Range Analysis for Software Verification

Aleksandr Zaks, Franjo Ivančić, Hari Cadambi, Ilya Shlyakter, Zijiang Yang, Malay Ganai, Aarti Gupta, and Pranav Ashar

NEC Laboratories America
4 Independence Way
Princeton, NJ 08540

Abstract. One of the main challenges of formal software verification is the ability to handle programs of realistic size. Model checking suffers from the state explosion problem which is even further exacerbated in the context of software verification. In this paper, we propose the use of lightweight, efficient and sound abstraction techniques to statically determine possible ranges for values of program variables. Such range information for each variable can be used to improve the efficiency of model checking software by providing a smaller state-space to be considered by the back-end verification engines such as those based on BDDs or SAT-solvers. Our main method is based on the framework suggested in [25] which formulates each analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint system to a linear program, which can then be analyzed by the available powerful computation engines targeted for linear programs. Furthermore, such range information can also be used to improve the efficiency of other popular analysis techniques such as predicate abstraction and counterexample-guided abstraction refinement. We also present a second approach to compute ranges which tries to exploit the fact that the range information, if only used in a bounded model checking run of depth k, does not have to be sound for all computations of the program, but only for traces up to length k. By concentrating on a bounded length trace only, we are able to find tight bounds on many program variables that cannot be bounded using the first technique. Both methods have been implemented and successfully used with our F-Soft [2] verification framework. Initial experiments with our range analysis techniques show promising results in extending the power of state-of-the-art software verification tools.

Keywords: Software Verification, Model checking, Bounded model checking, Predicate abstraction, Range analysis, Linear programs

1 Introduction

Model checking is an automatic technique for the verification of concurrent systems. It has several advantages over simulation, testing, and deductive reasoning, and has been used successfully in practice to verify complex sequential circuit designs and communication protocols [7]. However, model checking suffers from

the state explosion problem which is even further exacerbated in the context of software verification.

In this paper, we propose the use of lightweight, efficient and sound abstraction techniques to statically determine possible ranges for values of program variables. Such range information for each variable can be used to improve the efficiency of software verification tools by providing a smaller state-space to be considered by the back-end model checking engines such as those based on BDDs or SAT-solvers. Furthermore, such range information can be used to improve the efficiency of other popular analysis techniques such as predicate abstraction and counterexample-guided abstraction refinement. In this paper we believe to be the first to propose the use of range analysis techniques for verification as well as a *bounded range analysis* computation for traces of bounded depth only. We furthermore describe the implementation of these range analysis techniques in our prototype software analysis tool for C called F-Soft [2].

Range analysis using linear program solvers. We present several lightweight approaches of determining lower and upper bounds for program variables using range analysis techniques to be used in the back-end verification engines provided in the F-Soft tool. The range information can significantly help in reducing the state-space and thus later stages of verification, extending the reach of formal methods for software analysis. Our main method is based on the framework suggested in [25] which formulates each analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint system to a linear program, which can then be analyzed by the available powerful computation engines targeted for linear programs. The solution to the linear program provides symbolic lower and upper bounds for the values of all integer variables. This includes variables that are de-sugared by various preprocessing steps of our F-Soft tool into integer variables, in particular pointer variables and integer (or pointer) elements of composite structures. In addition to the framework presented in [25], we have also incorporated various contributions to the analysis described there. For example, we perform the analysis for certain constraint systems that contain both disjunctions and conjunctions of linear inequalities, instead of just conjunctions as proposed in [25]. Furthermore, our particular modeling framework and verification application allow us to make certain simplifying assumptions that translate to a more efficient overall framework for computing tight lower and upper bounds on variables.

We use a publicly available linear program solver to solve the generated constraint systems. The solver is able to distinguish between purely linear programs (LP), where all variables are assumed to be rationals, and mixed-integer linear programs, also called MLP, where some variables are integers. The runtime for LP problems is polynomial, while MLP is NP-complete. The addition of disjunctions into the constraint system is resolved through an encoding that uses integer variables thus requiring us to solve an MLP problem. However, it should be noted that in practice the runtime largely depends on the number of integer variable. Through the use of various heuristics limiting the number of disjunctions, we are able to keep the number of integer variables very small. Additionally, in worst

case we can always fall-back to omit additional disjunctions from the constraint system through the use of various heuristics to pick which constraint to satisfy. Furthermore, in contrast to [25], we perform the range analysis purely as a pre-processing step for an even harder task – the verification of software. Since the verification is the actual bottleneck of the analysis, we allow the range analysis tool to perform its computation completely. In all our experiments so far, we have always been able to complete the range analysis computation within less than half a minute.

Range analysis for predicate abstraction. In the world of software verification, predicate abstraction has emerged to be a powerful and popular technique for extracting finite-state models from complex, potentially infinite state systems [3, 4, 11, 12, 15, 20, 22]. Tools such as Bandera [10], SLAM [4], Blast [16] and Feaver [17] have successfully applied a predicate abstraction paradigm for the analysis of C or Java programs. In contrast to the early work on predicate abstraction for software (such as [3, 16]), which uses extensive and inherently expensive calls to various theorem provers to compute the transition relations in the abstract system, F-Soft follows a more efficient SAT-based approach to compute the transition relation [9, 19]. The SAT-based enumeration can be further improved by providing tighter ranges for the various concrete variables using the techniques presented here, in comparison to the previous approaches which use full bit-blasting for concrete variables.

Range analysis for bounded model checking. While symbolic model checking algorithms using BDDs offer the potential of exhaustive coverage of large state-spaces, it often does not scale well enough in practice. An alternative approach is *bounded model checking* or BMC [5] focusing on the search for counter-examples of bounded length only. Effectively, the problem is translated to a Boolean formula, such that the formula is satisfiable if and only if there exists a counter-example of length k. In practice, k can be increased incrementally starting from one to find a shortest counter-example if one exists. However, additional reasoning is needed to ensure completeness of the verification when no counter-example exists [18, 26]. The satisfiability check in the BMC approach is typically performed by a back-end *SAT-solver*. Due to the many advances in SAT-solving techniques [13, 14, 21, 23], BMC can often handle much larger designs than BDDs.

Since the technique to compute ranges for program variables uses a linear program solver, we need to over-approximate certain program constructs in a safe manner which results in the most conservative bounds. The most conservative bound for a variable of type `int` typically corresponds to a 32 bit representation; that is, -2^{31} to $2^{31} - 1$. A second approach to compute ranges tries to exploit the fact that the range information, if used only in a bounded model checking run of depth k, does not have to be sound for all computations of the program, but only for traces up to length k. By concentrating on a bounded length trace only, we are able to find tight bounds on many program variables that cannot be bounded using the first technique.

Outline. In the next section we first give some background information on our software modeling approach. Section 3 then presents the generation of the constraint system, while section 4 discusses the analysis of the constraint system using a linear program solver. Section 5 then discusses our bounded range analysis technique. We present our initial experimental results for the use of range analysis for software verification in Section 6, before we end this paper with some concluding remarks.

2 Software Modeling

In this section, we briefly describe our software modeling approach that is centered around basic blocks as described in [2]. We first use certain code preprocessing steps to simplify the source code. For example, code simplification removes nested or embedded function calls inside other function calls by adding temporary variables. We then extract the control flow graph (CFG) of the source code. During the computation of the CFG, an edge from the calling block to the first block of the called function is created. If the function call returns a value, we add a statement assigning the return expression to the assigned variable. For functions that are not called recursively, we add statements that assign actual parameters to the corresponding formal parameters if parameters are needed. For non-recursive functions the return point of the called function in the program is recorded as a special parameter. The returning transitions from the function call are guarded with checks on this special parameter. To allow modeling of bounded recursion, we include a bounded function call stack, which is used to save the current local state and determine which basic block to return to. With respect to the subset of C that we currently consider, we support primitive data types, pointers, static arrays and records. As to dynamically created data structures such as dynamic arrays and linked-lists, an upper bound on the length is required and should be provided by the user. Furthermore, all control flow logic constructs are completely supported. A program counter (PC) variable is then introduced to monitor progress in the control flow graph for verification purposes.

An example for our control logic modeling is shown in Figure 1, where the right side shows the computed CFG for the simple C code on the left side. The example shows how the basic blocks are computed for various types. Each basic block is identified by a unique number shown inside the hexagon adjacent to the basic block. The source node of the CFG is basic block 0, while the sink node is the highlighted basic block 8. The example in Figure 1 pictorially shows how non-recursive function calls are included in the control flow of the calling function. A preprocessing analysis determines that function `foo` is not called in any recursive manner. The two return points are recorded by an encoding that passes a unique return location as a special parameter using the variable `rtr`.

The following list contains the main features that affect our range analysis computations presented in this paper.

```
int foo(int s){
  int t=s+2;
  if (t>6)
    t -= 3 ;
  else
    t--;
  return t;
}

void bar(){
  int x = 3, y = x - 3;
  while ( x <= 4 ){
    y++ ;
    x = foo(x);
  }
  y = foo(y);
}
```

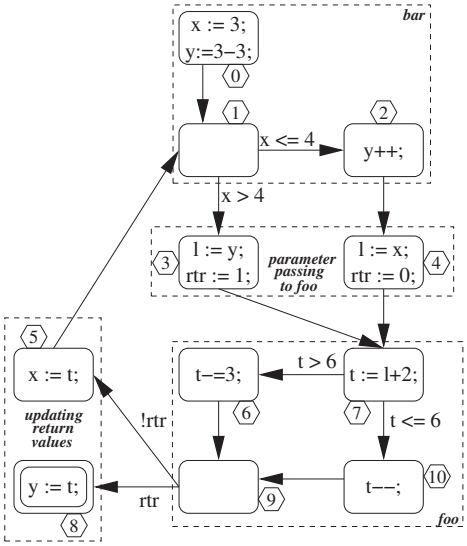

Fig. 1. Computing the control flow graph

- We compute the control flow graph of the program, perform program slicing with respect to the property under consideration [6], and guarantee that each variable is updated at most once in each basic block.
- All the functions are uniquely inlined, that is each function is inlined at most once. However, this allows us to focus only on intra-procedural analysis instead of the more complicated inter-procedural analysis described in [25]. This also allows us to handle return values of function calls which is not possible in the original framework suggested in [25].
- Memory allocations are modeled using a finite heap. In general, the preprocessing of pointers creates additional variables which are de-sugared to be of type `unsigned int`. Fortunately, the later can be successfully exploited by the range analysis techniques discussed here.

The result of the preprocessing described here produces a single control flow graph that includes all function calls, which serves as an actual input to our tool. A more detailed description of our modeling approach can be found in [2].

3 Constraint System Generation

The first step of the range analysis is the generation of a symbolic constraint system. In this section we describe the generation of this constraint system, which will then be analyzed by a linear program solver as described in Section 4. There are two basic rules to follow:

– For each assignment, update the bounds at the corresponding basic block of the variable on the left hand side with the bounds of the expression found on the right hand side.
– If a transition can be taken from a block B_i to some block B_j, a range of a variable v at the beginning of block B_j must include all possible values the variable v can have right before such a transition. In particular, the rule enforces that if a transition from B_i to B_j can always be taken (no guard), then the range of a variable v at the beginning of the block B_j must include the range of v at the end of the block B_i.

3.1 Symbolic Bounds

For each basic block B_i in the control-flow graph of a procedure f we define two program locations: pre_i to represent the start of the basic block B_i and $post_i$ to represent the end of the block. The set of local integer variables after preprocessing of the code, that is including pointer variables, of the procedure f is denoted by V_f. A variable v such that $v \in V_f$ is called *a range variable*. $P_f \subseteq V_f$ is a set of formal parameters of procedure f that are defined to be integers. We use v_{loc} to denote the value of the variable v at the program location loc. For each formal parameter p we use p_0 to symbolically represent the value of the actual parameter that corresponds to p. Similar to [25] we focus the rest of our discussion to the case when the values of the actual parameters are positive. It should be noted that the set P_f of formal parameters is often empty and generally small, since we consider one control flow graph with a single entry basic block. Additionally, since we consider one entry function f we only need to consider one set V_f and one set P_f.

For each variable $v \in V_f$ and a program location loc let L^v_{loc} and U^v_{loc} represent the lower and upper bounds of the value of v at the program location loc. We initially set L^v_{loc} and U^v_{loc} to be a linear combination of the parameters of f with unknown rational coefficients. Formally, we define $L^v_{loc} = C + \sum_{p \in P_f} C_p \cdot p_0$. We can similarly define U^v_{loc}.

We define $l(e, loc)$ to represent the lower bound of an expression e at location loc. We compute $l(e, loc)$ for a constant c, a variable v and expressions e, e_1 and e_2 as follows:

$$
\begin{aligned}
l(c, loc) &= c \\
l(v, loc) &= L^v_{loc} \\
l(e_1 + e_2, loc) &= l(e_1, loc) + l(e_2, loc) \\
l(c \cdot e, loc) &= \begin{cases} c \cdot l(e, loc) & c \geq 0 \\ c \cdot u(e, loc) & c < 0 \end{cases}
\end{aligned}
$$

Whenever we cannot compute a bound we let $l(e, loc) = -\infty$, where "$-\infty$" correspond to the most conservative lower bound (the minimum value) of the particular integer data type used. As an example, for `unsigned int` we would use 0 as the most conservative lower bound. Similarly, we can define $u(e, loc)$ for upper symbolic bounds of expressions.

3.2 Constraint Types

We generate *initialization constraints* for the location pre_0 that represents the beginning of the initial basic block B_0. For each $p \in P_f$ we require that $L^p_{pre_0} = U^p_{pre_0} = p_0$. For each $v \in V_f \backslash P_f$ we require that $L^v_{pre_0} = -\infty$ and $U^p_{pre_0} = +\infty$, if the user does not specify additional environmental constraints (on global variables, for example). Recall, that we use the symbols "$-\infty$" and "∞" to represent various constants depending on the actual type of the variable. For an assignment within block B_i of the form $v = e$, where $v \in V_f$, we generate the following *assignment constraint*: $L^v_{post_i} = l(e, pre_i) \bigwedge U^v_{post_i} = u(e, pre_i)$. Assignment constraints define the bounds after execution of the expressions in a basic block. In case a variable v is not reassigned in block B_i, we generate the following *propagation constraint*: $L^v_{post_i} = L^v_{pre_i} \bigwedge U^v_{post_i} = U^v_{pre_i}$. These constraints are used to propagate the bounds for variables that are not changed inside a basic block.

Whenever we can make a transition from a basic block B_i to a basic block B_j we require that for every $v \in V_f$, the range of v at the beginning of B_j includes the range of v at the end of B_i. Formally we add the following *flow constraint* to the constraint system: $L^v_{pre_j} \leq L^v_{post_i} \bigwedge U^v_{pre_j} \geq U^v_{post_i}$.

3.3 Handling of Conditionals

The constraint system as defined in the previous paragraph, consisting of initialization, assignment, propagation and flow constraints is comprehensive enough to ensure the soundness of the bounds that are solutions of the constraint system. However, the information implied by conditionals (guards) in the program or control flow graph may help to minimize the resulting ranges. For example, consider the case that the range of a variable v before a conditional is $v \in [0, 100]$, but the condition guarding the transition to a new block is $v \geq 20$. If there is no other incoming edge to the new block, then the actual lower bound for v in the new block can be safely assumed to be 20 and not 0.

In general, consider a transition from a basic lock B_i to a basic block B_j and a guard of the form $v \geq e$, where $v \in V_f$. Assume that the following constraint is satisfied: $L^v_{pre_j} \leq l(e, post_i)$. Then, whenever we can make a transition from B_i to B_j we are guaranteed that the lower bound of v at the beginning of B_j is less or equal to the value of v at the end of B_i at the time of the transition. So, we can relax the corresponding flow constraint to:

$$L^v_{pre_j} \leq L^v_{post_i} \bigvee L^v_{pre_j} \leq l(e, post_i).$$

Often we can omit the flow constraint altogether. However, since we do not know a priori the relationship between $L^v_{post_i}$ and $l(e, post_i)$ we introduce a disjunction that ultimately results in higher precision. Other comparison operators $\{\leq, <, >\}$ can be handled in a similar fashion. Note, that we may first need to solve the conditional in terms of a specific variable before generating the constraint.

However, the addition of disjunctions into an otherwise purely conjunctive constraint system presents a challenge to the general approach advocated here.

As mentioned earlier, we are hoping to gain from the powerful computation engines provided by linear program solvers to perform an efficient analysis of the linear constraint system. Adding disjunctions into the constraint system prevents us to use pure LP solvers where all variables are rational, since disjunctions are not linear. Therefore, the original work suggested in [25] does not allow the addition of disjunctions into the constraint system. We postpone a more detailed discussion of our handling of such disjunctions in the constraint system until Section 4.1.

As a short example, consider the following code for variables x and y of type `unsigned int`:

```
while (x ≤ y + 1){
  y − −;
}
```

The condition $x + 1 \leq y$ can be solved in terms of both x and y, generating two constraints: $L^y_{pre_j} = l(x - 1, post_i)$ as well as $U^x_{pre_j} = u(y + 1, post_i)$. In general, at most one of the constraints generated from a conditional is useful; the other will often be ignored by satisfying the corresponding flow constraint. In our example, we want to use the condition and bound y from below by $x - 1$. Otherwise, from the point of view of range analysis, the loop is equivalent to:

```
while (true){
  y − −;
}
```

and in such a case we have to set the lower bound of y to the most conservative value.[1] On the other hand, bounding x from above by $y + 1$ may be useless if y was unbounded before.

In contrast to our approach, the original algorithm described in [25] would simply generate only one constraint per conditional, bounding the variable on the left hand side by the right hand side. For the above example, $U^x_{pre_j} = (y + 1, post_i)$ is added to the system of constraints, regardless of the body of the `while`-loop. Also, since the approach in [25] does not allow disjunctions, this constraint is actually used instead of the corresponding flow constraint. Clearly, the constraint system does not gain any precision from the conditional by doing this.

Moreover, if we change the body of the loop to read $y = \text{sqrt}(y)$, where sqrt represents the (non-linear) square root function defined for `unsigned int`, using the conditional would adversely affect the upper bound of x. Since, in this scenario, y is updated using a non-linear function, it is therefore unbounded. In

[1] This example brings up an orthogonal issue also, namely handling of over- and underflow of variables. Since we use our range analysis tool only as a preprocessing tool for a followup verification run, we do not consider over- or underflows at this point. Thus, the lower bound of y in this sample code fragment would be $-\infty$, while the upper bound may or may not be ∞ depending on other constraints in the constraint system. The verification using F-Soft can then check independently whether an over- or underflow can possibly occur.

particular, its upper bound then becomes ∞. In the approach used in [25], this requires the upper bound of x to be greater than or equal to the upper bound of $y + 1$, which would force x to be unbounded for the upper bound as well. However, using the approach described here, we would still be able to satisfy the flow constraint instead of the conditional, thus finding that the upper bound of x does not change during processing of the `while`-loop.

As an example, consider the program presented in Figure 2. The corresponding control flow graph as well as the generated symbolic constraint system generated for it are shown in Figure 3. We omit the generated constraints for the lower bounds in order not to clutter the figure.

```
void foo(int N){
  int i = 0;
  while( i ≤ N){
    i++;
  }
}
```

Fig. 2. A Sample Program

3.4 Objective Function

As mentioned before, the generated constraints guarantee the soundness of the resulting ranges. However, we are interested in the optimal, most precise range information. Therefore, we add the following objective function to the linear program, which minimizes the total number of values to be considered:

$$\sum_{v \in V_f} \sum_{B_i \in R_v} | U^v_{pre_i} - L^v_{pre_i} |,$$

where $R_v = \{B_i \in B \mid v \text{ is read in block } B_i\}$.

3.5 Constraint System Simplification

We perform several fast constraint system simplifications before we invoke the linear program solver. In the following we present our simplifications of the constraint system, which are used to reduce the number of variables and constraints in the system:

- If a variable v is not reassigned in a block B_i instead of generating a propagation constraint of the form $L^v_{pre_j} \leq L^v_{post_i} \bigwedge U^v_{pre_j} \geq U^v_{post_i}$, we can just replace L_{post_i} and U_{post_i} with L_{pre_j} and U_{pre_j}, in all linear inequalities in our constraint system. This eliminates two variables from the constraint system.
- If a block B_j has only one parent B_i instead of generating a flow constraint of the form $L^v_{pre_j} \leq L^v_{post_i} \bigwedge U^v_{pre_j} \geq U^v_{post_i}$, we can just replace $L^v_{pre_j}$ and $U^v_{pre_j}$ with $L^v_{post_i}$ and $U^v_{post_i}$ in our constraint system.

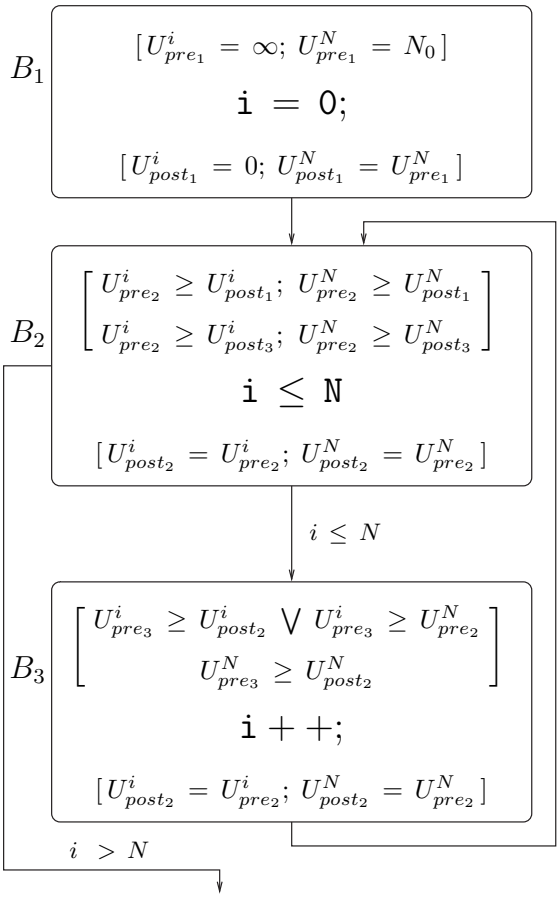

Fig. 3. Symbolic Constraint System

– If all parents of a block B_j share the same symbolic expression L^v_{post} for the lower bound of some variable v, we can just replace L_{pre_j} with L^v_{post}. The same rule can be applied for upper bounds.
– Consider a variable v and a cycle in the control flow graph. Assume the cycle does not include any edges with guards that refer to v. In addition, assume none of the basic blocks in the cycle reassigns v. Then we can use the same symbolic expression for the lower bound of v for all the locations within the cycle. The same applies for upper bounds.

3.6 Constraint System Decomposition

Intuitively, it is clear that some bounds are independent of some other bounds. To formalize this notion we follow [25] and introduce a dependency graph of bounds. The nodes in the graph represent bounds. For a block B_i and a variable v there are exactly 4 nodes in the graph that correspond to $L^v_{pre_i}$, $U^v_{pre_i}$, $L^v_{post_i}$, and $U^v_{post_i}$. For every generated constraint in the constraint system, there is an edge from the node that represents the bound on the left hand side to any node that represents a bound from the right hand side. We then decompose the graph to strongly connected components and process each component separately in the reverse topological order. There are two major benefits of the constraint system decomposition:

– Efficiency is substantially improved since solving an MLP problem is NP-complete and it is thus easier to deal with smaller components. Of course, in practice, a good MLP solver would perform some decomposition anyway. The only advantage is that we have a natural orientation of the edges in the dependency graph. In particular, for an assignment constraint, the left hand side of the assignment depends on the right hand side, not the other way around. The MLP solver cannot make such a distinction.
– Unboundedness of any bound variable usually forces the whole system to be declared infeasible. Decomposition prevents the propagation of such effects between unrelated variables. This allows us to find good bounds for many variables even though there are some other variables that cannot be bounded by this approach.

4 Analyzing a Constraint System

Even without any modifications to the generated constraint system, it resembles a linear program. There are, however, several important distinctions. First of all, lower and upper bounds are linear expressions with unknown rational coefficients, not just variables. Of course in the case when f does not have any parameters the symbolic expressions are reduced to rational variables and can be directly used as linear program variables. The second important difference is that, in general, linear program solvers do not support arbitrary Boolean connectives, but rather there is an implied conjunction of all constraints. For our algorithm to work, we need to handle disjunctions as well.

Consider a symbolic constraint of the form $L^v_{loc'} \leq L^v_{loc}$, where $L^v_{loc} = C + \sum_{p \in P_f} C_p \cdot p_0$ and $L^v_{loc'} = C' + \sum_{p \in P_f} C'_p \cdot p_0$. For such a case, we generate the following linear inequality constraint that can be directly submitted to the linear program solver:

$$C' \leq C \wedge \Big(\bigwedge_{p \in P_f} C'_p \leq C_p \Big).$$

Assuming positivity of parameters, the new constraint is actually stronger than the original one. Hence, the transformation preserves the soundness of the bounds. Other constraints can be handled the same way. For cases where we cannot assume positivity of the parameters, we may need to perform an inefficient case split of the analysis for the possible combinations of positive and negative parameters.

Similarly we convert a symbolic objective function of the constraint system into a linear program objective function. Assuming that $L^v_{pre_i} = X^v_{pre_i} + \sum_{p \in P_f} X^v_{pre_i,p} \cdot p_0$ and $U^v_{pre_i} = Y^v_{pre_i} + \sum_{p \in P_f} Y^v_{pre_i,p} \cdot p_0$, we then rewrite the objective function $\sum_{v \in V_f} \sum_{B_i \in R_v} \mid U^v_{pre_i} - L^v_{pre_i} \mid$ to

$$\sum_{v \in V_f} \sum_{B_i \in R_v} \mid (Y^v_{pre_i} - X^v_{pre_i}) + \sum_{p \in P_f} (Y^v_{pre_i,p} - X^v_{pre_i,p}) \mid .$$

4.1 Handling Disjunctions

As described earlier in this paper, the addition of disjunctions into an otherwise purely conjunctive constraint system presents a challenge to the approach using linear program solvers. One possibility to handle disjunctions is to replace inequalities with propositions and use a SAT-solver to find all satisfying assignments. Each particular assignment corresponds to one LP problem without any disjunctions. This is the so called "lazy" approach and it works quite well for many problems. Unfortunately, in the context of our range analysis this algorithm would degenerate to pure case splitting; that is, we would need to consider all possible combinations.

Another approach would be to approximate linear inequalities with separation logic [24] formulas and use a fast decision procedure like TSAT++ [1] to find a satisfying assignment. The main drawback of this approach is that the propositional assignment is usually not optimal, although a solution would guarantee some bounded ranges. However, as long as our linear inequalities are "close" to their separation logic counterparts the assignment guarantees the feasibility of the resulting MLP problem.

The approach that is actually used in our tool is based on encoding disjunctions via integer variables. We also employ several heuristics beforehand to reduce the number of disjunctions.

We describe our approach using a small example. Consider the following constraint: $L^v_{pre_j} \leq L^v_{post_i} \bigvee L^v_{pre_j} \leq l(e, post_i)$ We introduce two new binary variables D_1 and D_2 and M denotes some large positive number. Our original

constraint is then replaced with the following constraint:

$$\big(D_1 + D_2 \leq 1\big) \bigwedge \big(L^v_{pre_j} + M \cdot D_1 \leq L^v_{post_i}\big) \bigwedge \big(L^v_{pre_j} + M \cdot D_2 \leq l(e, post_i)\big).$$

The new constraint is stronger than the original one, and the two constraints are actually equivalent if M is sufficiently large. Note that the newly introduced variables D_1 and D_2 are the only variables that need to be pure integer variables, while all others are still assumed to have rational values. Without disjunctions the constraint system can be solved much easier using a LP solver.

In the following we briefly describe some of the heuristics we employ to resolve some disjunctions before we invoke the appropriate linear program solver for the resulting constraint system.

- *Drop a constraint if some bound on the right hand side is close to M.* For several practical matters, mostly due to lack of necessary precision of floating-point numbers, we may not be able to set M as high as we wish. If we were not to drop such disjunctive constraints, we may actually treat the disjunctive constraint as a conjunctive constraint instead. Therefore, we resolve this problem by removing the particular constraint that causes this problem from the disjunction.
- *Drop a constraint if some bound on the right hand side has not yet been determined.* As in the previous case the unknown bound value might be close to M and we would have the same issue that was described in the previous heuristic. Furthermore, if the unknown value cannot be bounded, the whole constraint system would be declared infeasible and thus unbounded.
- *Prefer to satisfy the flow constraint.* It is clear, that we cannot drop both sides of a disjunction. That is, if the aforementioned rules ask for both constraints of a disjunction to be removed, we need to keep at least one of the two sides in the resulting constraint system. In such a case, we prefer to leave the flow constraint in the constraint system, since it refers to one program variable only.

5 Bounded Range Analysis

Recently, there has been a growing interest in utilizing Bounded Model Checking(BMC) for program verification [2, 8]. We propose the idea of *bounded range analysis*, which tries to compute ranges by exploiting the fact that the range information, if only used in a bounded model checking run of depth k, does not have to be sound for all computations of the program, but only for traces up to length k. By concentrating on a bounded length trace only, we are able to find tight bounds on many program variables that cannot be bounded using the linear program based technique described earlier. As an example, consider the following code:

```
int i = 0;
while(true){
  i++;
}
```

If one were to consider all possible traces, then the upper bound for i would have to declared unbounded. However, if we are only concerned with the traces up to k steps, it is safe to conclude that the value of i will always be in the range from 0 to k.

5.1 Bounded range analysis via iterative updates

A straightforward way to compute such ranges is to perform a BFS on the control flow graph with depth limit set to k which updates the lower and upper bounds for the individual basic blocks. Although this approach results in very precise ranges, it may not be very efficient with respect to runtime for large k. We also propose the following algorithm that can be easily implemented on top of the aforementioned constraint based approach and, for a fixed number of steps, has a quadratic runtime in terms of the size of the code:

```
Intialize all the bounds to the least conservative values;
for(i = 0; i < #steps; i++){
  foreach basic block B_j{
    foreach variable v, v ∈ V_f{
      update L^v_pre_j, U^v_pre_j, L^v_post_j, and U^v_post_j using constraints
    }
  }
}
```

This simple algorithm can be further improved in several ways, in particular to support non-linear functions. In case a function does not have any parameters, we can easily extend the algorithm to support many important non-linear functions. The restriction on the presence of parameters is not severe. Expressing bounds as a linear combination of parameters is mostly useful for inter-procedural analysis. Since F-Soft inlines all function calls, parameters of called functions can be ignored. Consider an assignment $y = x^2$ in a block B_i. The following rules can be used to update $L^y_{post_i}$ and $U^y_{post_i}$: $L^y_{post_i} = 0$ and $U^y_{post_i} = \max(\mid L^x_{pre_i} \mid, \mid U^x_{pre_i} \mid)^2$.

5.2 Bounded range analysis via underapproximation

Another approach to bounded range analysis relies on the ability of SAT solvers to identify, in an unsatisfiable problem, constraints that are irrelevant to proving unsatisfiability. To a BMC problem instance, we can add provisional constraints restricting ranges of variables. If a counterexample is found with the provisional constraints in place, that counterexample is valid. If the instance is found unsatisfiable, we can check which of the provisional constraints were actually used to prove unsatisfiability. If some constraints weren't used, that means they represent valid assumptions on variable ranges. (Constraints not used for the final unsatisfiability proof may still speed up search.)

If some of the provisional constraints were used in showing unsatisfiability, these constraints must be relaxed and the analysis repeated. To reduce the number of repeats, several increasingly more stringent provisional constraints can be added simultaneously. For an integer variable `i`, the constraints `i<2`, `i<4`, `i<8` can be added simultaneously; a constraint is valid if the constraint and all less stringent constraints weren't used in the unsatisfiability proof. A collection of increasingly stringent constraints on an integer variable can be compactly encoded as unit clauses asserting that the higher-order bits of the variable are zero. If all unit clauses asserting falsity of bits higher than k are not used for unsatisfiability proof, then $i < 2^k$ is a safe assumption to make.

In the scheme described here, the Boolean encoding of a BMC problem takes into account the analyzed property. The resulting ranges may thus be smaller than if only program semantics were taken into account: traces that utilize higher ranges but satisfy the property may be quickly ruled out without using the provisional constraints.

The approach described in this subsection has not been implemented yet, but is planned for implementation.

6 Experiments

We have implemented our technique in the F-Soft verification platform for the analysis of C source code. We have used our first range analysis technique to perform an unbounded and sound analysis of four verification benchmarks. The benchmarks that we used are the network protocol PPP [27], an air-traffic controller (TCAS), and two device drivers (FLOPPY and NEC-1).

The experimental results are summarized in Table 1. We perform our analysis after some preprocessing steps as described earlier. It should be noted that we also use program slicing [6] to focus our analysis only on the parts of the program relevant to the property under consideration. The table includes the following information on the preprocessed code:

- The column *#Variables* describes the number of used variables in the analyzed program.
- The column *#Bits Saved* describes how many bits were removed from a bit-level description of the program after performing range analysis compared to a standard bit-level translation. The number of bits saved assumes for example, that a variable of type `int` or `unsigned int` would be represented by 32 bits, when no range analysis is performed or when the linear program solver cannot find appropriate smaller ranges for this variable.
- The column *Reduction Ratio* shows the percentage of bits saved by the use of our range analysis technique compared to a standard bit-level translation of the program, as done in CBMC [8].

Category	Benchmark	#Variables	#Bits Saved	Reduction Ratio
Network Protocol	PPP	132	2697	94%
Air Traffic Control	TCAS	81	1624	82%
Device Driver	FLOPPY	316	866	96%
Device Driver	NEC-1	172	1263	83%

Table 1. Range Analysis Benchmarks.

7 Conclusions

In this paper, we proposed the use of lightweight and efficient range analysis techniques for improving the performance of software verification tools. Our main method formulates the range analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint to a linear program. A second approach to compute ranges exploits the fact that the range information, if only uses in a bounded model checking run, needs to only be sound for traces of the program up to some depth bound. Both methods have been implemented in our F-Soft [2] verification framework. We also presented experimental results on a variety of verification benchmarks.

References

1. A. Armando, C. Castellini, E. Giunchiglia, M. Idini, and M. Maratea. TSAT++: an open platform for satisfiability modulo theories. In 2^{nd} *workshop on Pragmatics of Decision Procedures in Automated Reasoning*, July 2004.
2. P. Ashar, M. Ganai, A. Gupta, F. Ivančić, and Z. Yang. Efficient SAT-based bounded model checking for software verification. In *1st International Symposium on Leveraging Applications of Formal Methods*, 2004.
3. T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.
4. T. Ball and S. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 2000 Workshop on Model Checking of Software*, LNCS 1885, pages 113–130. Springer, 2000.
5. A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 317–320, 1999.
6. E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Programs slicing for VHDL. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):125–137, October 2002.
7. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. MIT Press, 2000.
8. E.M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. In *Proc. of the Model Checking for Dependable Software-Intensive Systems Workshop, San-Francisco, USA*, 2003.
9. E.M. Clarke, M. Talupur, H. Veith, and D. Wang. SAT based predicate abstraction for hardware verification. In *Sixth International Conference on Theory and Applications of Satisfiability Testing*, May 2003.

10. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of 22nd International Conference on Software Engineering*, pages 439–448. 2000.
11. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
12. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification*, LNCS 1633, pages 160–171. Springer, 1999.
13. M.K. Ganai, L. Zhang, P. Ashar, and A. Gupta. Combining strength of circuit-based and CNF-based algorithms for a high performance SAT solver. In *Design Automation Conference*, 2002.
14. E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.
15. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.
16. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
17. G.J. Holzmann and M.H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.
18. D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *4th Intern. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *LNCS*, pages 298–309. Springer, January 2003.
19. S.K. Lahiri, R.E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Computer Aided Verification*, volume 2725 of *LNCS*. Springer, 2003.
20. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design Volume 6, Issue 1*, 1995.
21. J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.
22. M.O. Möller, H. Rueß, and M. Sorea. Predicate abstraction for dense real-time systems. In E. Asarin, O. Maler, and S. Yovine, editors, *Theory and Practice of Timed Systems*. Springer-Verlag, 2002.
23. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.
24. V. Pratt. Two easy theories whose combination is hard. Technical report, 1977.
25. R. Rugina and M.C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, 2000.
26. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT solver. In *Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, November 2000.
27. W. Simpson. PPP: The Point-to-Point Protocol. RFC 1661, IETF, June 1994.