACB{:}1} \wedge \mathsf{ACB{:}2})\\
\mathsf{ACB{:}1} &:= (\mathsf{i}(l_1, k_1) \wedge k_1 <_d \underline{n_d} \wedge \mathsf{m}(l_1, v_1) \wedge \mathsf{a}(k_1, v_2) \wedge \mathsf{s}_r(l_1, l_2) \wedge v_1 >_v v_2)\\
&\quad\ \to \mathsf{m}(l_2, v_2)\\
\mathsf{ACB{:}2} &:= (\mathsf{i}(l_1, k_1) \wedge k_1 <_d \underline{n_d} \wedge \mathsf{m}(l_1, v_1) \wedge \mathsf{a}(k_1, v_2) \wedge \mathsf{s}_r(l_1, l_2) \wedge v_1 \le_v v_2)\\
&\quad\ \to \mathsf{m}(l_2, v_1)\\[1em]
\neg\mathsf{PAs} &:= \exists k_s \in D\ \exists v_t, v_s \in V\\
&\quad \big(\ \underline{0_d} \le_d k_s\ \wedge\ k_s <_d \underline{n_d} \wedge\\
&\quad\ \ \mathsf{m}(\underline{n_r}, v_t)\ \wedge\ \mathsf{a}(k_s, v_s)\ \wedge\ v_s <_v v_t\big)
\end{aligned}$$

Observe that LI is an FO$[\tau_{\mathsf{LI}}]$ sentence and AC is an FO$[\tau_{\mathsf{AC}}]$ sentence; PAx and $\neg\mathsf{PAs}$ are FO$[\tau]$ sentences.

The following lemma is easy to see.

Lemma 4.4. *The sentences* PC *and* $\mathsf{LI} \wedge \mathsf{AC}$ *are equivalent over all* τ*-structures. Hence, so are* $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$ *and* $\mathsf{PAx} \wedge \mathsf{LI} \wedge \mathsf{AC} \wedge \neg\mathsf{PAs}$.

Given Lemma 4.4, the proof of Lemma 4.3 is obtained by showing a structural characterization of the models of $\mathsf{PAx} \wedge \mathsf{LI}$. Again Lemma 4.4 is used in proving Theorem 4.2 by proving the same statement as in Theorem 4.2, with $\mathsf{PAx} \wedge \mathsf{PC} \wedge \neg\mathsf{PAs}$ replaced with $\mathsf{PAx} \wedge \mathsf{LI} \wedge \mathsf{AC} \wedge \neg\mathsf{PAs}$. The details can be found in [22].

5 The class MLL

We define a class of programs that generalizes the min program and to which we believe (cf. Remark 5.1) our analyses for the min program generalizes, when the logical formulations of the assertions are in *universal FO*. We call this class *Monotone-loop programs without Lookback or Lookahead*, or MLL in short. The intuitive description of any program in this class is as below. Below SSA denotes “Single Static Assignment”. It is known that SSA is a normal form (semantics preserving) for any program.

```
// Individual variable and array declarations.
// Initializations that are either of the form x = const or x = y
// where x and y are either individual variables or array elements.

// A loop-free set of statements that could involve conditions.

i := 0;
while (i < n)
{
    // The main loop body which is a sequence
    // of statements in SSA form that could
    // involve conditions and that satisfy the
    // following constraints:
    // a. feature only i as index variable
    // b. do not modify i
    // c. use only comparison as an operator
    // d. refer only to a[i] for an array a
    //    if they at all refer to any element
    //    of a
    // e. Assignment statements appear only
    //    at the ends of branches of the
    //    control flow graph of the main
    //    loop body.

    i++;
}
```

We now formalize the above intuitive description. We give below the syntax of the declarations, the initializations, the loop-free body, and the loop body of MLL programs. Overall an MLL program has the following syntax.

$$\mathsf{mllprog} \rightarrow \mathsf{decl\ allinits\ loopfreebody\ loopsyntax}$$

MLL declarations syntax: The syntax of the declarations is as below. Below n is an input variable of type int.

$$\begin{array}{lcl} \mathsf{decl} & \rightarrow & \mathsf{int}\ \ i, \mathsf{varlist} \\ \mathsf{varlist} & \rightarrow & \mathsf{indvar} \mid \mathsf{arrvar} \\ \mathsf{indvar} & \rightarrow & m_1 \mid m_2 \mid \ldots \\ \mathsf{arrvar} & \rightarrow & \mathsf{arrvarname}[n] \\ \mathsf{arrvarname} & \rightarrow & a_1 \mid a_2 \mid \ldots \end{array}$$

MLL initialization syntax: Defined below is the syntax for initializations in MLL. Let $\mathsf{Constants}$ be a finite set of integers.

$$
\begin{array}{lcl}
\mathsf{allinits} & \rightarrow & i = 0; \mid \mathsf{indivinit}; \mathsf{allinits} \\
\mathsf{indivinit} & \rightarrow & \mathsf{indvarinit} \mid \mathsf{arrvarinit} \\
\mathsf{indvarinit} & \rightarrow & \mathsf{indvar} = \mathsf{const} \mid \mathsf{indvar} = \mathsf{indvar} \mid \\
 & & \mathsf{indvar} = \mathsf{arrvarname}[\mathsf{const}] \\
\mathsf{arrvarinit} & \rightarrow & \mathsf{arrvarname}[\mathsf{const}] = \mathsf{const} \mid \\
 & & \mathsf{arrvarname}[\mathsf{const}] = \mathsf{indvar} \mid \\
 & & \mathsf{arrvarname}[\mathsf{const}] = \mathsf{arrvarname}[\mathsf{const}] \\
\mathsf{const} & \rightarrow & c \quad (\text{for } c \in \mathsf{Constants})
\end{array}
$$

MLL loop body syntax: The grammar below defines the syntax of the loop body. Below "ϵ" denotes the empty string.

$$
\begin{array}{lcl}
\mathsf{loopsyntax} & \rightarrow & \mathsf{whilecond}\ \{\ \mathsf{mainloopbody}\ \mathsf{iteratorinc}\ \} \\
\mathsf{whilecond} & \rightarrow & \mathtt{while}\ (i < n) \\
\mathsf{iteratorinc} & \rightarrow & \mathtt{i++;} \\
\mathsf{mainloopbody} & \rightarrow & \mathsf{condstmts}\ \mathsf{assignstmts} \\
\mathsf{condstmts} & \rightarrow & \epsilon \mid \mathsf{singlecondstmt}\ \mathsf{condstmts} \\
\mathsf{singlecondstmt} & \rightarrow & \mathsf{ifthenstmt} \mid \mathsf{ifthenelsestmt} \\
\mathsf{ifthenstmt} & \rightarrow & \mathtt{if}\ (\mathsf{cond})\ \{\ \mathsf{mainloopbody}\ \} \mid \\
 & & \mathtt{if}\ (\mathsf{cond})\ \mathsf{singleassignstmt} \\
\mathsf{ifthenelsestmt} & \rightarrow & \mathsf{ifthenstmt}\ \mathtt{else}\ \{\ \mathsf{mainloopbody}\ \} \\
 & & \mathsf{ifthenstmt}\ \mathtt{else}\ \mathsf{singleassignstmt} \\
\mathsf{cond} & \rightarrow & \mathsf{atomcond} \mid \\
 & & \mathsf{cond} \wedge \mathsf{cond} \mid \mathsf{cond} \vee \mathsf{cond} \mid !\mathsf{cond} \\
\mathsf{atomcond} & \rightarrow & \mathsf{loopvar} \circledast \mathsf{loopvar}\ (\text{for } \circledast \in \{==, <, \leq, >, \geq\}) \\
\mathsf{loopvar} & \rightarrow & \mathsf{indvar} \mid \mathsf{arrvarname}[i] \\
\mathsf{assignstmts} & \rightarrow & \epsilon \mid \mathsf{singleassignstmt}\ \mathsf{assignstmts} \\
\mathsf{singleassignstmt} & \rightarrow & \mathsf{loopvar} = \mathsf{loopvar};
\end{array}
$$

MLL loop-free body syntax: The grammar below defines the syntax of the loop-free body. Below $\mathsf{condstmts}$ and $\mathsf{assignstmts}$ are given (inductively) by the same set of production rules as in the loop body syntax above. The only change is in the production rule for $\mathsf{loopvar}$ which is as below. The production rule for const is as given in the grammar for initializations above.

$$
\begin{array}{lcl}
\mathsf{loopfreebody} & \rightarrow & \mathsf{condstmts}\ \mathsf{assignstmts} \\
\mathsf{loopvar} & \rightarrow & \mathsf{indvar} \mid \mathsf{arrvarname}[\mathsf{const}]
\end{array}
$$

We present below examples of programs in MLL. Figure 3 shows two "relativized" min programs that compute the minimum of subarrays of the given array satisfying certain "quantifier-free" properties. Figure 4 shows the "second-min" program that computes the second minimum in a given array, if it exists, a search program that searches for a given element in a given array, and the copy program that copies a given array into another.

One can use the mentioned programs in conjunction to reason about more complex programs. For example, another implementation of second-min can be obtained as follows. We first run the min program on a given array a; let m be the computed minimum. We then pass m as the value of the parameter p in the Rel-min-I program and run this program on array a. The value computed by the Rel-min-I program would then be the second minimum of a (if a contains at least two distinct values). Since we have verified the min program for its intended behaviour, and also claim that (we explain this in Section 5.2) Rel-min-I program is also correct for its intended behaviour, the composite program that we have just described for second-min would also be correct for its intended behaviour.

```
Relmin-I(n, a[n], p):
++++++++++++++++++++
m = a[0];
i = 0;
while (i < n)
{
    if (m > a[i] && a[i] != p)
        m = a[i];
    i++;
}
assert(A1);
```

```
Relmin-II(n, a[n], p[n]):
+++++++++++++++++++++++++
m = a[0];
i = 0;
while (i < n)
{
    if (p[i] == 1) {
        if (m > a[i])
            m = a[i];
    }
    i++;
}
assert(A2);
```

$$\texttt{A1} = \forall j (j \geq 0 \wedge j < n \wedge a[j] \neq p) \rightarrow m \leq a[j])$$
$$\texttt{A2} = \forall j (j \geq 0 \wedge j < n \wedge p[j] = 1) \rightarrow m \leq a[j])$$

Figure 3: Two "relativized" min programs. The program on the left computes the minimum of the subarray of a consisting of elements that are not equal to input value p. The program on the right computes the minimum of the subarray of a that is not "masked by" by the input array p of equal length. Both programs can be seen to be in MLL.

We now describe why our analysis for the min program and assertion might generalize to any program of MLL, in the remark below.

Remark 5.1. The reason why we believe our analysis for the min program and assertion generalizes to all MLL programs is given the observations about the proof for min as given in [22]. (These are marked in blue in the report.) All the programs of MLL can be seen to satisfy the conditions identified in these observations. The conditions either are directly on the structure of the statements in the program, or are on the structure of the logical formulations of the statements. Both of these are very easy to check for any MLL program.

Since its is only these structural conditions that make the proof for the min program go through, they should allow for the proof to lift to any program satisfying the structural conditions, in particular programs of MLL.

5.1 Allowing for bounded lookback and lookahead

The syntax of MLL while it allows for expressing interesting programs, might seem restrictive given the constraint of not having any lookback or lookahead. Many more interesting programs come into view when some lookback or lookahead is allowed. An example is the sortedness program depicted in Figure 5 that checks if a given array is sorted; the natural formulation of this program uses a lookback of 1. We therefore define the extension of MLL that we call *Monotone-loop programs with Bounded Lookback and Lookahead*, that we denote simply as MLL+. The syntax of MLL+ is identical to MLL except for the condition that in any iteration i, for any array a, not just can $a[i]$ be accessed, but also $a[i-r]$ and $a[i+r]$ for a number r is that upper bounded by a given constant. Therefore the syntax of the loop body of MLL+ programs is identical to that above for MLL, except for the production rule for loopvar which is modified as

$$\mathsf{loopvar} \quad\rightarrow\quad \mathsf{indvar} \mid \mathsf{arrvarname}[i] \mid \mathsf{arrvarname}[i-r] \mid \mathsf{arrvarname}[i+r]$$

where r is a number that is at most the given constant. The syntax for declarations and initializations are identical to those for MLL.

We now reason how MLL+ programs can be compiled into MLL programs. The bounded lookback in any MLL+ program can be implemented using *additional variables* that store the array values at the indices up to the lookback. For instance, for a lookback $a[i-r]$, we introduce variables $m_1, \ldots, m_r$ such that m_j would store the value of $a[i-j]$ for $j \in [1, r]$. These variables can be updated at the end of each loop by "shifting their values" suitably. Then $a[i-r]$ can be referred to without any lookback by using m_r in its place. On the other hand, bounded lookahead can be implemented by *unrolling the loop till the maximum lookahead* to a loop-free program which would only refer to fixed (constant) indices in the arrays. Then an affine transformation of the form $i \rightarrow i-s$ where s is the maximum lookahead would bring the "remainder" of the loop into the class of bounded lookback loops which then can be transformed into loops without lookback as explained above. Figure 5 presents an example.

5.2 Experiments

We ran some preliminary experiments using our (currently limited) implementation of our methods, on 4 programs: min as in Figure 1, its two relativized versions as in Figure 3 and the search program of Figure 4. We were able to experimentally obtain the values of k for which the "removal of iterations" that takes place in Stage II of our overall approach goes through (cf. Section 3). Using this, we computed the small model bounds $f(k)$ for these programs for the assertions that capture their intended behaviour (the assertions are defined in the figures mentioned). The table below presents all these values along with the time taken to compute k.

S.No.	Program	Assertion	k	$f(k)$	Time (sec)
1	Min	Min	3	7	0.05613589286804199
2	Rel-Min-1	Rel-Min-1	4	9	0.058405160903930664
3	Rel-Min-2	Rel-Min-2	4	9	0.15024709701538086
4	Search	Search	4	9	0.043929815292358

Table 1: A table showing the values of k computed using Z3, that admit "removal of iterations" in the corresponding programs (cf. Section 3). Using the k's, the values $f(k)$ are computed (theoretically) that serve as the small model bounds for the corresponding programs and their natural assertions.

```
Second-min(n, a[n]):
+++++++++++++++++++
m1 = a[0];
m2 = a[0];
i = 0;
while (i < n)
{
    if (a[i] < m1)
    {
        m2 = m1;
        m1 = a[i];
    }
    else if ((m1 == m2 &&
                m2 < a[i]) ||
             (m1 < a[i] &&
                a[i] < m2))
    {
        m2 = a[i];
    }
    i++;
}
assert(A3);
```

```
Search(n, a[n], p):
+++++++++++++++++++
f = 0;
i = 0;
while (i < n)
{
    if (a[i] == p && f != 1)
        f = 1;
    i++;
}
assert(A4);
```

```
CopyArray(n, a[n], b[n]):
++++++++++++++++++++++++
i = 0;
while (i < n)
{
    b[i] = a[i];
    i++;
}
assert(A5);
```

assert( );

$$\begin{aligned}\texttt{A3} =& (\forall j_1 \forall j_2\ ((j_1 \geq 0 \land j_1 < n) \land (j_2 \geq 0 \land j_2 < n)) \rightarrow a[j_1] = a[j_2]) \lor \\ & (\forall j\ (j \geq 0 \land j < n) \rightarrow (x < a[j] \land y \leq a[j]) \lor (x = a[j] \land x < y)) \\ \texttt{A4} =& \forall j (j \geq 0 \land j < n \land a[j] = p) \rightarrow f = 1 \\ \texttt{A5} =& \forall j (j \geq 0 \land j < n) \rightarrow b[j] = a[j]\end{aligned}$$

Figure 4: On the left is a program computing the second minimum of a given array a (if there are at least two distinct elements in the array. On the right are two programs, one searching for a given number p in array a, and the other copying a given array b in array a. All these programs can be seen to be in MLL.

```
Sortedness-I(n, a[n]):
+++++++++++++++++++++
sorted = 1;
i = 1;
while(i < n)
{
    if (sorted != 0 &&
            a[i] < a[i-1])
        sorted = 0;
    }
}
i++;
}
assert(A5);
```

```
Sortedness-II(n, a[n]):
+++++++++++++++++++++++
sorted = 1;
m = a[0];
i = 0;
while(i < n)
{
    if (sorted != 0 && i != 0)
    {
        if (a[i] < m)
            sorted = 0;
    }
    m = a[i];
    i++;
}
assert(A5);
```

Figure 5: Two programs checking if a given array a is sorted. The program on the left is not in MLL but is in MLL+. This program can be algorithmically transformed to the equivalent (with respect to computation of the 'sorted' variable) program on the right which is in MLL.

6 Conclusion and Future Work

In this work, we have shown that a given piece of code that includes an assertion can be translated to a multi-sorted first order logic formula that can then be analysed automatically to check if it enjoys a small model property. We focused on the particular case of the min program that computes the minimum of a given array, and the min assertion that states that the value computed by the program is the minimum of the given array. We showed that the formulae corresponding the min program and assertion have small models whose sort sizes are bounded by 7. Consequently, a bounded model checker or SAT solver can be used to check if the program is correct with respect to the assertion. We have also presented a syntactic class of programs and assertions for which we believe our approach can be used to show that the small model property holds (if it does hold), and provided examples of interesting programs that belong to the class. Our proof technique involves showing that for any large model of the logical formulation of the given program and assertion, a small part of the model that represents a small number of iterations of the loop of the program, can be removed, and other parts of the model can be adjusted to get another model of the formulation that is smaller in size. This operation can be applied repeatedly to get a small model whose size can be determined algorithmically.

Although our technique has been shown to work for the min program, we have not yet established rigorously that it also works for our syntactic class of programs which are structurally close to the min program, though we believe this strongly. We would therefore like to prove this as a part of our future work. Beyond our syntactic class, we believe our technique can be extended to larger classes of programs, those that use interpreted operators, such as those in arithmetic. Again, we would like to extend our work to other classes of programs, particularly ones that modify the given input array (that currently we assume to be read-only), like the partition function of quicksort. These programs present new challenges that we would like to address as part of our future work.

References

[1] Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1

[2] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min ́e, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 85–108. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36377-7

[3] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min ́e, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the PLDI 2003, pp. 196–207. ACM, New York (2003)

[4] Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2 15

[5] Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1 40

[6] Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1 40

[7] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM, New York (1977)

[8] Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: SIGPLAN Not, vol. 46, no. 1, pp. 105–118, January 2011

[9] Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6 14

[10] Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proceedings of the POPL 2002, pp. 191–202. ACM, New York (2002)

[11] Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: SIGPLAN Not. vol. 40, no. 1, pp. 338–350, January 2005

[12] Halbwachs, N., P´eron, M.: Discovering properties about arrays in simple programs. In: SIGPLAN Not, vol. 43, no. 6, pp. 339–348, June 2008

[13] Jana, A., Khedker, U.P., Datar, A., Venkatesh, R., Niyas, C.: Scaling bounded model checking by transforming programs with arrays. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 275–292. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4 16

[14] Monniaux, D., Alberti, F.: A simple abstraction of arrays and maps by program translation. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 217–234. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9 13

[15] Monniaux, D., Gonnord, L.: Cell morphing: from array programs to array-free Horn clauses. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 361–382. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7 18

[16] Kumar, Shrawan, Amitabha Sanyal, and Uday P. Khedker. ”Value slice: A new slicing concept for scalable property checking.” Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21. Springer Berlin Heidelberg, 2015.

[17] Kumar, Shrawan, et al. ”Property checking array programs using loop shrinking.” Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24. Springer International Publishing, 2018.

[18] Börger, Egon, Erich Grädel, and Yuri Gurevich. The classical decision problem. Springer Science & Business Media, 2001.

[19] Turing, Alan Mathison. ”On computable numbers, with an application to the Entscheidungsproblem.” J. of Math 58.345-363 (1936): 5

[20] Trakhtenbrot, Boris (1950). ”The Impossibility of an Algorithm for the Decidability Problem on Finite Classes”. Proceedings of the USSR Academy of Sciences (in Russian). 70 (4): 569–572.

[21] Löwenheim, Leopold (1915), ”Über Möglichkeiten im Relativkalkül” , Mathematische Annalen, 76 (4): 447–470, doi:10.1007/BF01458217, ISSN 0025-5831, S2CID 116581304

[22] Nakshatra Gupta, Prajkta Kodavade, Shrawan Kumar, Abhisekh Sankaran, Akshatha Shenoy, R. Venkatesh. Verifying the min program using small models. Technical report, September 2023.

[23] Libkin, Leonid. Elements of finite model theory. Vol. 41. Heidelberg: springer, 2004.