Model Checking Software with First Order Logic Specifications Using AIG Solvers

Mohammad A. Noureddine and Fadi A. Zaraket

Abstract—Static verification techniques leverage Boolean formula satisfiability solvers such as SAT and SMT solvers that operate on conjunctive normal form and first order logic formulae, respectively, to validate programs. They force bounds on variable ranges and execution time and translate the program and its specifications into a Boolean formula. They are limited to programs of relatively low complexity for the following reasons. (1) A small increase in the bounds can cause a large increase in the size of the translated formula. (2) Boolean satisfiability solvers are restricted to using optimizations that apply at the level of the formula. Finally, (3) the Boolean formulae often need to be regenerated with higher bounds to ensure the correctness of the translation. We present a method that uses And-Inverter-Graph (AIG) sequential circuits, and AIG synthesis and verification frameworks to validate programs. An AIG is a Boolean formula with memory elements, logically complete negated conjunction gates, and a hierarchical structure. Encoding the validation problem of a program as an AIG (1) typically provides a more succinct representation than a Boolean formulae encoding with no memory elements, (2) preserves the high-level structure of the program, and (3) enables the use of a number of powerful automated analysis techniques that have no counterparts for other Boolean formulae such as CNF. Our method takes an imperative program with a first order logic specification consisting of a precondition and a postcondition pair, and a bound on the program variable ranges, and produces an AIG with a designated output that is $true$ when the program violates the specification. Our method uses AIG synthesis reduction techniques to reduce the AIG, and then uses AIG verification techniques to check the satisfiability of the designated output. The results show that our method can validate designs that are not possible with other state of the art techniques, and with bounds that are an order of magnitude larger.

Index Terms—Software verification, static analysis, Boolean satisfiability solvers, Hoare triplet

1 Introduction

Recent advances in propositional and first order logic (FOL) satisfiability solvers that operate on conjunctive normal form (CNF) and satisfiability modulo theory (SMT) formulae, respectively, enabled static verification techniques [1], [2], [3], [4], [5] to check real programs. However, these programs often need to be partial, leaving out important functionality aspects, to enable the analysis to complete. Moreover, the analysis in some of these techniques is typically bound to relatively small limits [4], [5].

The work in [6] and [7] takes a declarative formula program ϕ in FOL with transitive closure and an imperative C program with assertion statements therein, respectively. They use a bound on the universe of discourse (input size) and translate the input program into a sequential circuit expressed in VHDL. A sequential circuit is a Boolean netlist with a hierarchical structure and memory elements referred to as *registers* where each register is associated with an *initial* and a *next state* value function. They pass the VHDL to SixthSense [8], an IBM internal sequential circuit verification framework, and decide the validity of ϕ or the assertions of the C program within the bound. They scale to bounds larger than what is possible with Kodkod [1] and CBMC [2] which translate the input program into a formula in CNF via unrolling quantifiers, loops and recursion, and then check the CNF validity using *CNF satisfiability* (SAT) solvers such as MiniSat [9].

In this work, we present our method that takes an imperative program $\mathcal{S}$ with a specification, FOL precondition and postcondition pair (P,Q), and checks whether $\mathcal{S}$ satisfies the specification within a bound b on the domain of the program and specification variables ($\mathcal{S} \models (P,Q)|_b$); i.e., when the bounded inputs of $\mathcal{S}$ satisfy P, the outputs of $\mathcal{S}$ satisfy Q. The program is written in J, a subset of C++/Java that includes integers, arrays, loops, and recursion. Our method translates the problem $\mathcal{S} \models (P,Q)|_b$ into and *And-Inverter-Graph* (AIG), a sequential circuit whose gates are restricted to NAND gates (logically complete negated conjunction), and a designated output therein that is $true$ iff the program violates the specification within the bounded domain. Our method uses AIG synthesis reduction and abstraction techniques embedded in the open source ABC [10] framework to reduce the generated AIG. Then it uses ABC verification techniques to decide the satisfiability of the designated output. ABC either (1) proves the validity of the program, (2) generates a counterexample illustrating that the program violates the specifications, or (3) reports an inconclusive result as it exhausts computational resources. Our method translates the counterexample back to the program domain and provides the user with a visual debugging tool (GTKWave [11]) to trace the violation.

- *M.A. Noureddine is with the Department of Computer Science, University of Illinois at Urbana Champaign, IL. E-mail: nouredd2@illinois.edu.*
- *F.A. Zaraket is with the Department of Electrical and Computer Engineering, American University of Beirut, Beirut, Lebanon. E-mail: zfadi@utexas.edu.*

Manuscript received 19 Aug. 2014; revised 20 Nov. 2015; accepted 2 Jan. 2016. Date of publication 20 Jan. 2016; date of current version 19 Aug. 2016. Recommended for acceptance by T. Bultan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2520468

0098-5589 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Our method significantly extends the work of both [6], [7] as follows.

- It uses a program counter semantics to translate the program into an intermediary *one loop program* ($\mathcal{O}lp$), where a program counter is an additional variable that encodes the control flow of the program including its conditionals, loops, and function calls. An $\mathcal{O}lp$ encodes the data flow into ternary conditional expressions based on the value of the program counter. Intuitively, the structure of an $\mathcal{O}lp$ is similar to that of an AIG where variables and loop body assignments resemble memory elements and next state functions, respectively. The translation preserves the structure of the program, e.g., its data and control flows, as both embody programming abstractions that can be useful for the reduction and verification techniques. The translation minimizes the need for additional memory elements that programmers do not envision as needed program elements and this allows programmers to better map the reported counter example to their original code. The work in [7] performs only a source to source translation to VHDL.
- It directly translates the $\mathcal{O}lp$ into bit level representation using AIG while [6], [7] depend on the VHDL compiler and synthesis tools to do the translation to bit level.
- It supports imperative programs annotated with FOL specifications and also supports array boundary and overflow checks.
- It supports function calls including recursion.
- In case the original correctness check was not conclusive, it uses heuristics to guess a termination bound on the program execution time, it enables a termination guarantee check within the execution time bound, and it then uses the execution time bound with bounded model checking to decide correctness.
- It uses ABC [10], an open source sequential circuit synthesis and verification framework, instead of SixthSense [8] an IBM internal sequential circuit solver. ABC is a transformation-based verification (TBV) [10] framework that operates on sequential circuits and iteratively and synergistically calls numerous reduction and abstraction algorithms such as retiming [12], redundancy removal [13], [14], [15], [16], logic rewriting [17], interpolation [18], and localization [19]. These algorithms simplify and decompose complex problems until they become tractable for ABC verification techniques such as symbolic model checking, bounded model checking, induction, interpolation, circuit SAT solving, and target enlargement [8], [20], [21], [22], [23], [24]. Some of these techniques such as induction [20] and interpolation [25] use SAT solvers at the backend, to compute partial intermediate results. They use the intermediate results to complete verification and do not necessarily unroll the AIG in the typical manner.
- Our method is fully implemented as an open source tool ($\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$) available online.[1]

1. http://research-fadi.aub.edu.lb/dkwk/doku.php?id=sa.

We evaluated our method with the verification of standard algorithms, fundamental and complex data structures, real applications and programs from the software verification benchmarks and compared our method to CBMC and other tools ranking top in the software verification competition [26]. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ succeeded to find and report counterexamples for all defected programs. It found and reported defects that we were not aware of while developing the evaluation benchmarks. It scaled to verification bounds higher than those possible with the other tools, and proved specifications that were not possible by the other tools.

1.1 Limitations of Translation to Boolean Formulae

There are three limiting aspects of translating high-level programs to Boolean formulae.

Disadvantage 1. The translation to Boolean formulae depends on the bounds; a small increase in the bound on variable ranges can cause a large increase in the size of the translated formula due to unwinding loop and recursion structures in programs, or eliminating quantifiers in declarative first order logic.

Disadvantage 2. CNF SAT solvers are restricted to using optimizations, such as symmetry breaking [27] and observability don't cares (ODC) [28], that apply at the level of CNF formulae. However these optimizations usually aim at increasing the speed of the solver and often result in larger formulae as they add literals and clauses to the CNF formula to encode symmetry and ODC optimizations [29]. Often times when the analyzer successfully generates a large CNF formula, the underlying solver requires intractable resources.

Disadvantage 3. Often times the formula needs to be regenerated with higher bounds in case the unwinding bounds were not large enough for the loops to complete as is the case with CBMC and ESBMC.

To extend the applicability of static analysis to a wider class of programs as well as to check more sophisticated specifications and gain more confidence in the results, we need to scale the analysis to significantly larger bounds.

1.2 Advantages of AIG

We formally define AIG sequential circuits in Section 3.3; for now an AIG can be viewed as a restricted C++ program, specifically a concurrent program in which all variables are either integers, whose range is statically bounded, or Boolean-valued, and dynamic allocation is forbidden [30]. There are two key advantages to compiling programs into AIG rather than Boolean formulae:

Advantage 1. AIG circuits are more expressive than CNF and SMT formulae as they are imperative and state-holding while CNF and SMT formulae are declarative and state-free. Consequently, AIG encodings of program semantics are more succinct than SMT or pure combinational SAT encodings. For example, they can naturally represent the computation of quantifiers and loops with no need to force iteration bounds and unroll them. Moreover, they can store and reuse intermediate results in local variables. In cases, SAT and SMT encoding algorithms represent programs with formulae that

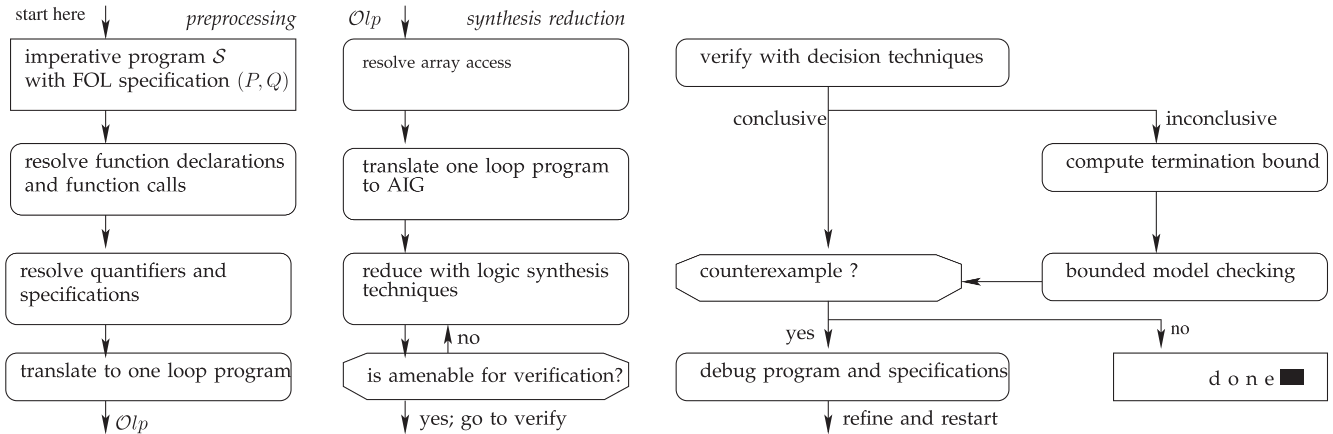

Fig. 1. Overview of verifying J programs with AIG synthesis and verification techniques.

are several orders of magnitude larger than the AIG representation. Appendix D, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2016.2520468, illustrates this as it compares the size of the AIG circuit produced by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and that of the CNF produced by CBMC.

Advantage 2. Casting the decision problem for a program specification as an invariant check on an AIG allows to leverage a number of powerful automated analysis techniques that we discuss in Section 3.4 and that have no counterparts in CNF or SMT analysis. For example, retiming [31] reduces the number of memory elements by moving sequential boundaries in a circuit, and reachability analysis [20] reasons about states and transitions, and those are not applicable for CNF and SMT formulae.

Other software verification techniques and tools exist that leverage predicate abstraction, interpolation, model checking, SMT, and other FOL and CNF solvers [5], [32], [33], [34], [35]. We discuss the tools and further compare our method to them in Section 7.

The rest of this paper is structured as follows. Section 2 provides an overview of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$, and illustrates the method with an array search example. Section 3 defines Boolean formulae, introduces $Jcore$ and J programs, $\mathcal{O}lp$, sequential and AIG circuits, and the ABC framework. Section 4 describes the translation of J programs into AIGs. Section 5 discusses the implementation. We discuss the results in Section 6, the related work in Section 7, and conclude with future work in Section 8. In the online supplemental material, available online, we provide a discussion of the automation of the method in Appendix A, available in the online supplemental material, a set of useful guidelines that help to select a successful and effective sequence of ABC techniques in Appendix B, available in the online supplemental material, additional results with memory comparison in Appendix D, available in the online supplemental material, and a sequential circuit example in Appendix E, available in the online supplemental material, that follows the example of Section 2.2.

2 Overview

Fig. 1 illustrates our method. First, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ preprocesses program $\mathcal{S}$ with its FOL specification pair (P, Q) given in J and transforms it into a program $\mathcal{S}'$ in $Jcore$; a subset of J that does not directly support function calls, specifications, and quantifiers. It then translates $\mathcal{S}'$ into a $\mathcal{O}lp$ where it encodes the control flow into an additional program counter variable and encodes the data flow into assignment statements with ternary conditional expressions that depend on the value of the program counter variable. Then $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ translates the generated $\mathcal{O}lp$ into an AIG circuit with bit vectors of fixed width that correspond to program variables. Array sizes are limited by the largest possible index and $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ translates each array element into a vector of AIG registers. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ resolves array access operations to refer to the vectors of registers, it then instantiates equivalent logic circuits to the expressions and connects the circuits to the initial value and next state value functions of the registers. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ designates one output of the AIG circuit to be $true$ when the program violates the specifications. Other AIG outputs signal out of bound array access and arithmetic overflow.

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ uses the ABC synthesis techniques to reduce the size of the generated AIG until it is amenable for verification. A verification engineer makes that decision in an interactive session. In automated sessions, the AIG is considered amenable for verification in case the number of registers was reduced to below a given threshold; e.g., 2^7, or several reduction techniques were called without achieving any significant reduction; e.g., a reduction of one register reduces the state space by half. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ then uses ABC verification techniques to check the designated output. The verification technique may find a violation and return a counterexample, return a proof that the program is correct within the bound, or return an inconclusive result after exhausting computational limits. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ translates the counterexample if any from the AIG level to traces of variable values in the original program and provides the user with a visual debugging view using GTKWave [11]. The user inspects and refines $\mathcal{S}$ and (P, Q) to find and fix the defect, and then restarts verification.

2.1 Termination Guarantee Bound

In case the ABC techniques do not reach a conclusive result, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ uses heuristics to compute a termination upper bound θ in terms of execution time, and calls ABC twice. The first call ($\Psi \equiv time >= \theta \rightarrow pc = last(\mathcal{S})$) is to verify

```
1  int ArraySearch(int[] a,int d,int s,int e,int n){
2   @pre as { 0<=s && s<=e && e<n && n<=MAXSIZE; }
3   int i = s;          // pc = pc+1
4   while ( i <= e) {   // pc = (i <= e) ? 5 : 10;
5    if (a[i] == d) {   // pc = (a[i] == d) ? 6 : 8;
6     break;}           // pc = 10;
7    else {
8     i = i+1;}         // pc = 4;
9   } //end while
10  return i;           // pc = 11; rv = i;
11 @post as {
12  (s <= rv && rv <= e) -> a[rv] == d
13  forall(int i)[s .. e]{
14    a[i] != d } -> (rv== −1) } }
```

```
1  @dotogether { // init-list for initial value0s
2   preas = 0 <= s && s <= e && e < n && n <= MAXSIZE;
3   pc = 0; notdone = true; postas = true; }
4  while (notdone) { @dotogether {
5   // next-list with next state functions
6   i = (pc == 3) ? s : (pc == 8) ? i+1 : i;
7   notdone = (pc == 11) ? false : true;
8   rv = (pc == 10) ? i : rv;
9   pc = (pc == 0) ? 3: (pc == 3) ? 4 :
10        (pc == 4) ? ( (i <= e) ? 5 : 10 ) :
11        (pc == 5) ? (a[i] == d) ? 6 : 8 ) :
12        (pc == 6) ? 10 : (pc == 8) ? 4  :
13        (pc == 10) ? 11 : pc;
14  postas = (rv >= s && rv <= e) -> a[rv] == d &&
15   forall(int i)[s .. e]{a[i] != d} -> (rv== −1); } }}
```

Fig. 2. Array search with rv defect in J and its equivalent $\mathcal{O}lp$ with the program counter variable.

that the program is guaranteed to terminate within θ. The second call uses θ as a bound with the ABC bounded model checking (bmc) technique to prove that the program does not violate the specification within θ. Intuitively, checking $\mathcal{S} \models (P, \Psi)|_b$ is often easier than checking $\mathcal{S} \models (P, Q)|_b$.

To compute θ, we simulate the program with several inputs of size b and keep track of the maximum number of loop iterations for each loop max_{loop} and the maximum recursive depth for each recursive call max_{rec} across runs. We set $\theta = \ell * max_{loop} + r * max_{rec}$ where ℓ and r are the numbers of loops and recursive calls in the program, respectively.

2.2 Array Search Example

The Array search program in Fig. 2 takes as input an array a, a start index s, an end index e, a data value d, and the number of elements in the array n. The precondition states that s and e are within array bounds and that n is within the bound of array sizes. The postcondition makes use of a special predefined function variable rv denoting the return value of the function. It states that if rv is valid between s and e inclusive, then $a[rv]$ must be equal to d, otherwise, rv must be invalid (-1) and all entries in a between s and e inclusive are not equal to d. Fig. 2 also shows a semantically equivalent $\mathcal{O}lp$ array search program in terms of the values of the common program variables at termination.

The equivalent program introduces Boolean variables preas, postas, and notdone to encode the precondition, postcondition, and termination state of the program, respectively. The equivalent program also introduces a program counter variable pc which encodes the control flow of the program as indicated in the comments of the original program with no need of loop iteration bounds or unrolling.

Variable notdone is initialized to *true*, and pc is initialized to the first executable line of the program 3. Once pc reaches the last executable line of the program 13, the program terminates and thus notdone becomes *false*. Assignment statements are grouped by target variables, and encoded into ternary conditional expressions that depend on the value of pc. For example, the iterator i is assigned to s when pc is 3, incremented when pc is 8, and remains the same otherwise. Furthermore, the assignment statements on Lines 2 and 3 assign initial values to the target variables. The assignment statements inside the while loop (Lines 5 to 15) compute the next state value of each of the target variables.

Our method translates the $\mathcal{O}lp$ to an AIG sequential circuit where an iteration of the while loop is equivalent to a single time step in the AIG. For example, variable pc ranges from 0 to 11 and is encoded with a 4 registers bit-vector. The corresponding initial value functions are set to 0 and the next state functions are set to gates $G(pc_i)$ representing the right hand side expression of the pc assignment statement. Section 4.4 discusses the translation from $\mathcal{O}lp$ to AIG and Appendix E, available in the online supplemental material, illustrates an example AIG circuit implementing variable i.

Our method takes the resulting AIG and designates a gate therein that represents $\neg(preas \wedge done \rightarrow postas)$ as the output gate, passes the AIG to ABC, and checks for validity. The ABC solver returns the counterexample of Fig. 14 with $a = [15\ 15\ 15\ 11], s = 3, e = 3, n = 4, d = 13, rv = 0$ where d is not in a, and the return value is $e + 1$, while the postcondition requires an invalid index (-1). The provided counter example can be used to fix the program. A possible fix is to replace Line 6 with `return i;`, and Line 10 with `return -1;`. Our method takes the fixed program, translates it into an AIG and completes a proof of correctness with an ABC symbolic model checking technique.

2.3 Recursive Array Search Example

Fig. 3 illustrates a recursive version of the array search example given in J and its translation to *Jcore*. Only the variable s that is essential to the recursion is declared as an argument to recArraySearch and the other variables a, d, e, n, and r are declared as global for readability of the *Jcore* translation. Line 14 of the J program makes the initial call to recArraySearch and is translated starting at line 44 of the *Jcore* program where execution starts. Since the function is recursive, its arguments (s) and local variables (ret) are translated into arrays where each entry in the array represents a stack frame. The stack pointer sp points to the current stack frame and is initialized to -1.

The translation of the function call of line 14 marshals the input argument into the next frame of s pointed to by

```
1  int [] a; int d, e, n, r;
2  // recursive function declaration
3  int recArraySearch(int s){
4   int ret = -1;
5   if (s > e) {
6     ret = -1;
7   } else if ( a[s] == d) {
8     ret = s;
9   } else { // recursive call
10    ret = recArraySearch(s+1);
11  }
12  return ret;} // rv = ret;
13 // initial recursive function call
14 r = recArraySearch(0);
```

```
20 int [] a; int d; int n;
21 int [MAXDEPTH] s, ret, //turn recursive
      variables into arrays
22   return-pc,// array to save where recursive
        calls return
23   retvar;// array to save recursive return
        values
24 int sp = -1; // recursion stack pointer
25 int fcall-retvar-1, fcall-retvar-2; //
      replace recursive function calls
```

```
26 L1:ret[sp] = -1;//replace local variable v occurrences by v[sp]
27    if (s[sp]) > e) {
28     ret[sp] = -1;
29    } else if (a[s[sp]] == d) {
30     ret[sp] = s[sp];
31    } else {// replace the recursive function call
32     s[sp+1] = s[sp] + 1;//marshal s+1 into next frame (sp+1)
33     return-pc[sp+1] = L2;//set the return pc for the next frame
34     sp = sp + 1; // increment the stack pointer
35     pc = L1;// goto L1
36     //when the recursion call returns here through line 23,
37 L2: fcall-retvar-1 = retvar[sp];//save the return value
38     sp = sp -1;//decrement the stack pointer
39     ret[sp] = fcall-retvar;//substitute result for call
40    }
41    retvar[sp] = ret[sp];//store return expression in stack frame
42    pc = return-pc[sp];//goto callee location
43 START:// entry point: original call recArraySearch(0);
44 L3:s[sp+1] = 0; // marshal input variable
45    return-pc[sp+1] = L4; // set where function should return
46    sp = sp +1; // increment stack pointer
47    pc = L1; // goto L1
48    //when the function call returns here through line 23,
49 L4:fcall-retvar-2 = retvar[sp];
50    sp = sp -1;
51    r = fcall-retvar-2;//store result in r
```

Fig. 3. J recursive array search example (top left) translated into its equivalent in $Jcore$: variable declarations (bottom left) and code (right).

$(sp+1)$. Execution should return to location $L4$ when done with the code corresponding to the recursive function call. The translation stores $L4$ in array $return-pc$; an array of return program counter locations to preserve control. The stack pointer is incremented and execution is transferred to the code corresponding to the function body by explicitly setting pc to code location $L1$. Once execution returns to location $L4$ (through setting pc on line 42), the result of the computation $fcall-retvar-2$ replaces the function call in the assignment statement of line 14 and the result gets consequently stored in r.

Location $L1$ corresponds to line 4; the first line of the function. Each argument and local variable v in the function body gets replaced by its array version $v[sp]$ indexed with the stack pointer. Global variables remain intact. The translation replaces the recursive call of line 10 by the code on lines 32-39. Similar to the translation of line 14, the argument $s[sp]+1$ is marshaled into $s[sp+1]$, location $L2$ is saved into $return-pc$, sp is incremented, and execution is transferred to the beginning of the function body by setting pc to $L1$. Once execution returns to $L2$, the result of the computation $retvar[sp]$ is saved into $fcall-retvar-1$, the stack pointer is decremented, and $fcall-retvar-1$ replaces the function call in the assignment statement of line 10.

The translation saves the value of the expression of the return statement of line 12 in the current frame of $retvar$ ($retvar[sp=0]$), and transfers control to the callee location by explicitly setting pc to the location stored in $return-pc[sp]$.

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ takes the $Jcore$ translation of the recursive array search, translates it to an $\mathcal{O}lp$ in a manner similar to the regular array search. Then it translates the $\mathcal{O}lp$ into an AIG.

3 Background and Definitions

In this section we introduce J, $Jcore$, $\mathcal{O}lp$, sequential and AIG circuits, and the ABC framework.

3.1 J Programs

The grammar on the left of Fig. 4 defines $Jcore$, the core subset of J. A $Jcore$ program is one or more declaration statements, followed by one or more statements. The directives `a-un-op`, `a-bin-op`, `b-un-op`, `b-bin-op`, and `ba-bin-op` denote arithmetic unary and binary operators, Boolean unary and binary operators, and Boolean arithmetic operators, respectively. The variables matching the `var` and `array-var` rules in a program $\mathcal{S}$ form the sets of scalar variables $V=\{v_1,v_2,\ldots,v_m\}$ and array variables $A=\{a_1,a_2,\ldots,a_n\}$, respectively.

Definition 1 (Terms). *A* target term *is either a variable $v\in V$, an array access term of the form $a[e_1]$ or $a[e_1][e_2]$ which denotes the e_1^{th} element of a or the $(c_2e_1+e_2)^{th}$ elements of a where c_2 is a constant, $a\in A$ and e_1 and e_2 are expressions. A* term *is either a target term, or a constant $c\in\mathbb{Z}$.* `base`$(a[e])$ *returns a,* `index`$(a[e])$ *returns e, and* `index`$(a[e_1][e_2])$ *returns $e_1c_1+c_2$.*

Definition 2 (Expressions). *An* expression *is a term, an arithmetic expression of the form $-e, e_1+e_2, e_1-e_2, e_1*e_2, e_1/e_2, e_1\%e_2$ where e, e_1, e_2 are expressions and $-,+,*,/,$ and $\%$ denote subtraction, addition, multiplication, division and remainder, respectively.*

Definition 3 (Boolean expressions). *A* Boolean expression *is either (1) a constant $c\in\mathbb{B}=\{true, false\}$, (2) arithmetic of the form $e_1\circ e_2$ where e_1,e_2 are expressions and $\circ\in\{<,\leqslant,>,\geqslant,==\}$ which denote smaller, less than or equal, bigger than,*

```
program: declaration-statement+ statement+
statement: assignment | conditional | loop | break ;
list-of-statements: statement*

// statements
assignment: target = expression ;
conditional: if (boolean-expr) { then-block } else { else-block }
loop: while ( boolean-expr ) { while-block }
then-block, else-block, while-block: list-of-statements

// declarations
declaration-statement: declaration ;
declaration: (variable-decl | array-decl)
variable-decl: type var
array-decl: type array-var [ constant? ] ([ constant ])?
var, array-var : id
type: int | bool

// expressions
boolean-expr: term | b-un-op boolean-expr | boolean-expr b-bin-op
    boolean-expr | expression ba-bin-op expression
term: constant | target-term
target-term: var | array-access
array-access: array-var [ expression ] | array-var [ expression ]
    [ expression ]
expression: term | a-un-op expression | expression a-bin-op
    expression | boolean-expr ? expression : expression
```

```
// statement extended with pre/post conditions
statement: assignment | conditional | loop | pre-condition
    | post-condition
// specification pre and postcondition
pre-condition: @pre specname { boolean-expr }
post-condition: @post specname { boolean-expr }

// boolean expression extended with quantifiers
boolean-expr: term | b-un-op boolean-expr | boolean-expr
    b-bin-op boolean-expr | expression ba-bin-op expression
    | quantifier
quantifier: (forall|exists) ( range ) { boolean-expr }
range: var [ expression .. expression ]

// declaration statement extended with function declaration
declaration-statement: declaration ; | function-decl
function-decl: type fname ( arg-decl-list? ) {
    declaration-statement* body-block }
arg-decl-list: variable-decl ( , variable-decl)*
body-block: list-of-statements return-statement
return-statement: return expression;

// terms extended with function calls
term: constant | target-term | function-call
function-call: fname ( arg-call-list? )
arg-call-list: expression ( , expression )*
specname, fname : id
```

Fig. 4. Grammar of *Jcore* (left), the core subset of the *J* imperative language (right).

bigger than or equal, and equal, respectively, (3) a unary Boolean of the form $\neg b$*, or (4) a Boolean of the form* $b \circ b'$ *where* $\neg$ *denotes negation,* b, b' *are Boolean expressions and* $\circ \in \{\&\&, \|, \rightarrow, ==\}$ *which denote logical conjunction, disjunction, implication, and equivalence, respectively.*

Definition 4 (First order logic formula). *A* first order formula *is either a Boolean expression, or a quantified formula of the form* $Qq.b(q)$*, where* $Q \in \{\forall, \exists\}$ *is a universal or an existential quantifier,* q *is a quantified variable, and* $b(q)$ *is a first order formula with* q *as a free variable.*

Definition 5 (Statement). *A* statement *is one of the following.*

a. *a declaration statement of the form type* v;*, type* $a[c_1]$;*, or type* $a[c_1][c_2]$ *where type is either* `bool` *or* `int` *denoting the domain of the variable values,* v *and* a *denote the variables in sets* V *and* A*, and* c_1 *and* c_2 *are constants denoting the size of array* a. `typeof(`v`)` *and* `typeof(`a`)` *denote type.*
b. *an* assignment *statement* g *of the form* $t = e$ *where* t *is a target term and* e *is an expression.* `target(`g`)` *and* `expr(`g`)` *denote* t *and* e*, respectively.*
c. *a* list of statements $\ell = \langle s_1; s_2; \ldots; s_{|\ell|}\rangle$ *that could be empty. The term* $\ell.s_i, 1 \leq i \leq |\ell|$ *denotes the* ith *statement in* ℓ*,* `first(`ℓ`)` *and* `last(`ℓ`)` *denote* $\ell.s_1$ *and* $\ell.s_{|\ell|}$*, respectively, and* `llist(`s`)` *denotes the largest list of statements* ℓ *containing* s *such that* $s = \ell.s_i$*,* $1 \leq i \leq |\ell|$.
d. *a* conditional *statement* d *of the form* $if(b_1)\{then-block\}$ $else$ $\{else-block\}$ *where* b_1 *is a Boolean expression, and* $then-block$ *and* $else-block$ *are lists of statements.* `condition(`d`)` *denotes* b_1.
e. *a loop statement* w *of the form* $while(b_2)\{while-block\}$ *where* b_2 *is a Boolean expression and* $while-block$ *is a list of statements.* `condition(`w`)` *and* `loop(`s`)` *denote* b_2 *and the innermost loop statement containing* s *if any, otherwise, it returns* nil*, respectively.*

The grammar of Fig. 4 (right) extends *Jcore* to form *J*. Boolean expressions are extended with existential and universal quantifiers that quantify a Boolean expression over a range of variable values defined by a start and an end expression. Parentheses are allowed around terms, expressions, and Boolean expressions and are left our from the grammar for clarity. Statements are extended with pre- and postconditions that share the same specification name (`specname`). Declaration statements are extended with function declarations that take an argument declaration list, and a body block including a return statement. Terms are extended with function calls with an argument call list.

3.1.1 *J* Program Semantics

The semantics of a program is defined in terms of traces of variable values across execution time. The assignment statements define the data flow of the program and their semantics is defined in terms of updating the value of the target term with the value of the expression of the assignment statement. The lists of statements, conditional statements, loop statements, function calls, and return statements define the control flow of the program. The program `declaration-statement+ statement+` starts execution at its entry point which is the first statement in `statement+`. The list of statements defines the order of execution. An

```
primary-input: primary declaration;

decl-list: primary-input*
    declaration-statement+
init-list: assignment+
next-list: assignment+

one-loop-program: decl-list
    @dotogether {
      init-list }
    while (notdone) { @dotogether {
      next-list } }
```

Fig. 5. $\mathcal{O}lp$ grammar.

if/else conditional statement s in a list of statements ℓ executes the `then-block` if the value of the Boolean expression b is $true$, otherwise it executes the `else-block`. The last statement in the `then-block` and the `else-block` moves execution to the statement next to s in ℓ. A loop statement s in a list of statements ℓ executes the first statement of the `while-block` if the value of the Boolean expression is $true$, otherwise it executes the statement next to s in ℓ. The last statement in the `while-block` moves execution back to the loop statement s.

A function call updates the values of the function argument declaration variables with the corresponding function call argument expressions and moves execution to the first statement of the `body-block`. The return statement in a function declaration substitutes the corresponding function call with the value of the return expression and moves execution back to the statement that made the function call.

A specification (P, Q) is a pair of first order formulae where P is the precondition specifying constraints on the inputs of $\mathcal{S}$ and Q is a postcondition relating the outputs of $\mathcal{S}$ to its inputs.

3.2 One Loop Programs

The grammar in Fig. 5 extends $Jcore$ with the $primary$ and $dotogether$ constructs to define one loop programs. The `primary` construct denotes a primary input variable declaration with non-deterministic values. The concurrent $dotogether\{together\text{-}block\}$ denotes that all statements of the $together\text{-}block$ list of statements execute simultaneously. An $\mathcal{O}lp$ starts with variable declarations. Then the `init-list` list of assignment statements concurrently initializes the $\mathcal{O}lp$ variables. After initialization, a loop keeps updating the values of the variables concurrently using the list of assignment statements `next-list` until the $\mathcal{O}lp$ is done as denoted by the $notdone$ Boolean variable. Intuitively, the structure of an $\mathcal{O}lp$ is similar to that of a sequential circuit where the variables correspond to registers, the `init-list` and `next-list` correspond to the initial and next state value functions, respectively.

3.3 AIG Sequential Circuit

Definition 6 (Sequential circuit). *A* sequential circuit *is a tuple $((U, E), G, O)$. Subset O of U is specified as the* primary outputs. *The pair (U, E) is a directed graph on vertices U and edges $E \subseteq U \times U$. The function $G : U \mapsto types$ maps vertices to types. A type can be* primary input, bit-register *(which we refer to as* register*), and one of the logical* gates *such as two input conjunction and disjunction. Vertices mapped to primary inputs by G are denoted by I and have no predecessors in E. Vertices mapped to registers by G are denoted by R and each has an* initial value, *and a* next-state function *predecessor in E. For a vertex v mapped to a logical gate, $G(v)$ denotes the logical function of the gate applied to the predecessors of v in E.*

Definition 7 (Fanins). *The direct* fanins *of a gate u is defined as $\{v \,|\, (v, u) \in E\}$, i.e., the set of source vertices connected to u in E. The* support *of u is given by $\{v \,|\, (v \in I \vee v \in R) \wedge (v, u) \in *E\}$ that is all source vertices in R or I that are connected to u with $*E$, the transitive closure of E.*

For a sequential circuit to be syntactically well-formed, vertices in I should have no fanins, vertices in R should have two fanins (the next-state function and the initial-value function of that register), and every cycle in the sequential circuit should contain at least one vertex from R. The initial-value functions of R shall have no registers in their support. In the following, we consider only well-formed sequential circuits.

The ABC analyzer reasons about AIG which are sequential circuits with only NAND gates restricted to have 2 fanins. Since NAND is functionally complete, this is not a limitation.

3.3.1 Semantics of Sequential Circuits

Definition 8 (State). *A* state *$\sigma : R \mapsto \mathbb{B}$ is a Boolean valuation to vertices in R.*

Definition 9 (Trace). *Given a sequence of input valuations $\rho : I \times \mathbb{N} \mapsto \mathbb{B}$, and an initial state σ_0, a* trace *is a mapping $t : U \times \mathbb{N} \mapsto \mathbb{B}$ that gives a value to vertices in U across synchronized time* steps *denoted as indexes from $\mathbb{N}$. The mapping must be consistent with E and G in the following sense. Term u_j denotes the source vertex of the jth incoming edge to v, implying that $(u_j, v) \in E$. The term $t(v, i)$ denotes the value of gate v at time i in trace t.*

$$t(v,i) = \begin{cases} \rho(v,i) & : v \in I \\ \sigma_0(v) & : v \in R, i = 0, u_1 := \sigma(v,0) \text{ initial-state of} v \\ t(u_2, i-1) & : v \in R, i > 0, u_2 := \text{next-state of } v \\ G(v)\big(t(u_1,i), & \\ \quad \dots, t(u_n,i)\big) & : v \text{ is a combinational gate with function } G(v). \end{cases}$$

We will refer to the transition from one valuation to the next as a *step*. The timing model is synchronous since all registers update values simultaneously.

A node in the AIG is said to be satisfiable if there is an input sequence which, when applied to an initial state, will result in that node taking value $true$. A node in the circuit is valid if its negation is not satisfiable. We will refer to targets and invariants in the circuit; these are vertices in the circuit whose satisfiability and validity is of interest, respectively. A sequential circuit can naturally be associated with a finite state machine (FSM), which is a graph on the states. However, the circuit is different from its FSM; among other differences, it is exponentially more succinct in almost all cases of interest [36].

TABLE 1
Brief Description of Selected ABC Synthesis and Verification Techniques

Technique	Synthesis technique description	Command
Balancing [10]	Logic balancing applies associativity transformation to reduce AIG levels.	`balance`
Sweep	Structural register sweep (SRS) reduces the number of registers in the circuit by eliminating stuck-at-constant registers [37].	`ssweep`
Correspondence	Signal correspondence (Scorr) computes a set of classes of sequentially-equivalent nodes using k-step induction [37].	`scl -l`
Rewriting	AIG rewriting iteratively selects and replaces rooted subgraphs with smaller pre-computed subgraphs in order to reduce AIG size [38].	`rewrite`
Refactoring	Refactoring is a variation of rewriting. It uses a heuristic to compute a large cut for selected AIG nodes, then replaces the sub-graph that corresponds to the cut with a refactored structure if an improvement is observed [39].	`refactor`
Retiming	Retiming manipulates register boundaries and count in a given logic network, while maintaining output functionality and logic structure [31].	`retime`
Equivalence	Computes correspondence between circuit nodes based on partitioning and speculation and performs simple and $k-$step induction to compute sequential equivalence checking [37]	`dsec`
Induction	Temporal induction uses circuit SAT and BDD solvers to carry simple and k-step induction proofs over the time steps of the AIG [40].	`ind`
Interpolation	Interpolation-based algorithms find interpolants and over-approximate the reachable states of the AIG with respect to the property [25].	`int`
Reachability	Property directed reachability (Pdr) tries to prove that there is no transition from an initial state of the AIG to a bad state [20].	`pdr`
	BDD-based reachability computes a symbolic representation of the reachable state space and employs fixed point analysis [21].	`dprove`

3.4 ABC Synthesis and Verification Framework

ABC is an open source synthesis and verification framework for sequential circuits. ABC operates on sequential circuits in AIG format and checks the satisfiability of a designated output gate therein. ABC applies several reduction and abstraction techniques and leverages the synergy between them to simplify and decompose the problem into smaller problems. It then calls decision techniques to decide the simplified problems. The current version of ABC houses more than 400 commands under several categories such as abstraction (11), verification (39), liveness (5), combinational synthesis (27), sequential synthesis (26), basic (14), disjoint support decomposition (DSD) manager (7), various (61), and pre-release ABC9 (104) commands. It also has commands to read and write several sequential circuit input formats, map AIG circuits into technology libraries, and print the AIG circuits and its representation for debugging and visualization. Table 1 briefly summarizes selected synthesis reduction techniques that we often used in $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ such as balancing, structural register sweeping, signal correspondence, logic rewriting, refactoring, and retiming. The table also discusses the sequential equivalence checking, induction, interpolation and reachability verification techniques. Appendix A, available in the online supplemental material, details few other techniques and illustrates in pseudocode an automated implementation of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ that selects ABC techniques for reduction and verification.

Some of the verification techniques such as BDD based reachability and sequential equivalence checking perform verification without necessarily using SAT solvers. Other techniques use solvers including SAT at the back end. However, they require a transition system represented in AIG, and they do not necessarily unroll the transition system. For example, property directed reachability [20] is an efficient implementation of "SAT-Based model checking without unrolling" [41]. It issues several queries to the SAT solver to test whether various formulas are 1-step inductive. Those queries are usually trivial compared to those made using unrolling based techniques. If the technique converges, it eventually computes lemmas that are 1-step inductive strengthening of the property.

Interpolation [18] enables a purely SAT-based method of *unbounded* model checking. It performs several SAT queries, and computes interpolants from the refuted SAT queries that over-approximate the reachable state space. Unlike typical SAT-based model checking techniques that require unrolling the transition system up to the length of the longest simple path between two states, interpolation "does not require unfolding beyond the diameter of the state space, and in practice often succeeds with shorter unfoldings" [18].

Both induction and interpolation techniques might reduce the problem into a smaller problem even if they do not succeed in solving it.

4 Transformation

In this section we present source to source transformation algorithms to translate a program $\mathcal{S}$ written in $Jcore$ into an $\mathcal{O}lp$. We then present algorithms that preprocess a J program with functions and specifications and translate it into $Jcore$. Finally, we present algorithms for translating the generated $\mathcal{O}lp$ into an AIG circuit. The algorithms follow the execution semantics of J and the correctness of the translation holds by construction. We first present helper functions and terms that are used in the algorithms.

4.1 Helper Functions

Definition 10 (Labels). *Function* label *maps each statement* s *from* $\mathcal{S}$ *to a unique number* `label(s)` $\in \mathbb{N}$ *such that* $\forall s_1, s_2 \in \mathcal{S}. s_1 \neq s_2 \rightarrow$ `label(`s_1`)`$\neq$`label(`s_2`)`.

```
generateOneLoopProgram(S)
 generateDeclarationList(S)
 generareInitList(S)
 // translate data flow into next-list
 foreach variable v in V ∪ A
  if v ∈ V
   v-next-list = generateVarNextList(v,S)
  else // v ∈ A
   v-next-list = generateArrayNextList(v,S)
  endif
  append v-next-list to next-list
 endfor
 // translate control flow into next-list
 pc-next-list = generatePCNextList(S)
 append pc-next-list to next-list
 append notdone = !(pc == done); to next-list
```

```
generateVarNextList(v,S)
 foreach statement s with target(s) = v
  append (pc ==label(s)) ? expression(s) : to
      v-expr
 endfor
 //otherwise v stays the same
 append v; to v-expr
 let v-next-list be v = v-expr ;
```

```
generateDeclarationList(S)
 foreach variable v in V ∪ A
  append typeof(v) v; to decl-list
  // handle variables that are not assigned
  if v is used before it is assigned
   append primary typeof(v) vnondet; to decl-list
  endif
 endfor
 append int pc;
        bool notdone; to decl-list
```

```
generateInitList(S)
 foreach variable v in V ∪ A
  //variables used before assignied are
      nondeterministic
  if v is used before it is assigned
   append v = vnondet; to init-list
  else
   append v = 0; to init-list
  endif
 endfor

 // initialize program counter
 append pc =label(first(S));
        notdone = true; to init-list
```

Fig. 6. Algorithms for generating the $\mathcal{O}lp$ `decl-list`, `init-list`, and `next-list` from $\mathcal{S}$.

Definition 11 (Next). *Function* next *takes a statement s from $\mathcal{S}$ and returns the label of its successor* `label` *(s_{i+1}) in $\ell =$* `list` *$(s) = \langle s_1, s_2, \ldots, s_{|\ell|}\rangle$ where $s = s_i, 1 \leq i < |\ell|$ except for when s is a* `break` *statement and* `loop` *$(s) \neq nil$, then* `next` *(s) returns* `label(loop(s))`. *When $i = \ell$; i.e., s is the last statement of ℓ, several cases arise.*

- a. *If s is a* `return` *statement,* `next` *(s) will never be called since s will be replaced by two assignment statements when $\mathcal{S}$ is translated to $\mathcal{O}lp$ (Fig. 8). For completeness,* `next` *(s) returns* `label` *(s).*
- b. *If ℓ is the* `while-block` *of a loop $l =$* `loop` *(s),* `next` *(s) returns* `label` *(l).*
- c. *If ℓ is the* `body-block` *of a function declaration* `next` *(s) returns the label of the* `return` *statement.*
- d. *If ℓ is a* `then-block` *or an* `else-block` *of a conditional c,* `next` *(s) returns* `next` *(c).*
- e. *Finally, if ℓ is the last statement in $\mathcal{S}$,* `next` *(s) returns a special label done.*

Functions `then` and `else` take a conditional statement d of the form `if (b) {then-block} else {else-block}`. They return the labels of the first statement of the then and else blocks, `label(first(then-block))`, and `label(first(else-block))`, respectively. If the list of statements is empty, both functions return `next(d)`.

Function body takes a loop statement or a function declaration. The loop statement is of the form `while (b) {while-block}`. The function declaration is of the form `type fname(arg-decl-list) {body-block}` where `type`, `fname`, and `arg-decl-list` are the return type, name, and argument declaration list of the function, respectively, and `body-block` is a list of statements followed by a return statement. For a function declaration, body returns the label of the first statement in `body-block` (`label (first(body-block))`). For a loop, body returns the label of the first statement in `while-block`. In case `while-block` is empty, it returns the label of the loop.

4.2 *Jcore* Programs to $\mathcal{O}lp$

Algorithm `generateOneLoopProgram` of Fig. 6 takes a program $\mathcal{S}$ written in *Jcore* and generates the `decl-list`, `init-list`, and `next-list` of an equivalent $\mathcal{O}lp$. Algorithm `generateDeclarationList` generates a declaration for each variable in $\mathcal{S}$, and also generates a primary input `vnondet` variable for each variable that is used before being initialized in $\mathcal{S}$. It also declares the program counter pc and the termination guard $notdone$ variables. Algorithm `generateOneLoopProgram` calls `generateInitList` that generates `init-list` where all variables are initialized to 0 except those that are used before being assigned in $\mathcal{S}$, those are initialized with their corresponding non-deterministic primary input variables. The program counter pc and the termination guard $notdone$ are initialized to the label of the first statement of the program, and to $true$, respectively.

The `generateOneLoopProgram` Algorithm builds one assignment statement `v-next-list` for each variable v and appends it to `next-list`. If v is a regular variable, `generateVarNextList` iterates over each assignment s where v is the target and builds a nested ternary conditional expression `v-expr` that evaluates to `expr(s)` when pc points to the label of s. When pc does not point to a statement that assigns to v, `v-expr` evaluates to the original value of v which is denoted by appending v as the value of the last choice in in the nested ternary expression.

If v is an array variable, Algorithm `generateArrayNextList` from Fig. 7 iterates over all statements s where v is target. It aggregates all expressions and index expressions into two nested ternary conditional expressions `a-index` and `a-expr` for the index expression and the right hand side expression of the `v-next-list` assignment statement. The ternary conditional expressions depend on the position of pc to return the corresponding expressions and index expressions. In case pc does not point to a statement with v as a target, then `v-next-list` makes sure that no array element gets modified by appending 0 and $v[0]$ as the last default choices in the ternary conditional expressions

```
generateArrayNextList(v,S)
 //iterate over all statements assigning to v
 foreach assignment s with base(target(s)) = v
  //s is of the form v[ie] = e
  let ie be index(target(s))
  let e be expr(s)
  //aggregate array access in one expression
  append (pc ==label(s)) ? ie : to a-index
  append (pc ==label(s)) ? e : to a-expr
 endfor
 //when no array access happens, v[0] does not
     change
 append 0 to a-index
 append v[0] to a-expr
 //construct one assignment statement for the
     array
 let v-next-list be v[a-index] =a-expr;
```

```
generatePCNextList(S)
 // encode control flow into pc
 let pc-next-list be pc =
 // iterate over all statements
 foreach s in S
  if s is a conditional //handle conditional
   append (pc ==label(s)) ?
           (condition(s) ? then(s) : else(s)) :
         to pc-next-list
  elseif s is a loop    // handle loops
   append (pc ==label(s)) ?
           (condition(s) ? body(s) : next(s)) :
         to pc-next-list
  elseif s is an assignment with target(s) = pc
   // handle direct assignment to pc
   // which result from function call resolution
   append (pc ==label(s)) ? expr(s): to pc-next-list
  else // handle other statements (assignments)
   append (pc ==label(s)) ? next(s): to pc-next-list
  endif
 endfor
 // when done, program counter does not change
 append pc; to pc-next-list
```

Fig. 7. Algorithm to aggregate array variable assignments (top) and encode control flow into program counter pc (bottom).

a-index and a-expr, respectively. Effectively, this corresponds to the statement $v[0] = v[0]$ which is enough at the $\mathcal{O}lp$ level to preserve semantics. Algorithms resolveArrayAccess and resolveArrayTargetTerms of Fig. 13 make sure that only the targeted array entry changes value, and the other array entries explicitly retain their values in the AIG.

Finally, generateOneLoopProgram calls generatePCNextList from Fig. 7 to encode the control flow of $\mathcal{S}$ into pc-next-list with a nested ternary conditional expression that defines the value of pc. Algorithm generatePCNextList iterates over all statements and encodes their execution flow semantics. When s is at a conditional statement, pc moves to then-block if the Boolean condition evaluates to $true$, and to the else-block otherwise. When s is at a loop statement, then pc moves to the first statement of body-block if the Boolean condition of s evaluates to $true$, and to the statement next to the loop otherwise. This encodes the semantics of a loop with no need of bounds and unrolling. When s is an assignment statement such that target$(s) = pc$, then pc moves to the expression of s; this is similar to a goto statement and may result from resolving function calls as discussed in Section 4.3.1. For other statements, pc points to the next statement.

4.3 Preprocessing J Programs

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ translates a program $\mathcal{S}$ written in J to a program $\mathcal{S}'$ in $Jcore$ where function declarations, return statements, functions calls, pre and post conditions, and quantifiers are not directly supported.

4.3.1 Non Recursive Functions

Algorithm resolveNonRecursiveFunction in Fig. 8 considers the declaration and function calls of a function $fname$. It declares two additional variables in the function. Variable $return\text{-}pc$ is added to arg-decl-list and it holds the label that pc should return to after the function is done. Variable $retvar$ is added to func-decl-list and it holds the value of the return expression when the function is done. A specification such as the postcondition of Fig. 2 can use $retvar$ or rv for short to denote the return value. The func-return-statement is then replaced by func-return-list that sets $retvar$ and pc to expr(func-return-statement) and $return\text{-}pc$, respectively.

For each statement s containing an $fname$ function call, resolveNonRecursiveFunction constructs a list of assignment statements that marshal the arguments and set the $return\text{-}pc$ to the label of fcall-ret-statement. It then adds an explicit statement that sets pc to the label of the first statement in $fname$. The fcall-ret-statement reads the value of $retvar$ into a unique function call return variable $fcall\text{-}retvar\text{-}i$. This is necessary to resolve multiple function calls in the same expression. Statement s' substitutes the function call by $fcall\text{-}retvar\text{-}i$. The argument marshaling, fcall-ret-statement, and s' form the list func-call-list. Finally, statement s is replaced by the list of statements func-call-list.

4.3.2 Recursive Functions

Algorithm resolveRecursiveFunctionDeclaration of Fig. 9 resolves a recursive function declaration $fname$ into $Jcore$ by emulating stack frames with depth $max\text{-}depth$. The argument and local variables of $fname$ become arrays. References to them in the body of the function are replaced with the corresponding array variables indexed by an additional stack pointer $sp\text{-}i$ that points to the current recursive depth of the function. The rest of the translation follows similarly to Algorithm resolveNonRecursiveFunction of Fig. 8 and declares additional variables to hold the return, and return program counter values.

Algorithm resolveRecursiveFunctionCalls of Fig. 10 works similarly to resolving function calls in Fig. 8 with additional attention to the stack pointer $sp\text{-}i$. Variable $sp\text{-}i$ is used to appropriately marshal the arguments and the return program counter to the current frame of the recursive function. Note that $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ checks for out of bound array access and in case $max\text{-}depth$ was not enough for the program to terminate, then the stack out of bound would be fired similar to an out of stack limit warning on regular operation systems. In this case, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ should be called with a higher $max\text{-}depth$.

4.3.3 Specifications and Quantifiers

Algorithm resolveQuantifier of Fig. 11 takes a quantifier expression q and the statement s that contains it and performs quantifier instantiation. It declares a Boolean variable

```
resolveNonRecursiveFunction(fname,S)
 let S be of the form decl-list statement-list
 let type fname(arg-decl-list) {
       func-decl-list func-body func-return-statement }
   be the non-recursive function declaration of fname

 // add a return program counter variable to function argument
 // declarations to hold the label where the function should return
 append , int return-pc to arg-decl-list
 // add a return variable to the function variable declaration
 append type retvar; to func-decl-list
 // turn the return statement into an assignment statement
 append retvar = expression(func-return-statement); to func-return-list
 // when function done, move program counter back to the caller
 append pc = return-pc; to func-return-list
 replace func-return-statement with func-return-list
 // iterate over all statements with calls to fname
 foreach statement s containing an fname(arg-call-list) function call
  // marshal the argument list
  foreach argument expression e in arg-call-list
   let a be the corresponding variable to e in arg-decl-list
   append a = e; to func-call-list
  endfor
  // save the label of the statement the function should return to
  // and transfer control to the function
  append return-pc = label(fcall-ret-statement);
         pc = label(first(func-body)); to func-call-list
  // declare a unique function call return variable
  let i be a unique number
  append type fcall-retvar-i; to decl-list
  // create a statement that reads the return value from the function
  let fcall-ret-statement be fcall-retvar-i = retvar;
  // place the return point from the function
  append fcall-ret-statement to func-call-list
  // fix the statement to use the returned value
  s' = substitute fname(arg-call-list) by fcall-retvar-i in s
  append  s' to func-call-list

  replace s with func-call-list
 endfor
```

Fig. 8. Resolve non-recursive function calls and declarations.

q-i in the scope of the quantifier to hold the value of q. It initializes q-i to $true$ when q is universally quantified, and to $false$ when q is existentially quantified. It translates the quantifier statement to a loop that iterates over the range of the quantified variable v and accumulates the value of the Boolean expression with a conjunction in case q was a universal quantifier, and a disjunction in case q was an existential quantifier. Algorithm `resolveQuantifier` substitutes for q in s by q-i to construct s'. Finally, it replaces s with the constructed loop and s'. Alternatively, if the Boolean expression in the quantifier does not include function calls, the quantifier can be unrolled into a sequence of conjunctions (disjunctions in case of an existential quantifier) of the Boolean expression for the range of v. This allows for a smaller execution time.

Algorithm `resolvePrePost` of Fig. 11 declares a Boolean variable `pre-specname` for each precondition statement s of the form `@pre specname {boolean-expr}`. It then replaces the precondition statement with an assignment statement that updates `pre-specname` with the value of the corresponding Boolean expression `boolean-expr`. Algorithm `resolvePrePost` works similarly for the postcondition. Of special interest is the expression $pre{-}specname \wedge pc = label(s) \rightarrow post{-}specname$ which expresses that the program satisfies the specification.

4.4 $\mathcal{O}lp$ to AIG Circuits

Algorithm `variables` of Fig. 12 instantiates vectors of AIG registers and primary inputs to translate the corresponding $\mathcal{O}lp$ variables and stores the translation in a lookup function `vargates`. The width of a bit vector can be selected by the user, or set to match the default width of the declared type. Typically the default values for the bit width are 32 bits for an integer. Arrays are represented by a fixed number of array elements and each element is then treated as a regular variable. The number of array elements is either specified in the program, bounded by the user, or fixed to a constant by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$.

4.4.1 Assignment Statements

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ considers each assignment statement s in `init-list` and `next-list` and traverses the right hand side expression of s with the recursive `traverse` Algorithm of Fig. 12. If expression exp refers to a variable (base case), or an array access operator with a constant index, then `traverse` returns `vargates(`exp`)`. If exp was an array access $a[ie]$ where a is the array variable and ie is an index expression, then `resolveArrayAccess` of Fig. 13 translates exp into a nested ternary expression where the conditions depend on the value of ie and the values are array access terms with constant indices between 0 and $size(a) - 1$. Algorithm

```
resolveRecursiveFunctionDeclaration(fname,S)
 let S be of the form decl-list statement-list
 let type fname(arg-decl-list) {
       func-decl-list func-body func-return-statement }
   be the recursive function declaration of fname
 //increase the dimensionality of arguments
 foreach argument declaration arg-decl in arg-decl-list
  // type var becomes type var[max-depth] and
  // type array-var[c] becomes type array-var[c][max-depth]
  append arg-decl[max-depth], to arg-decl-list-sp
 endfor
 // add a return-pc variable to the argument list
 append int return-pc[max-depth] to arg-decl-list-sp
 replace arg-decl-list with arg-decl-list-sp
 // similarly, increase dimensionality for local variables
 foreach declaration decl in func-decl-list
  append decl[max-depth]; to decl-list-sp
 endfor
 // add return variables
 append int retvar[max-depth]; to decl-list-sp
 replace func-decl-list with decl-list-sp

 // variable  sp-i plays a stack pointer role for fname
 let i be a unique number
 append int sp-i; to decl-list

 // starts at negative 1
 // once incremented it is at the first recursive frame
 prepend sp-i = -1; to statement-list

 // replace variables with frame variables
 foreach variable v in arg-decl-list and func-decl-list
  replace all occurrences v with v[sp-i] in func-body
  replace all occurrences v with v[sp-i] in func-return-statement
 endfor
 // similarly for array variables
 foreach array variable a in arg-decl-list and func-decl-list
  replace all occurrences a[c1] with a[c1][sp-i] in func-body
  replace all occurrences a[c1] with a[c1][sp-i] in func-return-statement
 endfor
 // fix the return statement to return to the callee
 append retvar[sp-i] = expression(func-return-statement);
        pc = return-pc[sp-i]; to func-return-list
 replace func-return-statement with func-return-list
```

Fig. 9. Resolve recursive function declarations.

`resolveArrayAccess` calls `traverse` again on the generated expression to produce the corresponding AIG.

If the expression is a logical, conditional, or arithmetic expression, then the `library` routine looks it up in a complete table of AIG circuits with the adequate bit width. For example, if the expression is a ternary conditional statement of the form $b\ ?\ e_1 : e_2$, then `library` instantiates a multiplexer, connects its two data fanins to the nodes corresponding to e_1 and e_2, connects its control fanins to the nodes corresponding to b, and returns its fanouts. Alternatively, users have the choice to abstract multiplication, division, and remainder by non-constant factors using a command line option. The abstraction happens by replacing the abstracted expression with fresh non-deterministic primary input variables.

4.4.2 Connections and Array Access Target Terms

For assignment statements with target terms that are regular variables, or array access with constant index expressions, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ connects the nodes corresponding to the right hand side expressions in `init-list` and `next-list` to the initial and next state value fanins of the corresponding register gates, respectively.

An assignment statement s of the form $a[ie] = expr$;, where a is an array variable, ie is a non constant index expression, and $expr$ is an expression, requires preprocessing before being translated to AIG. Algorithm `variables` instantiates $size(a)$ register vectors corresponding to the elements of array variable a, each requiring initial and next state value functions. Algorithm `resolveArrayTargetTerms` of Fig. 13 takes as input an assignment statement s and replaces it with $size(a)$ assignment statements of the form $a[j] = (j == ie)?expr : a[j]$; where j ranges from 0 to $size(a) - 1$ such that each array element $a[j]$ evaluates to $expr$ if ie evaluates to j, otherwise, $a[j]$ keeps its value.

5 Implementation

We fully implemented $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ using C++ and ANTLR [42] and integrated that with the ABC synthesis and verification framework [10] and the GTKWave visualization tool [11].

Table 2 provides a short list of the $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ commands. The frontend supports the J syntax using the `read` command. The `start-prove` command specifies (1) bounds on variable data width, array size, and recursion depth, and

```
resolveRecursiveFunctionCalls(fname, S)
 let S be of the form decl-list statement-list
 let type fname(arg-decl-list) {
       func-decl-list func-body func-return-statement }
   be the recursive function declaration of fname
 let sp be the stack pointer corresponding to fname in decl-list

 foreach statement s containing an fname(arg-call-list) function call
  // declare a unique function call return variable
  let i be a unique number
  append type fcall-retvar-i; to decl-list
  // create a statement that reads the return value from the function
  append fcall-retvar-i = retvar[sp]; to fcall-ret-statement
  // marshal the argument list
  foreach argument expression e in arg-call-list
   let a be the corresponding variable to e in arg-decl-list
   append a[sp+1] = e; to func-call-list
  endfor
  // save the label of the statement the function should return to
  append return-pc[sp+1] = label(fcall-ret-statement); to func-call-list
  // increment the stack pointer
  append sp = sp+1 to func-call-list
  // transfer control to the function
  append pc = label(first(func-body)); to func-call-list
  // place the return point from the function
  append fcall-ret-statement to func-call-list
  // decrement the stack pointer
  append sp = sp-1 to func-call-list
  // fix the statement to use the returned value
  s' = substitute fname(arg-call-list) by fcall-retvar-i in s
  append  s' to func-call-list

  replace s with func-call-list
 endfor
```

Fig. 10. Resolve recursive function calls.

(2) optional automatic checks on array out of bound access and arithmetic overflow. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ provides a fully automated verification path of execution where it executes a well selected sequence of synthesis reduction and verification techniques that work well for software verification tasks as discussed in Appendix A, available in the online supplemental material. Command `prove-scheme1` is a short predefined sequence of ABC commands that balances the AIG, performs signal correspondence in order to reduce the number of registers, and verifies the resulting AIG with `dprove` and its default parameters. Command `prove-scheme2` differs in that it performs sequential sweeping with the circuit rewriting option turned on, and tries induction with a time limit of 60 seconds before calling `dprove`.

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ also supports an interactive shell where the user can save and use intermediary results, try different synthesis and verification strategies using the `abc` commands directly, and inspect counterexamples using the `debug` command. Command `vdebug` maps the counterexample to program variable values and invokes GTKWave to help the user debug the program. Fig. 14 shows a trace that highlights the defect of the array search program of Fig. 2 where $d \notin a$ and $rv \neq -1$. Commands `start-sim` and `sim-variable` provide program based simulation utilities that allow users to perform *whatif* analysis on values of program variables. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ is available online as an open source tool with tutorial and documentation.[2]

2. http://research-fadi.aub.edu.lb/dkwk/doku.php?id=sa.

6 Results

We evaluated $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ by verifying software artifacts including library algorithms and array based implementations of data structures with complete sophisticated specifications from the Calculus of Computation book [43], an array based implementation of the red black balanced binary search tree (RBBBST) with specifications formalized from [44], and a memory allocation model (MMAN) with complete specifications provided from the Frama-C ANSI C Specification Language (ACSL) [45]. Since *Jcore* does not support pointers directly, allocation of data structure nodes is supported by a memory allocation module based on array indirection. Intuitively, pointers are positions in arrays that represent memory and a bit-vector keeps their consistency. Table 3 reports the results with several bounds (equal to the number of elements in the array plus one) and compares against CBMC [2]. The results show that $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ scales to bounds larger than CBMC by at least one order of magnitude.

We also used verification task benchmarks from the competition on software verification (SVComp) [26] to evaluate the effectiveness and efficiency of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ in practice and as compared to state of the art verification tools. Table 4 reports $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ results compared to leading SVComp verification tools on verification tasks that most tools completed successfully. Table 5 reports results on benchmarks where $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ successfully completed the verification tasks and the leading tools either timed out or produced erroneous output.

```
resolveQuantifier(q, s, S) // statement s contains quantifier q
 let q be of the form Q (v [e1 ... e2]) (boolean-expr)
 let scope-decl-list and scope-statement-list be the declaration
     and statement lists of the scope of s, respectively
 let i be a unique number // create a unique Boolean variable
 append bool q-i; to scope-decl-list
 append v = e1; to q-list // initialize the quantified variable
 if Q is a universal quantifier
  prepend q-i = true; to scope-statement-list // initialize q-i
  // translate the quantifier into a loop
  append while(v ≤ e2 ∧ q-i){q-i = q-i∧ boolean-expr; v = v + 1;} to q-list
 else // Q is an existential quantifier
  prepend q-i = false; to scope-statement-list
  append while(v ≤ e2 ∧ ¬q-i){q-i = q-i∨ boolean-expr; v = v + 1;} to q-list
 endif
 s' = substitute q by q-i in s // substitute back into the code
 append s' to q-list
 replace s with q-list
```

```
resolvePrePost(S)
 //declare a Boolean variable for each pre/
     post condition and replace the
     statement by an assignment to the
     declared variable
 foreach precondition statement s of the
     form (@pre specname{boolean-expr})
  append bool pre-specname; to decl-list
  replace s with pre-specname =boolean-expr;
 endfor
 foreach postcondition statement s of the
     form (@post specname{boolean-expr})
  append bool post-specname; to decl-list
  replace s with post-specname =boolean-expr;
 endfor
```

Fig. 11. Resolve quantifiers (top) and pre- and post condition statements (bottom).

6.1 Algorithms and Data Structures

Table 3 shows that $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ was able to verify the correctness of programs with respect to sophisticated specifications faster than CBMC and for a number of elements (array size) that CBMC could not verify. The table shows the number of memory elements (registers), gates, and logic levels before and after the ABC reductions. It also shows the time taken to verify the program after reductions and the total verification time including parsing and reductions. The CBMC columns show the size of the generated CNF formulae in terms of the number of CNF variables and clauses. We developed the programs and the specifications and called $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC to verify their correctness incrementally. Table D3 in Appendix D, available in the online supplemental material, further illustrates that the difference in memory usage between $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC is consistent with the difference between AIG size (register and gate count) and CNF size (variable and clause count). We repeated the same experiments with Java PathFinder (JPF) [33] and its symbolic execution engine (JPFSE) [46] to further validate and evaluate our work. The JPF and JPFSE experiments are detailed in Appendix C, available in the online supplemental material, including runtime results in Tables C1 and C2.

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$, CBMC, JPF, and JPFSE returned counterexamples that we used to correct the programs and the specifications. For programs with array indirection where array elements are used as indexes of other arrays such as `next`, and `left`, `right`, and `parent` inf the linked list and red black balanced binary search tree, all tools succeeded in returning counterexamples for index out of bound violations. However, once we corrected those issues, CBMC mistakenly reported that the programs are correct while $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$, JPF, and JPFSE correctly reported counterexamples related to the cardinality of the data structures after insertion or removal of elements. The rows with MISTAKE in them report the time taken by CBMC to verify the defected code and return a wrong result versus the time taken by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ to verify the corrected code.

Note that most of the verification techniques we used with ABC are unbounded verification techniques such as `pdr`, `dprove`, and `ind` discussed in Table 1. CBMC is a bounded model checker that uses bounds for unwinding loops and recursion and that translates the resulting formulae to CNF SAT. The comparison between $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC is valid since (1) the ABC verification techniques address an AIG produced by forcing a bound on input domains within a program, and (2) CBMC similarly forces a bound on input domains, and includes the ability to inject loop (recursion) unwinding assertions that fire in case the unwinding bound was not enough for the loop (recursion) to terminate. In case the SAT solver returns a counterexample with such an assertion, CBMC increases the unwinding bounds and tries again.

```
traverse(exp)
 // base case of recursion
 if (exp is a variable)
  return vargates(exp)
 elseif (exp is constant array access)
  return vargates(exp)
 elseif (exp is array access)
  return resolveArrayAccess(exp)
 endif
 // traverse operands and store
     results in wirevec
 for i from 1 to exp.operands.size()
  wirevec[i] = traverse(exp.operands[i])
 endfor
 //lookup AIG circuit for operation in
      library
 return library(exp.operation, wirevec)
```

```
variables(decl-list)
 foreach variable v in decl-list
  if v is an array variable
   //handle array variables
   for j from 0 to size(v) - 1
    vargates(v[j]) =
      instantiate-registers(v[j],type(v[j]))
   endfor
  elseif (v is a primary input)
   // handle nondeterministic variables
   vargates(v) =
     instantiate-primary-inputs(v,type(v))
  else
   vargates(v) =
     instantiate-registers(v,type(v))
  endif
 endfor
```

Fig. 12. Expression traversal and register instantiations for variables.

```
resolveArrayAccess(exp)
  let a be base(exp)
  let ie be index(exp)
  let N be size(a) - 1

  for j from 0 to N-1
   append (ie == j) ? a[j]  : to
       a-expr-resolve
  endfor
  append a[N] to a-expr-resolve
  return traverse(exp)
```

```
resolveArrayTargetTerms(S)
  foreach statement s where target(s) is array-access
   let a be base(array-access)
   let ie be index(array-access)
   let N be size (a) - 1

   for j from 0 to N
    append a[j] = (j == ie) ? expr  :  a[j]; to a-resolve
   endfor
   replace s with a-resolve
  endfor
```

Fig. 13. Resolve array access expressions and resolve array target terms.

TABLE 2
Summary of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ Commands

Command	Description
`read`	Parses a program and generates a parse graph.
`prove-scheme1`	Performs the following sequence of synthesis and proof commands `balance;zero;scorr;dprove`.
`prove-scheme2`	Performs `balance;zero;ssweep -r;ind -T 60;dprove`.
`abc cmd`	Performs the ABC techniques specified in `cmd`.
`debug`	Generates a program counterexample from the AIG counterexample and allows the user to inspect the values in a textual format.
`vdebug`	A visual version of `debug` that allows the user to view and interact with the traces using GTKWave.
`start-sim`	Starts the simulator tool to help analyze the code and perform what-if analysis on variable values.
`sim-variable`	Runs the simulator helper tool to generate and display values for specific variables.
`start-prove`	Configures the prove engine with parameters such as variable bit width, array size bounds, and recursion depth bounds, as well as overflow check, out of bound access check, and quantifier unrolling.

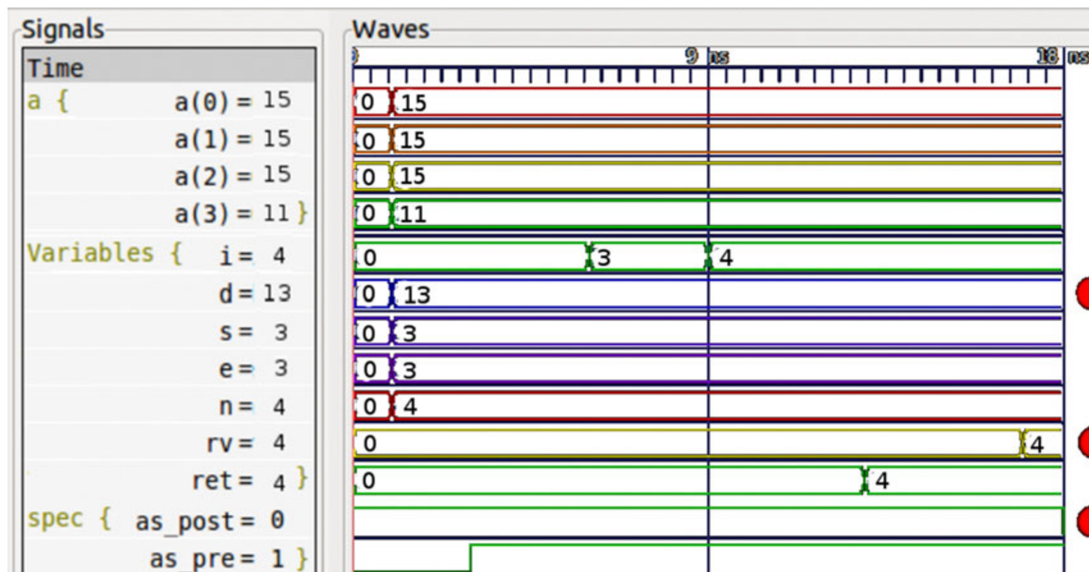

Fig. 14. Array search visual debugging with GTKWave [11].

It stops when it exhausts memory or timing resources. That is, if the SAT solver used by CBMC with the unwinding assertion option returned a proof result, the proof guarantees that the program terminates within the unwinding bound.

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ performed better than CBMC as follows.

- Most of the times where both $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC succeeded to verify the programs, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ took less time to complete the verification task.
- $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ succeeded to verify the programs for bounds larger than what CBMC could verify within the three hours timeout.
- The AIG register count is several orders of magnitude smaller than the number of CNF variables. The same applies to the number of AIG gates as compared to CNF clauses.
- CBMC failed to generate the CNF formula for some programs with relatively larger structures while $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ succeeded to verify some of them and generated AIG circuits but timed out while verifying the rest. This is significant since model checking *typically* returns counterexamples fast when they exist, and takes long time to report a proof for unsatisfiable (correct) claims. That is,

TABLE 3
Results of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC to Verify Array Based Algorithms and Data Structures with Increasing Array Sizes

	Elements	AIG size before/after reductions			$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$		CNF size		CBMC
		registers	and	level	verif.	total (s)	vars	clauses	time (s)
array search	3	86/41	719/313	24/15	0.33	4.36	2,416	6,784	0.016
	7	118/68	1,064/568	27/19	3.89	12.4	4,612	15,008	722.4
	15	174/119	1,781/1,116	30/21	2.41	16.87	9,112	34,496	TO
	31	286/226	3,362/2,346	45/24	1.43	33.67	18,332	84,928	TO
	63	526/461	6,895/5,100	78/26	4.57	99.64	37,216	230,208	TO
	127	1,054/984	14,780/11,315	143/28	21.32	397	75,876	695,616	TO
	255	4,798/4,718	70,742/55,364	529/33	682.1	8,022.1	TO	TO	TO
binary search	3	94/56	879/555	30/19	0.11	1.04	6,503	24,533	0.085
	7	151/83	1,832/850	42/17	0.54	1.47	16,172	68,130	1.91
	15	268/143	5,185/1,943	62/20	25.42	27.69	42,461	197,223	38.493
	31	491/262	18,355/4,903	68/28	112.84	115.13	100,379	412,410	4,955.223
	63	874/454	64,457/8,608	102/28	1,132	1,152.22	2,660,677	13,689,632	TO
	127	1,911/976	277,662/19,907	170/32	TO	TO	TO	TO	TO
bubble sort	3	114/44	1,198/393	29/16	0.29	5.79	15,534	43,332	0.174
	7	169/68	2,218/885	35/20	17.1	31.09	63,785	197,603	7.298
	15	276/117	5,607/2,106	47/22	1,390.25	1,426.98	341,552	1,082,480	455.5
	31	603/360	40,448/25,546	75/36	2,668.22	2,669.82	2,144,801	6,809,622	TO
	63	1,293/814	174,535/110,123	140/40	4,323.51	4,384.08	TO	TO	TO
	127	2,847/1,844	770,398/485,523	269/44	TO	TO	T0	TO	TO
selection sort	3	86/45	1,021 /679	26/19	20.62	20.69	17,941	55,151	0.37
	7	150/87	2,726/1,971	34/24	1,105.2	1,105.74	315,505	1,129,784	TO
	15	280/125	5,676/2,236	47/22	2,280	2,280.18	959,841	3,849,448	TO
	31	590/383	39,543/26,547	75/27	TO	TO	3,269,941	14,305,256	TO
	63	1,278/841	172,385/112,601	140/31	TO	TO	TO	TO	TO
array partition	3	110/57	1,896/689	33/19	0.87	2.79	36,636	121,478	2.43
	7	147/87	2,509/1,174	34/24	93.47	97.56	622,799	2,663,342	TO
	15	206/ 138	3,819/2,651	38/35	2,127	2,135.7	2,061,314	9,390,063	TO
	31	323/248	6,779/5,557	42/39	TO	TO	7,315,089	34,755,647	TO
	63	568/486	13,451/12,178	46/43	TO	TO	TO	TO	TO
linked list insert	3	237/98	4,310/871	38/19	109.63	118.89	28,347	93,711	0.194 (MISTAKE)
	7	344/179	6,117/1,693	41/26	1,800	1,811	83,079	299,415	0.527 (MISTAKE)
	15	503/345	10,521/3,763	47/26	TO	TO	281,759	1,080,852	2.027 (MISTAKE)
	31	839/634	19,624/10,060	53/36	TO	TO	1,039,655	4,149,044	13.575 (MISTAKE)
	63	1,559/1,482	40,375/16,396	59/32	ERROR	ERROR	3,946,601	16,177,278	152.92 (MISTAKE)
linked list remove	3	197/84	2,906/722	33/21	47.72	71.383	22,898	78,336	0.163 (MISTAKE)
	7	293/157	4,454/1,387	39/25	1,800.15	1,830.21	59,219	227,798	0.457 (MISTAKE)
	15	448/397	7,847/3,094	45/20	TO	TO	175,176	748,820	1.935 (MISTAKE)
	31	778/821	15,763/6,364	51/23	TO	TO	579,957	2,678,266	11.163 (MISTAKE)
	63	1,492/1,584	233,791/14,346	57/25	TO	TO	2,038,358	9,958,256	85.94 (MISTAKE)
RBBBST insert	3	779/407	14,284/6,451	122/41	313.11	314.22	2,765,445	10,145,891	7.169 (MISTAKE)
	7	1,555/1,077	40,794/29,281	128/54	3,600.48	4,087.1	5,602,125	20,473,165	15.456 (MISTAKE)
	15	1,834/1,221	50,540/28,928	134/43	TO	TO	10,487,436	38,445,954	TO
	31	2,810/2,124	100,841/66,558	152/49	TO	TO	21,680,107	21,680,107	TO
	63	4,789/4,034	213,576/147,577	186/56	TO	TO	MEMOUT	MEMOUT	MEMOUT
RBBBST remove	3	1,487/923	34,506/15,309	251/44	124.63	313.51	4,403,782	4,403,782	8.57 (MISTAKE)
	7	1,559/993	43,608/18,963	253/41	3,600.62	4,154.93	8,587,721	8,587,721	17.972 (MISTAKE)
	15	2,072/1,404	69,646/48,341	254/56	TO	TO	17,476,396	17,476,396	44.67 (MISTAKE)
	31	2,969/2,308	139,461/101,039	259/53	TO	TO	37,383,755	37,383,755	327.682 (MISTAKE)
	63	4,610/3,901	291,311/233,674	262/59	TO	TO	MEMOUT	MEMOUT	MEMOUT
memory manager	16	3,033/962	41,137/13,461	366/31	460.09	608.61	TO	TO	TO
	64	8,799/2,873	99,510/52,543	1,072/36	1,759.52	2,238.18	TO	TO	TO
average reductions	64.5%	51.2%	44.7%						

TO, MO, ERROR, and MISTAKE stand for timeout, memory out, runtime error and verification mistake, respectively.

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ succeeded to perform model checking for a significant time length, even when it timed out, as compared to CBMC that failed to generate the CNF formula, thus did not really test the program for correctness.

When JPF and JPFSE terminate the exploration of the explicit state space of a given benchmark looking for defects, generating *high path coverage* test cases and performing model checking, they may report a "no errors found" result. This result (1) is not a guarantee of correctness (see

TABLE 4
Runtime Results in Seconds for Verification Tasks Completed Successfully by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ Compared to the Leading Software Verification Competition Tools

	Category	$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$	BLAST	CBMC	CPAChecker	ESBMC	rank
test_locks_5_true	ControlFlowInt	0.15	23	1.6	2.1	0.43	1
test_locks_6_true	ControlFlowInt	0.23	44	2.4	2.7	0.42	1
test_locks_7_true	ControlFlowInt	0.31	140	2.8	5	0.45	1
test_locks_8_true	ControlFlowInt	0.37	280	3.6	21	0.48	1
test_locks_9_true	ControlFlowInt	0.55	610	4.5	53	0.52	2
test_locks_10_true	ControlFlowInt	0.84	TO	5.6	53	0.54	2
test_locks_11_true	ControlFlowInt	1.24	TO	7	53	0.6	2
test_locks_12_true	ControlFlowInt	0.99	TO	8.7	53	0.62	2
test_locks_13_true	ControlFlowInt	1.14	TO	11	52	0.66	2
test_locks_14_true	ControlFlowInt	1.56	TO	14	52	0.68	2
test_locks_15_true	ControlFlowInt	1.83	TO	15	53	0.77	2
test_locks_14_false	ControlFlowInt	0.46	0.14	0.22	2	1.4	3
test_locks_15_false	ControlFlowInt	0.51	0.13	0.21	2	1.5	3
Problem01_00_true	eca	7.5	TO	TO-T	4.5	4.6	3
Problem01_10_true	eca	8.59	730	TO-T	4.5	4.2	3
Problem01_30_true	eca	9.06	TO	TO-T	4.6	4	3
Problem01_40_true	eca	9.41	TO	TO-T	4.5	5.9	3
Problem02_00_true	eca	10.68	460	TO-T	4.3	6.3	3
Problem02_10_true	eca	9.71	580	TO-T	4.3	4.8	3
Problem02_20_true	eca	10.14	ERR	TO-T	4.3	4.8	3
Problem01_20_false	eca	9.35	TO	150	5.3	33	2
Problem01_50_false	eca	9.87	TO	22	4.8	33	2
Problem01_60_false	eca	5.38	780	0.56	2.8	30	3
s3_clnt_1_true	ssh-simplified	57.3	140	88	3.5	3.4	3
s3_clnt_2_true	ssh-simplified	47.1	TO	90	3.6	4.2	3
s3_clnt_3_true	ssh-simplified	77.64	350	110	3.7	3.9	3
s3_clnt_4_true	ssh-simplified	79.45	TO	90	3.5	3.6	3
s3_srvr_1a_true	ssh-simplified	20.11	1.2	3.4	1.8	0.69	5
s3_srvr_1b_true	ssh-simplified	1.68	1.2	1.3	1.6	0.48	5
s3_srvr_4_true	ssh-simplified	799	440	210	14	3.8	5
s3_srvr_6_true	ssh-simplified	694	TO	230	24	4.3	4
s3_srvr_2_true	ssh-simplified	567.51	410	200	4.4	3.9	5
s3_srvr_8_true	ssh-simplified	340	220	220	4.7	4.1	5
s3_clnt_1_false	ssh-simplified	47.15	23	2.1	2.8	13	5
s3_clnt_2_false	ssh-simplified	44.26	23	2.1	2.8	14	5
s3_clnt_3_false	ssh-simplified	27.56	23	2.5	2.8	17	5
s3_clnt_4_false	ssh-simplified	76.95	23	2.1	2.8	14	5
s3_srvr_1_false	ssh-simplified	35.47	4.2	3	2.4	22	5
s3_srvr_2_false	ssh-simplified	37.05	4.4	2.8	2.4	18	5
s3_srvr_6_false	ssh-simplified	14.24	0.15	0.21	19	21	3

TO and ERR stand for timeout and error, respectively.

Section 7) and (2) is qualitatively different than the "no error exists within bound" result reported by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC. That said, the JPF and JPFSE results of Appendix C, available in the online supplemental material, shown in Tables C1 and C2 (1) reinforce the correctness of the benchmarks built with $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ as both JPF and JPFSE exhausted resources without finding a counterexample in the benchmarks, and (2) provide evidence of the utility and effectiveness of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ to handle large bounds and sophisticated benchmarks where JPF and JPFSE exhausted resources without producing a "no errors found" favorable result.

For the array search program, the precondition checks that the indexes are in range and the postcondition checks whether the return value is consistent. For the binary search program both the precondition and the postcondition specify the sortedness property for the array. For the bubble and selection sort programs, the precondition checks the sanity of the size of the array, and the postcondition specifies that the resulting array holds the same elements of the original array in order. For the array partition program, the precondition specifies the sanity of the indexes and the size of the array, and the postcondition specifies that the resulting array holds the same elements of the original array partitioned in order to the left and right of the cursor. The linked list insert and remove precondition specifies a non circular, in order, list of elements, and the postconditions reassert the same specification with additional cardinality conditions. The RBBBST insert and remove precondition specifies the full characteristics of an RBBBST and that the index arrays involved all hold valid indexes. The postconditions reassert the same with additional cardinality conditions. The RBBBST insert also checks the sanity of the newly allocated node in the tree. The memory manager unit has several multidimensional arrays to book-keep the use of memory chucks and the precondition and the postcondition assert the sanity of the arrays and the consistency of the

TABLE 5
Results for Verification Tasks in Seconds Where $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ Terminated Successfully for the First 14 Tasks and Timed Out for the Last Five, while All the Leading Tools Timed Out or Produced Erroneous Output for the First Seven Tasks, Three Out of Four Tools Failed on the Next Four, and at Least One of the Tools Failed for the Next Three

	Category	$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$	BLAST	CBMC	CPAChecker	ESBMC
Problem07_00_true	ControlFlowInt/eca	620	TO	TO-T	TO	UNKNOWN-TO
Problem09_00_true	ControlFlowInt/eca	433	TO	UNKNOWN-TO	TO	UNKNOWN-190
insertion_sort_true	loops	154	MISTAKE-6.6	TO-T	TO	UNKNOWN-TO
Ackermann01_true	Recursive	723	N/A	TO-T	ERR-1.9	MISTAKE-TO
McCarthy91_true	Recursive	567	N/A	TO-T	ERR-1.9	MISTAKE-TO
Addition03_false	Recursive	11.2	N/A	MISTAKE-TO	ERR-2	TO_F
MultCommutative_true	Recursive	780	N/A	TO-T	ERR-1.9	MISTAKE-TO
jain_5_true	bitvector	1.44	N/A	0.53	TO	0.25
Ackermann02_false	Recursive	22	N/A	0.73	ERR-1.9	TO-F
McCarthy91_false	Recursive	1.2	N/A	0.19	ERR-1.9	TO-F
EvenOdd01_true	Recursive	0.68	N/A	0.96	ERR-1.9	MISTAKE-0.4
linear_search_false	loops	0.39	0.13	0.62	ERR-2.6	MISTAKE-0.3
EvenOdd03_false	Recursive	0.45	N/A	0.2	ERR-1.9	0.4
invert_string_false	loops	0.22	1.8	0.47	TO	0.6
Problem08_00_true	ControlFlowInt/eca	TO	TO	UNKNOWN-TO	TO	UNKNOWN-TO
s3_srvr_1_alt_true.BV	bitvector	TO	N/A	TO-T	TO	TO-T
s3_srvr_1_true	ssh-simplified	TO	UNKNOWN-0.1	160	4.6	3.9
s3_srvr_7_true	ssh-simplified	TO	TO	210	24	4.3
s3_srvr_3_true	ssh-simplified	TO	430	210	13	4

TO: timeout, TO-T: report true on timeout, TO-F report false on timeout, MISTAKE: report erroneous results, ERR: reported a runtime error, and UNKNOWN: report inconclusive result.

module. We made use of the `__CPROVER_assume` statement and loops to implement the preconditions and the quantifiers in CBMC. Similarly, we used the `Verify` API and the `jpf` configuration files to configure JPF and JPFSE.

The reduction algorithms used by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ were able to reduce the original problems on average to 64.5 percent of their memory elements, and to 51.2, and 44.7 percent of their logic gate and level counts in often insignificant time. Without these reductions $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ timed out on several benchmarks. These reductions have no counterparts in CBMC.

6.2 SV-COMP 2014 Benchmarks

SVComp is a "systematic comparative evaluation of the effectiveness and efficiency of the state of the art in software verification" [26]. We used benchmark verification tasks from SVComp to evaluate the correctness, effectiveness and efficiency of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and to compare its performance in practice with SVComp leading tools (CBMC [2], ESBMC [3], BLAST [4], and CPAChecker [47]). Each SVComp verification task has memory safety, termination, and error label reachability checks. Some of the tasks are $true$ where a proof is expected, and some are $false$ where a counterexample is expected. It is worth clarifying that the tools SVComp compares use various techniques and consequently their output differ in meaning when they report a $true$ result. We discuss the techniques in more details in Section 7. For completeness, and to put the following results in perspective, we qualify the meaning of a $true$ result for each tool as follows.

- CBMC and ESBMC:
  - with unwinding assertions: the program meets the specification within a bound on variable ranges since the unwinding assertion guards against insufficient unrolling.
  - without unwinding assertions: the program meets the specification within a bound on variable ranges, and within the specified recursion and loop unwinding bounds.
  - CBMC and ESBMC were configured to run with a sequence of fixed unwinding bounds and without unwinding assertions in SVComp.
  - According to the wrapper scripts provided from [26], CBMC and ESBMC report a $true$ result by default when they reach their timeout while the SAT or SMT solvers are running, they report an $unknown$ result when the timeout passes before they generate the first CNF or SMT instance to pass to the underlying solver. This is *probably reasonable* as CBMC and ESBMC consider the inability of the solvers to find counterexamples within the timeout period as an indication of the rarity or absence of such results. Thus we consider a $true$ result with a timeout time from CBMC and ESBMC as a timeout result for comparison purposes.
- BLAST is a counterexample based refinement approach that uses an SMT solver as a backend. If it completes analysis on all the trace formulae of a program, then it returns a $true$ result of the same value of ESBMC with unwinding assertions.
- The CPAChecker configuration used in SVComp was not clearly detailed in [26]. CPAChecker uses BLAST, shape analysis, and falsification based model checking as backend techniques. The meaning of its $true$ result depends on the technique that returns the result and also on the *merge*, *transfer* and *stop* operators that are supplied by the verification task and that switch the analysis between the different verification techniques.

- BLAST, and CPAChecker use infinite integer arithmetic with the SMT solver and thus are not 32- or 64-bit machine accurate. CPAChecker can be configured to be bit-accurate.
- A $true$ result returned from $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ means that the program is correct within a bound on variable ranges.

We evaluated $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ using the SVComp benchmarks and compared to the leading SVComp tools in the following sense.

- A $false$ non-spurious counterexample result presented by any of the tools has the same falsification value. They might differ though in their length and ease of use for debugging purposes which could be subject to future work.
- A non-conclusive result presented by any of the tools has also similar value. It reflects a degree of confidence that counterexamples are not easy to find by a specific technique within the same computational resources. The confidence is more reinforced when all tools return a $true$ or a non-conclusive result.
- A $true$ result from all the methods is a reinforcing result and is a good metric to evaluate (1) the correctness of the implementation of the tool and (2) the effectiveness of the benchmarks.
- Most importantly, while in theory the methods and their results differ, in practice the code of a major portion of the $true$ benchmarks includes explicit bounds on data elements, and limits on termination. The bounds and limits are hardcoded in the programs in the form of `if` conditions limiting parameters that eventually decide array size and termination. For example, the non-deterministic input in the `eca` benchmarks is limited between 1 and 6 using `if` conditions. The m and n inputs of the $true$ Ackermann function are limited between 1 and 3 for m and 1 and 23 for n. The input to a $true$ Fibonacci recursive call is once fixed to 9, and another time limited between 0 and 46, which limits the depth of the recursion. *This effectively makes the quality of the results of the tools on these specific benchmarks of relatively equal value.*

We selected the benchmarks that required no or minor syntax modifications for the $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ front end. We applied the minor modifications manually where needed. For $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ we used 4 as the bit width bound, 15 as the maximum number of array elements, and 15 as the maximum recursion depth. We increased the bounds appropriately where we received syntax errors such as an explicit constant $\geqslant 2^8$ or an explicit array size > 15. Table 4 reports the evaluation with SVComp results where some of the leading tools successfully completed the verification task and compares against them in terms of rank and running time. Table 5 reports the results where $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ succeeded to complete the verification tasks and the leading tools either timed out, reported runtime errors, or reported a wrong result (produced a counterexample when the benchmark is correct, or produced a claim of correctness when the benchmark had a known defect). Both tables show the name of the benchmark, its category and the running time needed by the tools to complete verification with a timeout of 900 seconds as required by SVComp regulations. The category denotes an SVComp classification of the benchmark based on its origin and intended functionality.

In the experiments, we used an automated script that ran reduction algorithms first with a timeout of 10 minutes with Table 4 and three minutes for Tables 4 and 5, followed by three instances of verification using the `pdr`, `dprove`, and `ind` ABC commands with proper timeout, number of frames, and BDD size configurations. We ran our experiments on a machine with similar settings to the server machine reported in [26] (3.4 GHz 64-bit Quad Core CPU, a 64 bit GNU/Linux operating system, and 32 GB of RAM).

In summary, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ reported no wrong results while CBMC, ESBMC and BLAST reported wrong results. CPAChecker reported no wrong results but reported runtime errors as shown in Table 5. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ automatically and correctly completed the first seven verification tasks of Table 5 that all four leading tools failed to complete correctly. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ reported running time comparable to the leading tools on most of the verification tasks and ranked first and second in the ControlFlowInt category for the $true$ verification tasks. The automated $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ script returned the expected result within the time limits of SVCOMP on all of the verification tasks except for the last five tasks in Table 5 where it timed out.

Interactive sessions. The automated $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ script timed out without producing a conclusive result for the last five verification tasks in Table 5. The leading tools also failed to complete verification for two of those tasks, and BLAST failed to complete verification for four out of the five tasks. We inspected those benchmarks and we were able to verify several of them interactively via trying ABC synthesis reduction and verification commands and techniques with different configurations and settings. This is not possible with the leading tools.

6.3 Discussion

Abstraction at program level. BLAST and CPAChecker did better in running time than $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and the rest of the tools for several verification tasks. They both use counterexample based abstraction and refinement and thus translate only a part of the program into the SMT solver. That part grows incrementally with each refinement step. This pays off very well in comparison to CBMC and ESBMC which use the translation of the whole program against the SAT solver especially in cases where the $true$ problem is provable using the first abstraction iterations, or the $false$ problem is detected with a non-spurious counterexample generated from the first abstraction iterations. The $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ automated script uses AIG abstraction and refinement techniques as well, however, it currently pays the overhead of translating the whole program into AIG and trying the reduction techniques before the verification techniques in all cases. We modified the script a little to try AIG abstraction and refinement before the reduction techniques when the AIG is of a small size, and we obtained slightly better results. In the future, we will investigate (1) performing abstraction at the program level, (2) using the $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ method as a solver with the larger refinements formulae, and (3) trying several paths of ABC verification commands where abstraction is used first in some of them.

Falsification. The automated $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ script performed well in several falsification tasks, however, it ranked third and fifth in running time for several of them. In addition to difference in overhead in comparison to the abstraction and refinement based techniques, and the mandatory overhead of using the reduction techniques before the verification ones in the automated $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ script, we found one more important reason. Once a counterexample is found, the $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and ABC commands try to minimize the length and size of the counterexample when they try to lift it back from one ABC technique to the other [48]. This hopefully improves the debug time once the counterexample is presented to the user to fix the defect. Up to our knowledge, other tools directly translate the counterexample produced from the backend solvers and present the translation to the user as is. CBMC has a `-beautify` option that might improve on the initially computed counterexample.

AIG sequence of commands. We experimented with and inspected several of the ABC reduction and verification techniques and devised a sequence of reduction and verification commands that we used with the presented benchmarks. The automated $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ provided excellent, good and acceptable run time results most of the time. However, the commands entailed some acceptable, yet not necessary, overhead for several benchmarks. We also had to interactively complete proof for five of the SVComp benchmarks.

Upon inspection and experimentation, we think that automating the selection of a successful and efficient sequence of reduction techniques is challenging. First, there are infinitely many sequences of ABC commands. Second, irreversible techniques such as abstraction require backtracking with additional memory required to save intermediate results. Third, AIG structural element metrics are good in general to judge the effectiveness of a reduction, however, at times a technique may reduce one metric but increase another. We discuss guidelines on selecting ABC commands and discuss the potential of automating the guidelines in Appendix B, available in the online supplemental material.

In the future, we plan on exploring a large number of traces of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ runs against software verification benchmarks to devise an expert $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ system. The system may use statistical based inference to infer the next reduction or verification command given values of structural and other easy to compute AIG metrics.

7 Related Work

VCC [5] is an industrial strength verification framework for concurrent low level C programs. It has a ghost type state system that tracks the validity of memory objects; i.e., references to memory objects do not overlap type states. It generates verification conditions that can be checked by Boogie; i.e., a high order logic analyzer. Boogie in turn uses the Z3 SMT solver [49] for automatic verification and Isabelle [50] for interactive verification. FrankenBIT [51] takes a bit-vector program with a verification condition and computes an unsound approximation (not over and not under) of the verification condition. Then it uses a logic solvers to decide the original verification condition strengthened with the inductive unsound approximation.

LLBMC [52] is a bit precise bounded model checker for low level C programs that works on an intermediate assembly representation of the program using an SMT solver. Ultimate Automizer [53] computes *appropriate* abstractions of programs based on statements as alphabet atoms in an automata framework. The work in [54] is implemented within the CPAChecker [47] platform to verify *event condition action* (ECA) systems using BDDs. Our method allows the use of symbolic model checking through ABC that makes use of BDD without the need to restrict ourselves to ECA systems. The CPAChecker [47] framework passes a program $\mathcal{S}$ and an invariant Ψ to several verification methods with different verification precisions such as model checking, falsification techniques, and counterexample based refinement. CPAChecker uses *configurable verification* with merge, transfer and stop operators that allow the transition from one verification method to the other. In case a verification method returns an inconclusive result, CPAChecker uses the partial result with the transfer operator to formulate a state predicate formula Φ where the invariant should hold. This incrementally reduces the work of the next verification method that checks $\neg\Phi$. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ uses ABC in a similar manner to merge and transfer results from one reduction or verification technique to the other. It differs in that the transfer and merge operators are predefined and guaranteed to be sound while they are configurable with CPAChecker. BLAST [4] is the predecessor of CPAChecker. It uses counterexample based abstraction and refinement techniques to check temporal properties of C programs. It computes abstractions of the program in terms of trace formulae (TF) and checks each one for satisfiability using an SMT solver. It avoids excessive unrolling by computing interpolants from refuted TF. Once a TF counterexample is reported, BLAST checks whether the counterexample is spurious or not. If spurious, BLAST refines the TF abstraction by strengthening the loop and trace invariants and tries again. UFO [55] provides a framework for exploring abstraction and interpolation techniques for software verification and uses SMT solvers as a backend.

CBMC [2] is a bounded model checker for ANSI-C programs that checks for properties such as pointer safety, array bounds and user assert statements. Given an ANSI-C program, a bound on the range of variables and a bound on recursion and loop unwinding, CBMC unrolls loops and recursive calls up to the unwinding bounds, and employs the single static assignment (SSA) technique to eliminate multiple variable assignments. Then CBMC encodes the resulting program with the assertion checks into a Boolean formula in CNF and checks the formula with SAT solvers. CBMC has an option to inject assertions that fire in case the unwinding bounds were not enough for loop and recursion termination. In that case, when the SAT solver returns a counterexample, CBMC checks whether an unwinding assertion was fired. If so, it increases the unwinding bounds and tries again. CBMC relies mostly on the power and speed of SAT solvers. SAT solvers often face an exponential blow up in the number of possible assignments to the atomic propositions. This problem, known as *state explosion*, and the large number of variables and

clauses used in the CNF encoding, limit CBMC analysis to relatively small bounds.

ESBMC [3] uses SMT solvers to verify multi-threaded C programs by forcing bounds on the number of context switches, loop iterations, and recursive calls. It also uses an aligned memory model that resolves pointers. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ differs in that it uses AIG solver and does not need to force bounds for loop iterations. In the future, we plan to extend our approach to concurrent programs by mapping a limit number of execution threads to separate fixed program counters. Similarly to CBMC, ESBMC has an option to inject unwinding assertions.

The work in [32] uses an encoding similar to that of CBMC, but instead of producing a CNF formula it produces an SMT formula. The SMT formula allows for variables with no range bounds. The loops are still unrolled up to a finite bound and loop completion assertions fire in case the bound imposed on the number of loop iterations was small and the loop guard was still satisfiable. The higher level solver may now decide to reproduce a new SMT formula with bigger loop unrolling bounds and call the SMT solver on the new formula. We differ in that our encoding to AIG circuits requires no bounds on loop iterations. This allows for more succinct representation of programs than with SMT.

Java PathFinder [33] (JPF), is a popular explicit-state model checker for Java programs. It implements a customized Java virtual machine (JVM) that supports state backtracking and allows programs to check properties of a wide range of Java programs without the need to respecify the programs in specialized languages. JPF does not apply program transformations. JPF also does not scale well in the presence of loops and branches with long running time. JPF operates at the word level, however, it does not make use of program specific properties and exhaustively searches the tree of possible executions. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ differs in that it operates at the bitlevel and it apples reduction techniques that allow backtracking algorithms to run faster as they operate on a smaller state space.

The *symbolic execution* extension to JPF (JPFSE) [46], [56] extends JPF with symbolic execution at the bytecode level to provide automated test case generation with *high path coverage* for inputs from unbounded domains. It performs symbolic model checking while generating test cases and reports defects that relate to exceptions such as array out of bound violations, null pointer access, and user defined assertions. JPFSE uses JPF to explore the state space of the program, employs *lazy variable initialization*, generates path conditions and resolves them with off the shelf SMT solvers. JPFSE addresses infinite execution trees caused by loops with (1) iterative deepening up to a bound on depth for counterexample generation, (2) breadth first search for test case generation, and also supports (3) a heuristic based search [56]. It does not target a guarantee that the software is error free. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ differs in that it targets model checking with *full path coverage* for inputs within *bounded* domains, operates at the bit-level rather than the bytecode level, and reduces the problem with synthesis reduction techniques before applying verification techniques including bit level symbolic engines.

The work of Xie et al. [35] translates software artifacts into equivalent semantics that are model-checkable. It applies compositional rules to the translated system to build its formal semantics in the context of message passing systems. We differ in that we use a specific program counter based semantics with a finitization rule to translate the software artifacts to an AIG, a model checkable formalization, where reductions, including compositional ones, can be used.

Work to render compiler level optimizations useful for verification exists [57], and uses techniques that originally aim at reducing run time to (1) reduce the state space of the program and (2) lift the resulting counterexamples to the original program. However, compiler transformations are sophisticated and machine dependent, and consequently not debug friendly. Consequently, engineers try to reproduce a defect trace originally produced with an optimized version of a program using a debug version of the same program without optimizations to facilitate debugging. These optimizations work at the word level, and once mature, they are complementary to $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ uses reduction techniques that work at the bit level, and therefore finds qualitatively different opportunities for reduction.

8 Conclusion

We presented $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ that takes a program and a pair of specifications, translates it into an AIG circuit, reduces the circuit using the ABC synthesis reduction techniques, and then checks the circuit for correctness using the ABC verification techniques. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ scales to bounds larger than that possible with existing tools, and solves verification tasks from the software verification competition benchmarks that leading tools at the competition did not solve within the specified timeout. In the future, we plan to extend $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ to verify multi-threaded programs. In short, our program counter based encoding enables the non-deterministic mapping of a program with up to N active threads into a system with up to K running threads using a simple scheduler with non-deterministic primary inputs that select the running threads. We plan to integrate $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ with existing frameworks that operate at word level such as CPAChecker, incorporate compiler optimizations to leverage both word and bit level techniques and extend our syntax coverage through existing front ends.

Acknowledgments

This work was funded by a two year University Research Board (URB) grant from the American University of Beirut.

References

[1] E. Torlak and D. Jackson, “Kodkod: A relational model finder,” in *Proc. Tools Algorithms Construction Anal. Syst.*, Mar. 2007, pp. 632–647.

[2] E. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” in *Proc. Tools Algorithms Construction Anal. Syst.*, Mar. 2004, pp. 168–176.

[3] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer, “ESBMC 1.22 - (competition contribution),” in *Proc. Tools Algorithms Construction Anal. Syst.*, Mar. 2014, pp. 405–407.

[4] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The software model checker Blast: Applications to software engineering,” *Int. J. Softw. Tools Technol. Transfer*, vol. 9, pp. 505–525, Oct. 2007.

[5] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, “VCC: A practical system for verifying concurrent C,” in *Proc. 22nd Int. Conf. Theorem Proving Higher Order Logics*, Aug. 2009, pp. 23–42.

[6] F. Zaraket, A. Aziz, and S. Khurshid, "Sequential circuits for relational analysis," in *Proc. Int. Conf. Softw. Eng.*, May 2007, pp. 13–22.

[7] F. A. Zaraket, A. Aziz, and S. Khurshid, "Sequential circuits for program analysis," in *Proc. 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Nov. 2007, pp. 114–123.

[8] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Proc. Formal Methods Comput.-Aided Des.*, Nov. 2004, pp. 159–173.

[9] N. Sorensson and N. Een, "MiniSAT v1. 13-a sat solver with conflict-clause minimization," in *System Description for the SAT Competition*, pp. 1–53, Jun. 2005.

[10] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. 22nd Int. Conf. Comput. Aided Verification*, Jul. 2010, pp. 24–40.

[11] T. Bybell, (2010), *GtkWave electronic waveform viewer*, [Online]. Available: http://gtkwave.sourceforge.net/.

[12] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *Proc. Comput.-Aided Verification*, Jul. 2001, pp. 104–117.

[13] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *Proc. 42nd Annu. Des. Autom. Conf.*, Jun. 2005, pp. 463–466.

[14] A. Kuehlmann, M. Ganai, and V. Paruthi, "Circuit-based Boolean reasoning," in *Proc. Des. Autom. Conf.*, Jun. 2001, pp. 232–237.

[15] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Proc. 3rd Int. Conf. Formal Methods Comput.-Aided Des.*, Nov. 2000, pp. 372–389.

[16] A. Aziz, T. Shiple, V. Singhal, R. Brayton, and A. Sangiovanni-Vincentelli, "Formula dependent equivalence for compositional CTL model checking," *J. Formal Methods Syst. Des.*, vol. 21, no. 2, pp. 193–224, 2002.

[17] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 2004, pp. 42–49.

[18] K. L. McMillan, "Interpolation and SAT-based model checking," in *Proc. 15th Itn. Conf. Comput. Aided Verification*, Jul. 2003, pp. 1–13.

[19] D. Wang, "SAT based abstraction refinement for hardware verification," PhD dissertation, Carnegie Mellon Univ., Pittsburgh, PA, USA, May 2003.

[20] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proc. Int. Conf. Formal Methods Comput.-Aided Des.*, Oct. 2011, pp. 125–134.

[21] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta, "Combining decision diagrams and SAT procedures for efficient symbolic model checking," in *Comput. Aided Verification*, pp. 124–138, Jul. 2000.

[22] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. 38th Annu. Des. Autom. Conf.*, Jun. 2001, pp. 530–535.

[23] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2000, pp. 120–126.

[24] J. Baumgartner, A. Kuehlmann, and J. Abraham, "Property checking via structural analysis," in *Proc. 14th Int. Conf. Comput.-Aided Verification*, Jul. 2002, pp. pp 151–165.

[25] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan, "An analysis of SAT-based model checking techniques in an industrial environment," in *Proc. Correct Hardw. Des. Verification Methods*, Oct. 2005, pp. 254–268.

[26] D. Beyer, "Status report on software verification," in *Proc. 20th Int. Conf. Tools Algorithms Construction Anal. Syst.*, Apr. 2014, pp. 373–388.

[27] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Solving difficult SAT instances in the presence of symmetry," in *Proc. 39th Des. Autom. Conf.*, 2002, pp. 731–736.

[28] Z. Fu, Y. Yu, and S. Malik, "Considering circuit observability don't cares in CNF satisfiability," in *Proc. Des. Autom. Test Eur.*, Mar. 2005, pp. 1108–1113.

[29] Q. Zhu, N. Kitchen, A. Kuelhman, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability dontcares," in *Proc. 43rd IEEE/ACM Des. Autom. Conf.*, Jul. 2006, pp. 229–234.

[30] S. A. Edwards, "The challenges of hardware synthesis from C-like languages," in *Des. Autom. Test Eur.*, pp. 66–67, Mar. 2005.

[31] A. P. Hurst, A. Mishchenko, and R. K. Brayton, "Fast minimum-register retiming via binary maximum-flow," in *Proc. Formal Methods Comput.-Aided Des.*, Nov. 2007, pp. 181–187.

[32] R. Arm, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," in *Proc. 13th Int. SPIN Workshop Model Checking Softw.*, 2006, pp. 146–162.

[33] W. Visser, K. Havelund, G. Brat, and S.-J. Park, "Model checking programs," *Autom. Softw. Eng. J.*, Apr. vol. 10, pp. 203–232, 2003.

[34] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli, "Computing finite models by reduction to function-free clause logic," *J. Appl. Logic*, vol. 7, pp. 58–74, 2007.

[35] F. Xie, J. C. Browne, and R. P. Kurshan, "Translation-based compositional reasoning for software systems," in *Formal Methods Europe*, pp. 582–599, Sep. 2003.

[36] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Inf. Comput.*, vol. 98, no. 2, pp. 142–170, 1992.

[37] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2008, pp. 234–241.

[38] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2004, pp. 42–49.

[39] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proc. 43rd Annu. Des. Autom. Conf.*, Jul. 2006, pp. 532–535.

[40] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electron. Notes Theoretical Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003.

[41] A. R. Bradley, "SAT-based model checking without unrolling," in *Proc. 12th Int. Conf. Verification, Model Checking, Abstract Interpretation*, Jan. 2011, pp. 70–87.

[42] T. Parr and R. Quong, "ANTLR: A predicated-LL (k) parser generator," *Softw.: Pract. Exp.*, vol. 25, pp. 789–810, 1995.

[43] A. R. Bradley and Z. Manna, *The Calculus of Computation: Decision Procedures with Applications to Verification*. New York, NY, USA: Springer, 2007.

[44] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[45] P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI C Specification Language (preliminary design V1.9)* [Online]. Available: http://www.frama-c.cea.fr/acsl.html.

[46] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proc. Int. Symp. Softw. Testing Anal.*, Jul. 2008, pp. 15–26.

[47] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: A technique to pass information between verifiers," in *Found. Softw. Eng.*, pp. 57:1–57:11, Nov. 2012.

[48] A. Mishchenko, N. Een, and R. Brayton, "A toolbox for counter-example analysis and optimization," presented at the *Int. Workshop Logic Synthesis (IWLS)*, Austin, TX, Jun. 2013.

[49] N. Bjrner and L. de Moura, "$Z3^{10}$: Applications, enablers, challenges and directions," in *Proc. 6th Int. Workshop Constraints Formal Verification (CFV)*, Grenoble, France, Jun. 2009.

[50] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL—A Proof Assistant for Higher-Order Logic.* New York, NY, USA: Springer, 2002.

[51] A. Gurfinkel, A. Belov, and J. Marques-Silva, "Synthesizing safe bit-precise invariants," in *Proc. 20th Int. Conf. Tools Algorithms Construction Anal. Syst.*, Apr. 2014, pp. 93–108.

[52] S. Falke, F. Merz, and C. Sinz, "The bounded model checker LLBMC," in *Proc. 28th Int. Conf. Autom. Softw. Eng.*, Nov. 2013, pp. 706–709.

[53] M. Heizmann, J. Hoenicke, and A. Podelski, "Software model checking for people who love automata," in *Proc. 25th Int. Conf. Comput. Aided Verification*, Jul. 2013, pp. 36–52.

[54] D. Beyer and A. Stahlbauer, "BDD-Based software model checking with CPAchecker," in *Proc. 8th Int. Doctoral Workshop Math. Eng. Methods Comput. Sci.*, Oct. 2012, pp. 1–11.

[55] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik, "UFO: A framework for abstraction and interpolation-based software verification," in *Proc. 24th Int. Conf. Comput. Aided Verification*, Jul. 2012, pp. 672–678.

[56] S. Khurshid, C. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proc. 9th Int. Conf. Tools Algorithms Construction Anal. Syst.*, Apr. 2003, pp. 553–568.

[57] D. Marinov, S. Khurshid, S. Bugrara, L. Zhang, and M. Rinard, "Optimizations for compiling declarative models into Boolean formulas," in *Proc. 8th Int. Conf. Theory Appl. Satisfiability Testing*, Jun. 2005, pp. 187–202.

Mohammad A. Noureddine received the bachelor and master degrees from the American University of Beirut (AUB) in 2011 and 2014, respectively. He is currently working toward the PhD degree at the Computer Science Department at the University of Illinois at Urbana Champaign (UIUC). Before joining UIUC, he worked on tools for software verification at AUB under the supervision of Professor Fadi Zaraket. In addition to software and logic verification, his research interests include secure and intrusion tolerant systems and the application of game theory for automated intrusion detection, response and recovery.

Fadi A. Zaraket received the bachelor and master degrees in computer and communication engineering from AUB in July 1996 and February 2001, respectively, and the PhD degree in ECE from the University of Texas at Austin in December 2007. He joined the Electrical and Computer Engineering (ECE) Department at the American University of Beirut (AUB) as an assistant professor in February 2009. He worked at IBM on logic verification and debugging tools between June 2001 and December 2008. He also worked at Sun Microsystems and Santa Cruz Operations on several projects including kernel development and cross platform portals between April 1999 and June 2001.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.