F-Soft: Software Verification Platform

F. Ivančić, Z. Yang[⋆], M.K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar[⋆⋆]

NEC Laboratories America, 4 Independence Way,
Suite 200, Princeton, NJ 08540
fsoft@nec-labs.com

1 Introduction

In this paper, we describe our verification tool F-Soft which is developed for the analysis of C programs. Its novelty lies in the combination of several recent advances in formal verification research including SAT-based verification, static analyses and predicate abstraction. As shown in the tool overview in Figure 1, we translate a program into a Boolean model to be analyzed by our verification engine DiVer [4], which includes BDD-based and SAT-based model checking techniques. We include various static analyses, such as computing the control flow graph of the program, program slicing with respect to the property, and performing range analysis as described in Section 2.2. We model the software using a Boolean representation, and use customized heuristics for the SAT-based analysis as described in Section 2.1. We can also perform a localized predicate abstraction with register sharing as described in Section 2.3, if the user so chooses. The actual analysis of the resulting Boolean model is performed using DiVer. If a counter-example is discovered, we use a testbench generator that automatically generates an executable program for the user to examine the bug in his/her favorite debugger. The F-Soft tool has been applied on numerous case studies and publicly available benchmarks for sequential C programs. We are currently working on extending it to handle concurrent programs.

2 Tool Features

In this section, we describe the software modeling approach in F-Soft and the main tool features. We perform an automatic translation of the given program to a Boolean model representation by considering the control flow graph (CFG) of the program, which is derived after some front-end simplifications performed by the CIL tool [9]. The transitions between basic blocks of the CFG are captured by control logic, and bit-level assignments to program variables are captured by data logic in the resulting Boolean model. We support primitive data types, pointers, static arrays and records, and dynamically allocated objects (up to a user-specified bound). We also allow modeling of bounded recursion by

[⋆] The author is at Western Michigan University.
[⋆⋆] The author is now with Real Intent.

including a bounded function call stack in the Boolean model. Assuming the program consists of n basic blocks, we represent each block by a label consisting of $\lceil \log n \rceil$ bits, called the program counter (pc) variables. A bounded model checking (BMC) analysis is performed by unrolling the block-wise execution of the program. Similar block-based approaches have been explored in a non-BMC setting for software model checking [3, 11]. However, by incorporating basic-block unrolling into a SAT-based BMC framework, we are able to take advantage of the latest SAT-solvers, while also improving performance by customizing the SAT-solver heuristics for software models. More details of our software modeling approach can be found in [6].

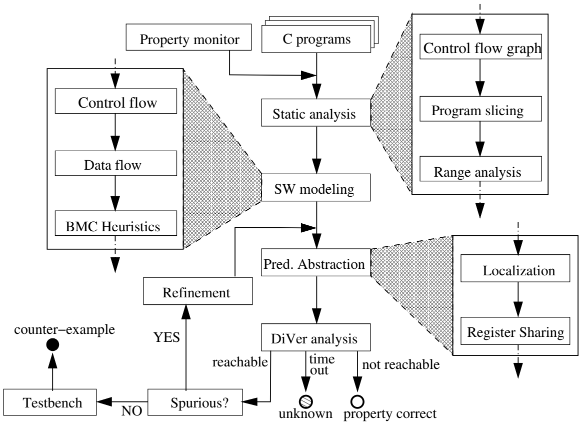

Fig. 1. F-Soft tool overview

2.1 Customized Heuristics for SAT-Based BMC

The Boolean models automatically generated by our software modeling approach contain many common features. We have proposed several heuristics in order to improve the efficiency of SAT-based BMC on such models [6]. In particular, a useful heuristic is a one-hot encoding of the pc variables, called selection bits. A selection bit is set if and only if the corresponding basic block is active. This provides a mechanism for word-level, rather than bit-level, pc decisions by the SAT solver. Furthermore, by increasing the decision score of the selection bits (or the pc variable bits), in comparison to other variables, the SAT-solver can be guided toward making decisions on the control location first. We also add constraints obtained from the CFG, to eliminate impossible predecessor-successor basic block combinations. These constraints capture additional high-level information, which helps to prune the search space of the SAT-solver.

Experimental results for use of these heuristics on a network protocol case study (checking the equivalence of a buggy Linux implementation of the Point-to-Point protocol against its RFC specification) are shown in Figure 2. For these experiments, the bug was found by SAT-based BMC at a depth of 119, and the figure shows the cumulative time in seconds up to each depth. All experiments were performed on a 2.8GHz dual-processor Linux machine with 4GB of memory. The graph labeled *standard* represents the default decision heuristics implemented in the DiVer tool, while the other three graphs show the effect of specialized heuristics – higher scoring of pc variables (*score*), one-hot encoding of pc variables (*one-hot*), and addition of constraints for CFG transitions (*trans*). The advantage of the the one-hot encoding heuristic can be seen clearly.

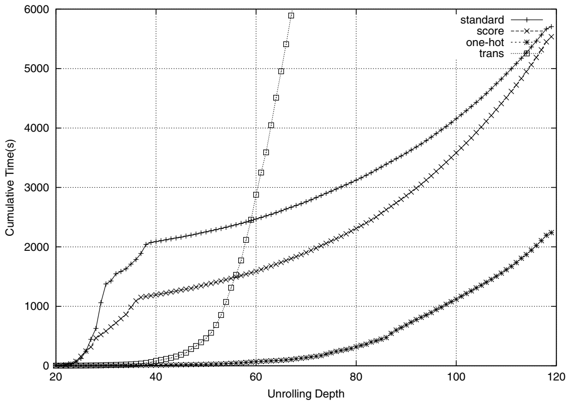

Fig. 2. Cumulative time comparison of SAT-based BMC heuristics for PPP

2.2 Range Analysis

F-Soft includes several automatic approaches for determining lower and upper bounds for program variables using range analysis techniques [12]. The range information can help in reducing the number of bits needed to represent program variables in the translated Boolean model, thereby improving the efficiency of verification. For example, rather than requiring a 32 bit representation for every `int` variable, we can use the range information to reduce the number of bits for these variables. As discussed in the next section, F-Soft also provides an efficient SAT-based approach for performing predicate abstraction. In this context too, SAT-based enumeration for predicate abstraction [8] can be improved by using tighter ranges for concrete variables, derived by using our automatic range analysis techniques.

Our main method is based on the framework suggested in [10], which formulates each range analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint system to a linear program (LP), such that the solution provides symbolic lower and upper bounds for the values of all integer variables. This may require over-approximating some program constructs to derive conservative bounds. Our second approach to computing ranges exploits the fact that in a bounded model checking run of depth k, the range information needs to be sound only for traces up to length k. This *bounded range analysis* technique is able to find tighter bounds on many program variables that cannot be bounded using the LP solver-based technique.

These range analysis techniques in F-SOFT have been applied to many examples, including a network protocol (PPP), an aircraft traffic alert system (TCAS), a mutual exclusion algorithm (Bakery), and an array manipulation example. For these control-intensive examples, we found that the LP-based range analysis technique reduced the number of state bits in the translated Boolean model by 60% on average. The bounded range analysis technique produced an additional 53% reduction on average. These resulted in considerable time savings in verification using both BDD-based and SAT-based methods.

2.3 Localized Predicate Abstraction and Register Sharing

Predicate abstraction has emerged as a popular technique for extracting finite-state models from software [1]. If all predicates are tracked globally in the program, the analysis often becomes intractable due to too many predicate relationships. Our contribution [7] is inspired by the lazy abstraction and localization techniques implemented in BLAST [5]. While BLAST makes use of interpolation, we use weakest pre-conditions along infeasible traces and the proof of unsatisfiability of a SAT solver to automatically find predicates relevant at each program location. Since most of the predicate relationships relevant at each program location are obtained from the refinement process itself, this significantly reduces the number of calls to back-end decision procedures in the abstraction computation.

The performance of BDD-based model checkers depends crucially on the number of state variables. Due to predicate localization most predicates are useful only in certain parts of the program. The state variables corresponding to these predicates can be *shared* to represent different predicates in other parts of the abstraction. However, maximal register sharing may result in too many abstraction refinement iterations; e.g., if the value of a certain predicate needs to be tracked at multiple program locations. We make use of a simple heuristic for deciding when to assign a *dedicated* state variable for a predicate in order to track it globally. While it is difficult to compare the performance of these techniques in F-SOFT with BLAST under controlled conditions, our experiments [7] indicated that the maximum number of active predicates at any program location are comparable for the two tools, even though BLAST uses a more complex refinement technique based on computation of Craig interpolants.

We have also applied our predicate abstraction techniques in F-SOFT to a large case study (about 32KLoC) consisting of a serial 16550-based RS-232 device driver from WINDDK 3790 for Windows-NT. We checked the correct lock usage property, i.e. lock acquires and releases should be alternating. Of the 93 related API functions, F-SOFT successfully proved the correctness of 72 functions in about 2 hours (no bugs found so far!).

3 Comparison to Other Tools

The most closely related tool to ours is CBMC [2], which also translates a `C` program into a Boolean representation to be analyzed by a back-end SAT-based

BMC. However, there are many differences. One major difference is that we generate a Boolean model of the software that can be analyzed by both bounded and unbounded model checking methods, using SAT solvers and BDDs. Another major difference in the software modeling is our block-based approach using a pc variable, rather than a statement-based approach in CBMC. (In our controlled experiments, the block-based approach provides a typical 25% performance improvement over a statement-based approach.) Additionally, the translation to a Boolean model in CBMC requires unwinding of loops up to some bound, a full inlining of functions, and cannot handle recursive functions. In contrast, our pc-based translation method does not require unwinding of loops, avoids multiple inlining, and can also handle bounded recursion. This allows our method to scale better than CBMC on larger programs, especially those with loops. The practical advantages over CBMC were demonstrated in a recent paper [6], where we also used specialized heuristics in the SAT solver to exploit the structure in software models.

We also differentiate our approach by use of light-weight pre-processing analyses such as program slicing and range analysis. Program slicing has been successfully used in other software model checkers [3, 11] as well. Although range analysis techniques have been used for other applications [10], to the best of our knowledge we are the first to use them for software model checking. In practice, it significantly reduces the number of bits needed to represent program variables in the translated Boolean model, compared to using a full bitwidth encoding, as in CBMC. Finally, F-Soft also allows abstraction of the software program using predicate abstraction and localization techniques. These are inspired by other model checkers [1, 5].

Acknowledgements. We thank Srihari Cadambi, Aleksandr Zaks and Himanshu Jain for their help in development of the F-Soft platform.

References

1. T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, 2001.
2. E.M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.
3. J. C. Corbett et al. Bandera: Extracting finite-state models from java source code. In *Int. Conf. on Software Engineering*, pages 439–448, 2000.
4. M.K. Ganai, A. Gupta, and P. Ashar. DiVer: SAT-based model checking platform for verifying large scale systems. In *TACAS*, volume 3340 of *LNCS*. Springer, 2005.
5. T.A. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM Press, 2004.
6. F. Ivančić, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. In *Symposium on Leveraging Formal Methods in Applications*, 2004.
7. H. Jain, F. Ivančić, A. Gupta, and M.K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS*, volume 3340 of *LNCS*. Springer, 2005.

8. 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.
9. G.C. Necula et al. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, volume 2304 of *LNCS*, pages 213–228. Springer-Verlag, 2002.
10. R. Rugina and M.C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI*, pages 182–195, 2000.
11. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), 2003.
12. A. Zaks, I. Shlyakhter, F. Ivančić, H. Cadambi, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Range analysis for software verification. 2005. In submission.