Sequential Circuits for Program Analysis

Fadi Zaraket
University of Texas at Austin
Austin, TX
zaraket@ece.utexas.edu

Adnan Aziz
University of Texas at Austin
Austin, TX
adnan@ece.utexas.edu

Sarfraz Khurshid
University of Texas at Austin
Austin, TX
khurshid@ece.utexas.edu

ABSTRACT

A number of researchers have proposed the use of Boolean satisfiability solvers for verifying C programs. They encode correctness checks as Boolean formulas using finitization: loops and recursion are bounded, as is the size of the input instances. The SAT approach has been shown to find subtle bugs with reasonable resources. However, it does not scale well; in particular, it lacks the ability to handle larger bounds. We present SEBAC, which can handle the same class of programs as the SAT approach, and scales to bounds that are orders of magnitude higher. The key difference between SEBAC and SAT techniques is SEBAC's use of imperative Boolean sequential circuits, which are Boolean formulas with memory elements instead of the Boolean formulas which are stateless.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification—*formal methods, model checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions, invariants, mechanical verification*

General Terms

Verification

Keywords

static analysis, program analysis, verification, model checking

1. INTRODUCTION

Several static analysis techniques have emerged lately addressing the verification of software programs [9, 17, 32, 31]. Software programs are undecidable and thus static analysis tools often resort to abstraction and finitization techniques to render them amenable to model checking. CBMC [9] is a bounded model checker for ANSI-C programs that checks for properties such as pointer safety and array bounds as well as user assert statements. Given an ANSI-C program and a bound on the range of variables therein, CBMC computes a Boolean formula that asserts the desired properties in the program. It does that by unwinding a complex state machine that describes the program and its properties into a Boolean formula in conjunctive normal form (CNF) and checks the formula using a satisfiability (SAT) procedure [13, 26, 6] to look for a counterexample.

SAT solvers often face an exponential blow up in the number of possible assignments to the atomic propositions. This problem, known as *state explosion*, along with the large number of variables used in the CNF encoding, often limits the SAT-based CBMC analysis to restricted variable ranges. By scaling the analysis of ANSI-C programs to larger bounds, we increase its applicability to real-world designs.

1.1 Sequential circuits for program analysis

While recent advances in SAT have enabled checking designs of real systems, these implementations often need to be substantially incomplete, leaving out important aspects of the systems, to enable the analysis to complete. Moreover, the analysis is typically limited to relatively small bounds, e.g., fewer than 16 entries in an array sort program as we will demonstrate in Section 5.

There are three limiting aspects of the current CBMC analysis. (1.) The translation to CNF depends on the bounds; a small increase in the bound on the variable ranges or the loop and recursion unwinding can cause a large increase in the size of the translated CNF formula. (2.) The SAT solver is restricted to using optimizations that apply at the level of CNF formulas. Often times when the analyzer successfully generates a large CNF formula, the underlying solver times out. (3.) CBMC has to translate the program into a CNF formula with a higher unwinding bound and invoke the SAT solver several times in case the unwinding bound on loops and recursion was not enough.

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

The limitations of the CNF encoding motivated us to develop *sequential encoding for bounded ANSI-C program analysis* (SEBAC), an algorithm which encodes ANSI-C programs as sequential circuits and decides them using a sequential circuit solver. A sequential circuit can be viewed as a restricted C++ program, specifically a multi-threaded program in which all variables are either Boolean-valued or

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

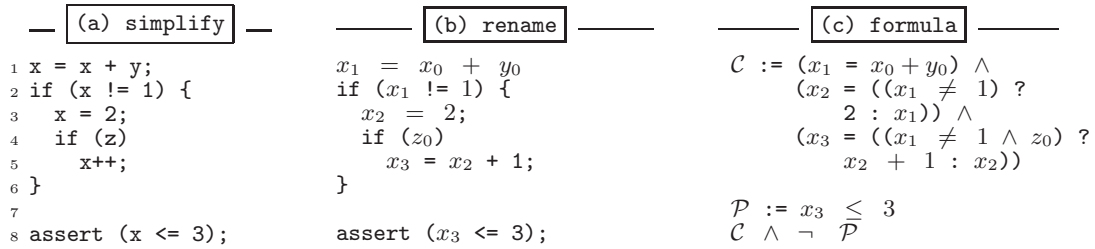


Table 1: CBMC transformation of ANSI-C programs into a Boolean formula

integers, whose range is statically bounded, and unbounded allocation is forbidden [33, 12].

Given an ANSI-C program and a bound, SEBAC automatically derives a sequential circuit and a Boolean variable therein that serves as an *invariant*, i.e., the variable can be set to *true* if and only if a property that the program asserts is violated within the bound.

We use *SixthSense* [23, 34], a tool developed at IBM, to automatically check invariants in sequential circuits. SixthSense reads designs expressed in the very high speed integrated circuit hardware description language (VHDL) [1]. Sequential circuits, as described in the preceding paragraph, can be translated into VHDL using inlining [11, 12].

In prior work we demonstrated that sequential circuit analysis scaled to bounds orders of magnitude higher than SAT analysis for checking satisfaction of formulas expressed in declarative first order logic (FOL) with transitive closure [33].

We hypothesize that a similar benefit holds for the analysis of imperative programs. We note that there are two key advantages to compiling ANSI-C programs into sequential circuits rather than CNF formulas:

Advantage 1 Our encodings are much more succinct than those generated by CBMC—in cases, CBMC encoding algorithms produce a data structure that uses several orders of magnitude more memory. For instance, CBMC needed a CNF formula with 4.7 million variables and 18.9 million clauses to encode an array sorting routine with a bound of 16 on the size of the array and the range of the values for the array entries.

Advantage 2 Casting the decision problem for a property of an ANSI-C program as an invariant check on a sequential circuit allows us to make use of a number of powerful automated analysis techniques that we discuss in Section 3.4 and that have no counterpart in CNF analysis. Empirically, our results show that SEBAC scales to bounds that are orders of magnitude higher compared to CBMC.

Intuitively, Advantage 1 holds because sequential circuits are imperative and stateful while CNF formulas are declarative and state-free. For example, sequential circuits can naturally represent the execution of sequential loops without the need for unwinding. Moreover, sequential circuits can store and reuse intermediate results in local variables.

We justify Advantage 2 by illustrating some automatic analysis techniques that SixthSense implements. These directly apply to sequential circuits generated by SEBAC.

Abstraction. Consider the verification of library code L which uses a sophisticated memory allocator for better performance.

Let the library $L*$ be L with L's allocator abstracted to a simpler allocator that nondeterministically selects a block from the set of free blocks. Since the simpler allocator uses nondeterminism, if an invariant holds of $L*$, it holds of L. The simpler allocator in $L*$ makes verifying invariants on $L*$ easier than verifying the same invariants on L.

While there exist efficient algorithms for automatically identifying components for abstraction in sequential circuits [22, 23], abstraction for CNF formulas is much harder. This is because there is no structure in a CNF formula to guide the abstraction algorithm—the clauses are unordered.

Note that an invariant may fail on $L*$, but hold of L, e.g., L's code makes use of details from the implementation of the allocator beyond those exported from the abstract interface. SixthSense will automatically identify a negative as false, and roll back the abstraction [22, 23].

Compositional minimization. Consider the verification of a spanning tree algorithm T which uses a balanced search tree (BST) to manipulate sets.

With respect to its abstract interface, a BST implementation of sets is functionally equivalent to a list implementation of sets. Let spanning tree algorithm T' be T with sets implemented using lists. Because a BST is more complex than a list, verification of T' is easier than verification of T. Since the list and BST representations of sets are equivalent with respect to their abstract interface, an invariant holds of T iff it holds of T'.

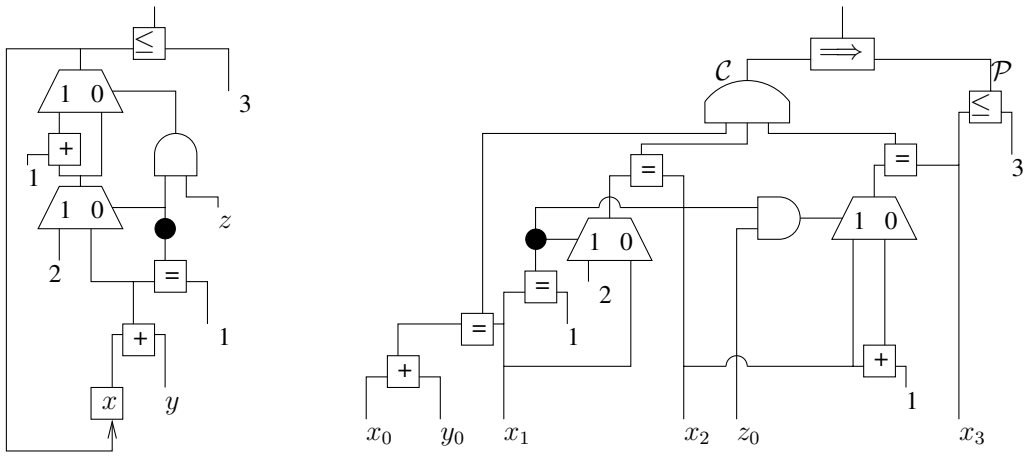
There exist several techniques for automatically identifying components and minimizing them in sequential circuits [7, 34, 3]. These techniques are based on the notion of equivalent states [18]. Analogous techniques do not exist for CNF formulas, as there is no notion of state.

Logic optimizations. There is an arsenal of techniques for automatically optimizing sequential circuits; examples include variable minimization via retiming [20], common subexpression extraction [8], and exploiting reduced observability and controllability at internal components [2, 28].

SixthSense automatically iterates through the techniques described above as well as some other techniques, to simplify its input sequential circuit. It decides the simplified circuit using a variety of techniques, such as bounded model checking, circuit SAT solving, invariant enlargement, and semi-formal search [25, 26, 16, 5, 23].

We make the following key contributions:

1. **Sequential analysis:** We enable the use of sequential circuit verification including many powerful reduction techniques for ANSI-C model checking.



(a) sequential circuit (b) Boolean formula before CNF

Figure 1: Sequential circuit encoding versus the CBMC Boolean formula

2. **Create verification flow for existing encoding:** We enable a software static analysis flow from SPARK, a fully automated hardware high-level synthesis tool that translates C to sequential circuits [14]. SPARK has not been used for verification purposes before.
3. **New encoding for ANSI-C programs:** We propose SEBAC, a novel algorithm to encode an ANSI-C program with a bound on ranges of variables into a sequential circuit. The SEBAC encoding is more optimal for verification purposes than the SPARK encoding as the latter targets optimizations such as time multiplexing and circuit area reduction on the expense of increasing the number of variables in the circuit.
4. **Evaluate TBV for C programs:** We evaluate TBV analysis of C programs by comparing SPARK and SEBAC coupled with a TBV solver against CBMC coupled with SAT. We apply these techniques to find real and subtle code bugs that were reported by Adam Barr from Microsoft as challenging and require careful code inspection to be revealed [4]. As our results show, sequential analysis techniques scale to bounds orders of magnitude higher than CNF and SAT techniques.

The rest of this paper is structured as follows. In Section 2 we illustrate our motivation using an example from [9]. In Section 3 we illustrate CBMC, sequential circuits, SPARK, and describe how SixthSense works. We introduce SEBAC and argue for its correctness in Section 4 We evaluate our approach in Section 5 and conclude in Section 6.

2. ILLUSTRATIVE EXAMPLE

We show in Column (a) of Table 1 the same example code used to illustrate the CBMC translation from ANSI-C into CNF in [9]. The first step is to transform the code into a static single assignment form where each variable is assigned only once. This is done by introducing new variables through variable renaming and the result is shown in Column (b) of Table 1. Then CBMC computes a set of constraints $\mathcal{C}$ and properties $\mathcal{P}$ and builds a Boolean formula $\mathcal{C} \wedge \neg\mathcal{P}$ by treating variables as bit vectors. A satisfying valuation to variables of this formula represents a counterexample. We show the combinational circuit corresponding to $\mathcal{C} \wedge \neg\mathcal{P}$ in Figure 1(b). The formula is then flattened into CNF which has only two levels of logical hierarchy ($\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} l_{ij}$ where n is the number of clauses, m_i is the number of variables in clause i, and l_{ij} is a literal expressing either the variable indexed by j in clause i or its negation). In the process, the structure of the circuit is lost. In contrast, the sequential circuit encoding shown in Figure 1(a) introduces no new variables and keeps the hierarchy of the program.

Note that in case the code in Table 1(a) was enclosed in a loop, CBMC would assume an unwinding bound w and eventually replicate the same circuit in Figure 1(b) w times with new variables each time. If the check on the CNF formula resulting from the unwinding with w fails because the unwinding was not enough, CBMC would increase w and re-encode the CNF formula. In contrast, the sequential circuit encoding in Figure 1(a) stays intact and expresses the loop via updating the state of x through executing the circuit as much steps as needed per the bounded model checker.

3. CBMC, SEQUENTIAL CIRCUIT ANALYSIS, AND SPARK

In this section we introduce the CBMC analysis of ANSI-C programs. We briefly introduce sequential circuits. We compare our encoding in SEBAC, which targets optimizations useful for static analysis, to the output of SPARK [14]; a fully automated high-level synthesis tool that generates sequential circuits out of restricted C programs. SEBAC performs better since it infers the sequential structures from within the program while SPARK treats every statement as a sequential step and uses various techniques to find statements that can be executed concurrently. Finally, we describe how transformation-based verification checks properties of sequential circuits.

3.1 CBMC

CBMC reduces the problem of checking a property of an ANSI-C program with bound N on the range of variables and with a vector of bounds W on the number of loop and recursion unwindings to the problem of determining the sat-

isfiability of a Boolean formula. To translate an ANSI-C program, CBMC unwinds loop constructs that can occur as `for` and `while` loops, backward-jump `goto` statements, as well as recursive function calls.

The `for` and `while` loops are unwound w_i times where w_i is the unwinding bound and i is the index of the loop into the vector of bounds W. An assertion is added to the last copy to ensure that the loop does not need more iterations to complete the computation. An unwinding assertion guides CBMC to increase w_i for a certain loop in case the unwinding assertion fails. Recursive function calls are assumed to recurse to a bounded depth and an assertion similar to the loop unwinding assertion is added to ensure that depth is enough. The backward-jump `goto` statements inducing loops are handled similarly.

Function calls are inlined within the calling function. The `return` statements are replaced by assignments and `goto` statements pointing to the end of the function expansion. This construction results into a simplified program with `if`, forward `goto`, assignment, assertion, and arithmetic statements as well as labels defining branching targets.

Table 1 illustrates through an example the translation from the simplified ANSI-C program into a static single assignment form. Column (a) in Table 1 shows a simplified piece of code. CBMC renames the variables so that each variable is assigned only once and the result is shown in Column (b) of Table 1. In the process variables x_1, x_2 and x_3 were added. Then CBMC computes a set of constraints $\mathcal{C}$ on all the variables, including the ones it introduced, and a set of properties $\mathcal{P}$ deduced from the assertions in the simplified program.

CBMC translates the formula $\mathcal{C} \wedge \neg\mathcal{P}$ into a CNF formula by considering each variable as a bit vector with sufficient width to represent the bound on the variable range.

When CBMC introduces new variables in the so-called renaming transformation, it is actually embedding internal hierarchical nodes in the structure of the program as illustrated in Figure 1(b). However, CBMC loses this hierarchy when it translates $\mathcal{C} \wedge \neg\mathcal{P}$ into CNF because CNF is a flat Boolean formula with only two levels of hierarchy (AND of ORs). Also CBMC needs may need to introduce new variables to represent the internal nodes since its final target (CNF) is stateless.

3.2 Sequential circuits

In Figure 1(a) we show a sequential circuit encoding of the ANSI-C program from Table 1. It keeps the exact hierarchy of the ANSI-C program and does not require additional variables since it has register variables; memory elements that keep the state of the circuit.

A sequential circuit is easily understood as a C++ object with Boolean variables describing its states and member functions describing its output and computation.

```
class SequentialCircuit {
  Boolean registerVariables[];
  Boolean done;

  void initialState( Boolean inputs[]);
  void nextState( Boolean inputs[]);
  Boolean outputFunction(Boolean inputs[]);

  Boolean executeCircuit(Boolean inputs[]){
    initialState(inputs);
    while(!done)
      nextState(inputs);
```

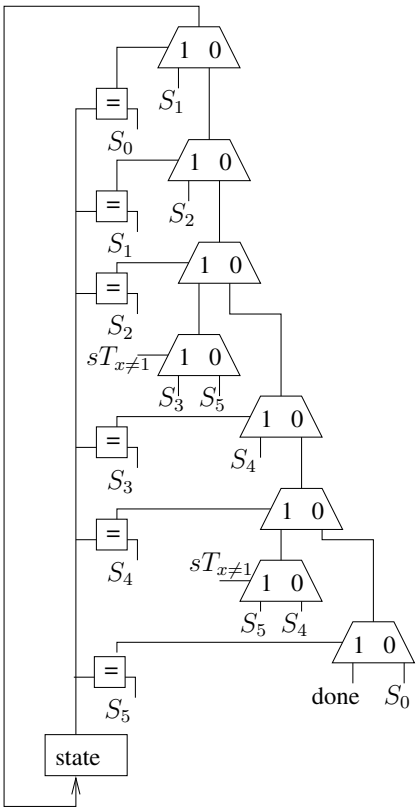


Figure 2: SPARK control flow circuit

```
    return outputFunction(inputs);
  }
}
```

The class `SequentialCircuit` has a set of register variables that describe its state along with the `done` variable that denotes the completion of the computation. It also accepts primary input variables as parameters to its functions.

The `initialState` function assigns the initial state of the sequential circuit by assigning values to its register variables that are restricted to be combinational functions of the `inputs`. The `nextState` function updates the state of the sequential circuit by assigning values to the register variables that are combinational functions of the `inputs` and the `registerVariables` themselves. The `nextState` function performs the computational function of the sequential circuit and assigns `done` to a `true` value once it completes.

The function `executeCircuit` actually executes the circuit by calling the `initialState` function to initialize it and then calling `nextState` to perform the computation. It then returns the output of the computation via calling the `outputFunction`. The `while` in `executeCircuit` models time where each iteration is a *step*.

The semantics of a sequential circuit are defined with respect to *input sequences*, where an input sequence is a sequence of assignments to `inputs`.

Given an input sequence and an initial state, the resulting *trace* is a sequence of Boolean assignments to all Boolean register variables in the circuit. A *transition* happens when an assignment changes at a specific step. The output of the circuit is *justifiable* if there is an input sequence which when

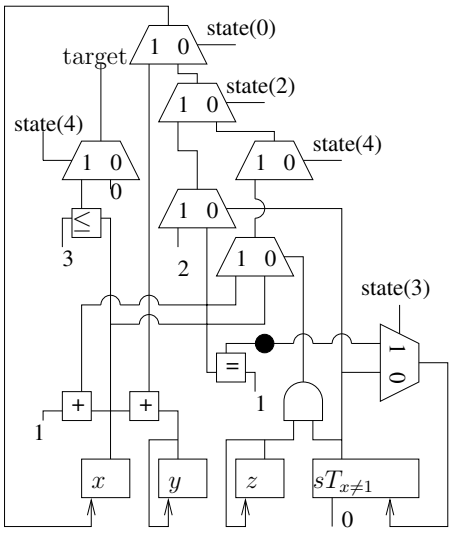


Figure 3: SPARK data flow circuit

applied to an initial state will result in the `outputFunction` returning the value *true*.

The sequential circuit in Figure 1(a) has one register variable x and two input variables y and z. The initial state of x sets x to a nondeterministic value, and the next state is a function of x, y, and z. The sequential circuit needs one step to complete and the output function returns the assertion in Column (a) of Table 1.

We compare that to Figure 1(b) which shows the Boolean formula CBMC generates before translating it to CNF. Notice that this formula has yet to be flattened into CNF and thus more variables may be introduced to transform it into two levels of hierarchy only.

3.3 C to sequential circuits using SPARK

SPARK is a fully automated high-level synthesis tool [14] that is designed for implementing systems and not for verifying them. It takes a subset of ANSI-C constructs and bounds on variable ranges as input and produces a synthesizable register-transfer level VHDL that describes a sequential circuit as output.

Briefly, SPARK recognizes atomic statements in the ANSI-C code and constructs a state machine that executes the program a statement per step. Then SPARK uses high-level synthesis techniques to schedule the execution of these statements in order to optimize performance, reduce the area of the circuit, or better utilize the limited hardware resources [14, 15].

SPARK performs renaming techniques and a set of heuristic transformations called *code motions* to move and merge the execution of the atomic statements inside and outside of conditionals and loop constructs. By doing so SPARK introduces new sequential elements to the ANSI-C program by assuming the execution of one statement per step. Then it tries to reduce the number of steps it needs to execute the ANSI-C program via heuristics that allow concurrent executions of independent atomic statement.

In Figures 2 and 3 we show how SPARK translates the example in Table 1 into a sequential circuit. Figure 2 shows the control flow circuit that implements the schedule computed by SPARK. Figure 3 shows the data flow circuit that executes the atomic computation statements. One can think of the two figures as two processes that execute concurrently and share variables. The variable `state` transitions between states $S_0, \ldots, S_5$, and done. SPARK introduced one bit-vector valued variable that encodes the state of the circuit and controls the schedule of execution. This is substantially less than the variables CBMC introduces.

The optimizations and transformations SPARK performs are not targeted to obtain a sequential circuit that is better amenable for static analysis. SPARK targets optimizations like choosing between complex structures such as ripple carry versus tree adders, reducing gate delays, and time multiplexing operations onto a few functional units. For example, we see that in Figure 3 SPARK minimized the logical depth of all combinational functions on the expense of needing more steps to complete the computation. We also observe that SPARK introduces additional sequential behavior to the program through its separate schedule state machine in order to provide a flexible infrastructure for the so called code motion transformations. For example, and without going into details, some code motions can be done by simply assigning new encodings for the states S_0 through S_5 to allow the execution of more than one statement at the same time.

3.4 Transformation-based verification

TBV applies reduction techniques iteratively on a sequential circuit to reduce the verification complexity by reducing the number of logical operations, register variables, and inputs in the circuit. Then it attempts to solve the problem via decision techniques such as bounded model checkers, circuit SAT solvers, or semi-formal searches. The decision techniques aim to find a satisfying *trace*, that is an assignment to the initial value functions of the register variables and a sequence of input valuations that result in asserting the output function to a `true` value at the last step of the trace. In Table 2, we briefly describe various transforms that were used in the context of this work and comment on their efficiency. SEBAC automatically found successful flows of transforms using techniques in [23].

Sequential circuits can be analyzed with public domain tools with TBV capabilities such as SIS [29] and VIS [24].

4. SEBAC

Given an ANSI-C program with assertions therein and a bound on the ranges of variables, SEBAC produces a sequential circuit and a Boolean invariant therein such that there is a trace of the sequential circuit that sets the invariant to `true` if and only if an assertion is violated.

Recall that CBMC translates an ANSI-C program with a bound on the ranges of variables to a Boolean formula via (1.) introducing new variables and (2.) embedding a constraint system hierarchy into the program. Then it loses even that when it flattens the Boolean formula into CNF before passing it to SAT. SPARK, in contrast, uses less variables than CBMC to encode the bounded program into a sequential circuit, it actually introduces state and sequential elements to the program that are not necessary for its execution. It also adds scheduling complexity and encodes the schedule control states in a way that allows flexible code motions on the expense of adding more Boolean variables.

Table 2: TBV algorithms in order of increasing complexity top to bottom.

COM	merges functionally equivalent logical gates using low complexity analysis [21].
EQV	makes intelligent guesses on equivalency [7] and performs expensive checks that allow huge gate merging reductions when they pass. It also exploits structural symmetry detection.
RET	reduces the number of registers by shifting them across combinational gates [20].
BMC	attempts a conclusive result within a given limit on steps or resources [25].
PRE	performs compositional minimization [34] by isolating a component in the sequential circuit and detecting equivalent states within the component, and then reduces the input space of the component.
BIG	replaces a target by a re-encoding, i.e., a set of states which will hit that target within k time steps [23].
LOC	This is a localization based *abstraction* and refinement engine. It overapproximates the target by replacing the gates on a boundary with free variables [23].
CUT	replaces a set of gates with a simpler yet equivalent sequential circuit; reduces input count via defining inputs as functions of each other [23].
SCH	conducts a semi-formal search for a target using a hybrid approach of random simulation, symbolic simulation, and induction [16, 21].

4.1 Overview

Intuitively, SEBAC aims at reducing the number of variables needed to sequentially encode the ANSI-C program and at reducing the sequential depth needed for the circuit to complete execution, thus allowing bounded model checking a better chance to complete the analysis. Unlike CBMC that uses the assignment statements to build a constraint hierarchy that gets lost later after translation to CNF, SEBAC uses the data structures and the branching logic in the program to build a circuit hierarchy.

In contrast to SPARK, SEBAC infers sequential elements from the semantics of the program instead of introducing sequential elements that are based on the ANSI-C semantics of sequential statements. SEBAC makes use of sequential elements such as dependent code blocks and loop iterators. To achieve this, SEBAC simply ignores high-level synthesis techniques, and translates the ANSI-C program into literally syntactically equivalent VHDL processes that use high-level VHDL constructs such as records, loops, and overloaded functions to match the ANSI-C high-level constructs. It also makes use of VHDL generic parameters to simplify specifying bounds for the program.

4.2 Correctness of SEBAC

Similar to Clarke *et al.* [10], we follow the C99 semantics of ANSI-C programs as modeled in [19]. The formal description and proof of correctness of SEBAC is based on specifying a grammar for C and sequential circuits, and a recursive algorithm that takes nodes from the parse tree and the data flow diagram of the C program P and maps them to a sequential circuit SEBAC(P) with an invariant $A(P)$. A proof that the invariant $A(P)$ is satisfied if and only if an assertion is violated in P is based on induction on the length of the parse tree of P.

We will demonstrate the equivalence of a SEBAC encoding to that of CBMC for those constructs where the encoding differs from or does not match directly with the ANSI-C semantics as in Boolean and integer arithmetic and assignments. As we described in Section 3.1, CBMC assumes the program is already preprocessed, and performs a series of transformations and rewrites to the program. The first transformations expand function calls and replace `break`, `continue`, `switch/case`, `for`, and `do/while` statements with equivalent `if/else` and `while` statements.

CBMC treats recursive functions and loops inferred from backward `goto` statements the same way it treats `while` loops by unwinding them. These transformations leave us with an ANSI-C program that consists of `if`, assignment, assertions, and `while` instructions.

We limit our attention to the correctness of encoding loops and pointers since the rest of the instructions map directly to VHDL with the same semantics.

CBMC performs additional transformations such as variable renaming and loop unwinding. A one step unwinding of `while(f){F}` will be `if(f){F; while(f) {F}}`. After many unwinding steps the tailing `while(f){F}` loop is replaced with an assertion `assert(!f)`. The assertion is essential to guarantee that the unwinding was enough. If the assertion fails for any possible execution, then the number of unwindings is increased until it is big enough.

SEBAC builds a sequential component for the same loop that executes `F` and evaluates `f` and sets its `done` property only when `f` evaluates to false. The semantics are similar to `while(!done){ whileLoop.nextState();}` with `nextState(){ if(!done){ F; done = !e;}}` where each iteration is a cycle as described in Section 3.2. The encoding is equivalent to the original loop with a simple expansion of the body of `nextState` and a substitution of `done` with the right handside of the assignment.

SEBAC treats pointers as indices into an array representing memory. This is correct since in ANSI-C `*p` is equivalent to `p[0]` and `*(p +i)` is equivalent to `p[i]` where `p` is a pointer to memory and `i` is an integer.

4.3 Mapping C to VHDL

We illustrate the process of mapping C to VHDL by looking at a buggy array selection sort algorithm from [4] with correctness properties checking whether the array is in order and whether an arbitrary entry in the array still exists

in it after the sort. We highlight interesting constructs and explain the details of the sort while illustrating the mapping.

Selection sort with bug in ANSI-C

```
1 /*! \brief array selection sort routine*/
2 void selsort (int a[], int size) {
3  int current, j, lowestindex, temp;
4
5  for (current=0; current < size - 1; current++) {
6   lowestindex = current;
7    /* find the index of the lowest value */
8    for (j = current + 1; j < size; j++) {
9     if (a[j] < a[current]) {
10     lowestindex = j;
11    }
12   }
13   if (lowestindex != current) {
14   /* swap a[current] and a[lowestindex]
15    * since difference exists*/
16    temp = a[current];
17    a[current] = a[lowestindex];
18    a[lowestindex] = temp;
19  }
20 }
21 }
22 /*! \brief checker for array selection sort */
23 void selsortproperty (int size) {
24  int a[64];
25  int i, iTest, jTest, aTest;
26
27  if (size == 0) return;
28  iTest = iTest % size;
29  jTest = jTest % size;
30  aTest = a[iTest];
31
32  selsort (a, size);
33
34 /* check if in order */
35  if (iTest < jTest )
36   assert ( (a[iTest] <= a[jTest]));
37
38 /* check if consistent */
39  for (i = 0; i < size; i++ )
40   if (aTest == a[i])
41    break;
42  assert(aTest == a[i]);
43 }
```

C Function to VHDL entity. SEBAC translates the `selsortproperty` function into a VHDL entity. The entity takes `size` as an input and `bound` as a generic parameter that configures the range of `integer` variables.

```
entity selsortproperty is
  generic (bound : integer := 64);
  port (signal size : integer range 0 to bound - 1);
end;
```

The behavior of the entity is described in a corresponding VHDL `architecture`.

```
architecture selsortproperty of selsortproperty is
  constant bound_lg2: integer := util_log2(bound);
  signal a : IntArray( 0 to bound - 1);
  signal i, iTest, jTest, aTest : integer range 0 to bound - 1;

  signal current: integer range 0 to bound - 1;
  signal currentV, currentVN : std_ulogic_vector(0 to bound_lg2);
```

Sequential loops. The header section in the architecture declares signals that represent `selsortproperty` and `selsort` variables as SEBAC inlines function calls. Signal `a` is declared as an `IntArray` which is a array of integers defined in a custom package we omitted for brevity. Signals `currentV` and `currentVN` are defined as logic vectors.

```
begin
  currentVN <= currentV when currentV = size else currentV + 1;
  currentV <= util_latch(currentVN);
  current <= util_type_conv(currentV);
```

Signal `currentVN` is the next state function of `currentV` and is connected to it through a latch function. The signal `current` is the only sequential iterator in the selection sort algorithm and plays the role of a program counter that schedules the execution of the algorithm. It is connected to `currentV` with a type conversion function which means that it is only a *wire* and no additional variables are needed for it.

```
  p1: process(current)
    variable lowestindex, j: integer range 0 to bound - 1;
  begin
  if (size /= 0 ) then
    if (current = 0 ) then
      aTest <= a(iTest);
    end if;
```

Function inlining. The VHDL process `p1` defines the variables needed in the inner loop of the selection sort algorithm. Note that `lowestindex` is declared as a variable and not as a signal since it does not carry information or state and is only used within process `p1` to compute a value. The check on Line 27 of `selsortproperty` that exits the program with a `return` statement is translated to the `if (size /= 0)` condition that encloses the process body. SEBAC inlines statements from both functions and thus stores in `aTest` the value of an arbitrary entry from the array indexed by `iTest` which is left uninitialized to denote a nondeterministic index.

Loops with dynamic bounds. SEBAC encloses the body of the loop from Line 5 in a conditional statement. The lowest index is computed at each execution step through the translation of the loop on Line 8.

```
    if (current < size - 1) then
      lowestindex := current;
      for j in 0 to bound - 1 loop
        if (j < current + 1) then
          next;
        end if;
        if (j < size) then
          if (a(j) < a(lowestindex)) then
            lowestindex := j;
          end if;
        end if;
      end loop;
      if (lowestindex /= current) then
        a(lowestindex) <= a(current);
        a(current) <= a(lowestindex);
      end if;
    end if;
```

Since SixthSense does not allow loops with dynamic bounds, we encode the ANSI-C `for` statement into a bounded VHDL `for` statement with two conditions. The first condition skips all array entries before `current + 1` through a `next` statement that is similar to the ANSI-C `continue` statement. The second condition looks at only entries of the array that are within its size. If the computed lowest index is different than `current` then a swap of values is enabled. Notice that `temp` is not needed since VHDL statements are executed concurrently.

Assertions. The assertions in `selsortproperty` should hold after the search is done. SEBAC encloses the assertions

with the conditional `current = size - 1` since the assertions are evaluated only when `selsort` completes its computation.

```
if (current = size - 1) then
  if( iTest < jTest) then
    assert (a(iTest) < a(jTest) or a(iTest) = a(jTest))
           report "order violated" severity error;
  end if;
  for i in 0 to bound - 1 loop
    if (i < size) then
      if( aTest = a(i)) then
        exit;
      end if;
    end if;
  end loop;
  assert (aTest = a(i))
         report "entry not found" severity error;
end if;
```

The indices `iTest` and `jTest` are left uninitialized to denote they are nondeterministic. The first assertion checks whether two array entries that are indexed by ordered indeces are actually ordered. The second checks whether `aTest`, the value of an arbitrary array entry that was stored before the sorting is still found in the array.

SEBAC currently does not handle recursion implied from branching (`goto` statements). It can be easily extended to unwind them with bounds in a fashion similar to CBMC.

We end up with a hierarchical VHDL design with asserted signals designated as the invariants. We pass the VHDL to SixthSense which, in case of satisfiability, provides a trace that satisfies the invariant, otherwise it returns with a proof.

5. RESULTS

To evaluate TBV, we compared SEBAC and SPARK coupled with SixthSense, (TBV:SEBAC) and (TBV:SPARK) respectively, against CBMC coupled with SAT (SAT:CBMC).

We took 4 challenging examples of C programs from Chapter 3 of [4]. All the three techniques were able to report helpful error traces within comparable and reasonable resources and small bounds (smaller than 8). We fixed the errors in the 4 programs (for `selsort` we changed Line 9 to `if (a[j] < a[lowestindex])`) and we report on the time that the analysis took to complete the proofs for a number of bounds in Table 3. The size column shows the number of variables and clauses in a CNF formula and the number of registers, inputs and AND gates (logically complete with inverters) in a sequential circuit as a measure of their complexity [27]. We report on bounds in powers of two since the analysis depends on bit vector encodings of variables and the bounds are on the ranges of these variables. We ran all experiments on a 1.7 GHz Pentium 4 machine with 1 GB memory and used CBMC version 2.5 and SPARK version 1.3.

By default, CBMC iteratively invokes the SAT solver with a larger formula that uses a bigger loop and recursion unwinding bound until no unwinding assertion is violated. The times reported for CBMC do not include the time needed for CBMC and SAT to perform these iterative checks. We provide CBMC through its command line interface with a tight bound on the loop unwinding to ensure a fair comparison.

5.1 Selection sort

The first example is the selection sort algorithm described in Section 4 with the bug on Line 9 fixed. We checked both the order of array entries and the data consistency assertions at the same time. CBMC was able to complete the check on the selection sort routine for a bound of 16 on the size of the array in 114 minutes. For bounds bigger than 16 CBMC could not complete before 6 hours. SixthSense was able to complete the check on the sequential circuit generated with SPARK for a bound of 64 in 132 minutes. For the same bound of 64, SixthSense took only 35 minutes to complete the proof on the sequential circuit generated with SEBAC.

5.2 Linked list insertion

The second example is a routine that inserts a node into an ordered linked list. List nodes are stored in an array and they are pointed to by indices. Each node has a key value and a next node index. The insert routine takes the index that points to the head of the list and an index that points to the new node to be inserted in the list and returns an index to the new head of the list. We check two properties of the list upon completion of the insertion assuming the properties held before the insertion.

Properties. The first property checks that the value of any node in the list is smaller than or equal to the value of its successor. The second property checks whether the size of the list is consistent (incremented by one after the insertion). CBMC was able to complete the analysis of the list insertion example for a bound of 16 within 164 minutes. However, it took more than 6 hours on larger bounds without completing the analysis. The sequential analysis completed on the sequential circuit generated with SPARK in 196 minutes. It took only 54 minutes to complete the proof on the circuit generated with SEBAC for the same bound of 64.

5.3 Linked list removal

The third example is a routine that removes a node from an ordered list. It takes as input the index to the head of the list, the key value of the node to be deleted, and a writable pointer that should be filled with the index of the deleted node. It returns the index of the new head of the list. We modeled the writable pointer with a static variable since SPARK does not handle pointers. We also relaxed the assertions CBMC generates for pointers to obtain a fair comparison.

Properties. The properties we checked for the removal routine are similar to those we checked for the insertion. We checked whether the order is preserved and whether the size of the list is consistent. It took CBMC 64 minutes to complete the proof with a bound of 16. CBMC generated a CNF formula for a bound of 32 but the analysis did not complete before 6 hours. We completed the proof on the sequential circuit generated with SPARK in 143 minutes for a bound of 64. Sequential analysis performed better with the circuit generated with SEBAC and completed in 74 minutes for the same bound.

5.4 Memory allocator and deallocator

Our last example is a memory allocator routine which takes as input a size of desired memory to allocate, returns an index into an array of bytes that represents memory, and signals failure by returning an invalid index (NULL). Internally memory is allocated in blocks of a constant size. The memory allocator considers consecutive groups of memory in the same allocation mode, either allocated or free, as *spans*. It tracks the usage of these blocks in another "in use" array. If a span is free, all its entries in the "in use" array contain a positive value which is the number of blocks in the span. If it is allocated, all entries in the "in use" array contain a neg-

Table 3: Comparison of SAT:CBMC, TBV:SPARK and TBV:SEBAC. In the size column, $|V|$ and $|C|$ denote the number of CNF variables and clauses respectively, and $|\mathcal{I}|$, $|\mathcal{R}|$, and |ANDs| denote the number of sequential circuit primary inputs, registers, and AND gates respectively. In the time column ∞ denotes a time-out of 360 minutes.

Selection sort			
Technique	Bound	Size	Time (minutes)
SAT:CBMC	16	$\|V\| = 4,753,354$; $\|C\| = 18,971,756$	114
	32	timed out before generating CNF	∞
TBV:SPARK	64	$\|\mathcal{I}\| = 392$; $\|\mathcal{R}\| = 552$; $\|\text{ANDs}\| = 30,334$	132
TBV:SEBAC	64	$\|\mathcal{I}\| = 12$; $\|\mathcal{R}\| = 397$; $\|\text{ANDs}\| = 92,320$	35
Linked list insertion			
Technique	Bound	Size	Time (minutes)
SAT:CBMC	16	$\|V\| = 159,370$; $\|C\| = 781,397$	64
	32	$\|V\| = 1,113,832$; $\|C\| = 6,402,757$	∞
TBV:SPARK	64	$\|\mathcal{I}\| = 611$; $\|\mathcal{R}\| = 984$; $\|\text{ANDs}\| = 22,711$	196
TBV:SEBAC	64	$\|\mathcal{I}\| = 18$; $\|\mathcal{R}\| = 622$; $\|\text{ANDs}\| = 53,636$	54
Linked list removal			
Technique	Bound	Size	Time (minutes)
SAT:CBMC	8	$\|V\| = 18,322$; $\|C\| = 107,422$	26
	16	timed out before generating CNF	∞
TBV:SPARK	64	$\|\mathcal{I}\| = 789$; $\|\mathcal{R}\| = 1240$; $\|\text{ANDs}\| = 27,505$	143
TBV:SEBAC	64	$\|\mathcal{I}\| = 24$; $\|\mathcal{R}\| = 792$; $\|\text{ANDs}\| = 71,893$	74
Memory allocator and deallocator			
Technique	Bound	Size	Time (minutes)
SAT:CBMC	16	$\|V\| = 1,411,745$; $\|C\| = 4,958,517$	93
	32	$\|V\| = 2,818,813$; $\|C\| = 9,912,211$	∞
TBV:SPARK	32	$\|\mathcal{I}\| = 1929$; $\|\mathcal{R}\| = 1084$; $\|\text{ANDs}\| = 44,802$	156
TBV:SEBAC	32	$\|\mathcal{I}\| = 59$; $\|\mathcal{R}\| = 792$; $\|\text{ANDs}\| = 103,433$	122

ative number which is the negative of the number of blocks in the span. We also consider a deallocation routine that takes an index to a memory location and frees it.

Properties. We assume that the memory and the "in use" array were initialized to be all free and that the indices passed to the deallocation routine are all aligned correctly with the block sizes. We run an arbitrary but bounded number of allocations and deallocations and then check the "in use" array for consistency with the sizes of allocation and deallocation operations we performed. CBMC completed the proof in 93 minutes for a bound of 16. It could not complete the proof for a bound of 32 in less than 6 hours. TBV:SPARK took 156 minutes to completed the proof for a bound of 32. TBV:SEBAC took 122 minutes to complete the proof for the same bound.

5.5 Discussion

The results in Table 3 show that by keeping the structure of the program and performing the analysis at the imperative sequential level, we were able to scale the analysis of ANSI-C programs to bounds that are much higher than those achievable by the stateless and flat hierarchy of SAT analysis.

TBV techniques were able to scale to the same bounds and found difficulty going beyond a bound of 64 for both SPARK and SEBAC. However, sequential circuits generated with SEBAC had a clear performance advantage against those generated by SPARK. We attribute the difference to the effort SixthSense needed to make in order to undo the scheduling logic and the additional state elements SPARK introduced to the program for synthesis optimizations and for ease of transformation purposes. Note that the sequential circuits generated by SPARK had more registers and inputs and less logic (AND gates) since SPARK aims at reducing the combinational depth of the circuit.

6. SUMMARY AND FUTURE WORK

In this paper we presented SEBAC, a novel static analysis technique for verifying imperative programs. We introduced the use of sequential circuits instead of pure combinational Boolean formulas to encode bounded ANSI-C programs and thus enabled the use of sequential solvers with reduction potentials that have no counterparts for combinational solvers. We were able to show that a bound of 64 is feasible with reasonable resources.

In the future we plan to explore how our approach of sequential encoding may be extended to other logic specifications such as S1S [2], and Presburger Arithmetic (PA) [30]. We are also interested in investigating security applications of PA program equivalence detection.

Acknowledgments.

The authors would like to thank Jason Baumgartner, Viresh Paruthi, Geert Janssen, Jessie Xu, Mark Williams, Hari Mony, Robert Kanzelman, and Ali El-Zein for contributions to the verification framework used in the experiments.

7. REFERENCES

[1] P. Ashenden. *The Designers Guide to VHDL*. Morgan Kaufmann, 2002.

[2] A. Aziz, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential Synthesis Using

S1S. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, Oct. 2000.

[3] A. Aziz, T. Shiple, V. Singhal, R. Brayton, and A. Sangiovanni-Vincentelli. Formula Dependent Equivalence for Compositional CTL Model Checking. *Journal of Formal Methods in System Design*, 2002.

[4] A. Barr. *Find The Bug: A Book of Incorrect Programs*. Addison-Wesley, 2002.

[5] J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In *Computer-Aided Verification*, July 2002.

[6] R. J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve real world SAT instances. In *National Conference on Artificial Intelligence*, 1997.

[7] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer-Aided Design*, November 2000.

[8] R. K. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. May 1982.

[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, March 2004.

[10] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Design Automation Conference*, 2003.

[11] G. De Micheli. Hardware synthesis from C/C++ models. In *Design Automation and Test in Europe*, Mar. 1999.

[12] S. A. Edwards. The challenges of hardware synthesis from C-like languages. In *Design Automation and Test in Europe*, 2005.

[13] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver. In *Design Automation and Test in Europe*, 2002.

[14] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, October 2004.

[15] S. Haynal. *Automata-Based Symbolic Scheduling*. PhD thesis, University of California, Santa Barbara, 2000.

[16] 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 *Int'l Conference on Computer-Aided Design*, Nov. 2000.

[17] G. Holzmann. The model checker SPIN. In *IEEE Transactions on Software Engineering*, May 1997.

[18] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[19] International Organization for Standardization. *ISO/IEC 9899:1999: Programming languages—C*. 1999.

[20] A. Kuehlmann and J. Baumgartner. Transformation-based verification using generalized retiming. In *Computer-Aided Verification*, July 2001.

[21] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design*, (12), 2002.

[22] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1993.

[23] H. Mony *et al.* Scalable automated verification via expert-system guided transformations. In *Formal Methods in Computer-Aided Design*, Nov. 04.

[24] R. K. Brayton *et al.* VIS: A system for verification and synthesis. In *Computer-Aided Verification*, July 1996.

[25] I.-H. Moon, G. D. Hachtel, and F. Somenzi. Border-block triangular form and conjunction schedule in image computation. In *Formal Methods in Computer-Aided Design*, Nov. 2000.

[26] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, June 2001.

[27] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In *Proceedings of the Int'l Conference on Computer Design*, Sept 2000.

[28] A. Saldanha, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Multi-Level Logic Simplification using Don't Cares and Filters. In *Design Automation Conference*, 1989.

[29] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical report, Electronics Research Lab, Univ. of California, Berkeley, May 1992.

[30] T. Shiple, J. Kukula, and R. Ranjan. A comparison of presburg engines for efsm reachability. In *Computer-Aided Verification*, June 1998.

[31] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems*, March 2007.

[32] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Automated Software Engineering Journal*, April 2003.

[33] F. Zaraket, A. Aziz, and S. Khurshid. Sequential circuits for relational analysis. In *International Conference on Software Engineering*, May 2007.

[34] F. Zaraket, J. Baumgartner, and A. Aziz. Scalable compositional minimization via static analysis. In *International Conference on Computer Aided Design*, Nov. 2005.