Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking*

Edmund Clarke
Carnegie Mellon University
Pittsburgh, PA 15213 USA
emc@cs.cmu.edu

Daniel Kroening
Carnegie Mellon University
Pittsburgh, PA 15213 USA
kroening@cs.cmu.edu

Karen Yorav
Carnegie Mellon University
Pittsburgh, PA 15213 USA
kareny@cs.cmu.edu

ABSTRACT

We present an algorithm that checks behavioral consistency between an ANSI-C program and a circuit given in Verilog using Bounded Model Checking. Both the circuit and the program are unwound and translated into a formula that represents behavioral consistency. The formula is then checked using a SAT solver. We are able to translate C programs that include side effects, pointers, dynamic memory allocation, and loops with conditions that cannot be evaluated statically. We describe experimental results on various reactive circuits and programs, including a small processor given in Verilog and its Instruction Set Architecture given in ANSI-C.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Register-Transfer-Level Implementation—*Design Aids*; F.3.1 [**Theory of Computation**]: Logics and Meanings of Programs—*Specifying and Verifying and Reasoning about Programs*

General Terms

Verification

Keywords

Verilog, ANSI-C, Equivalence Checking

*This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, DOD, ARO, or the U.S. government.

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

1. INTRODUCTION

When a new device is designed, a "golden model" is often written in a programming language like ANSI-C. This model is extensively simulated to insure both correct functionality and performance. Later, a Verilog implementation is created. It is essential to determine if the C and Verilog programs are consistent [1]. In general, ANSI-C programs are used as specifications in a variety of styles, using a language that designers are familiar with.

We automate the consistency test by using a formal verification technique called Bounded Model Checking (BMC) [2, 3]. In BMC, the transition relation for a system and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability using an efficient SAT procedure. If the formula is satisfiable, a counterexample is extracted from the output of the SAT procedure. Otherwise, the system and its specification are further unwound. This process terminates when the length of the potential counterexample exceeds the completeness threshold (i.e., is sufficiently long to ensure that no counterexample exists [4]) or when the SAT procedure exceeds its time or memory bounds. Our tool, called `CBMC`, takes as input a C program and a Verilog implementation. The two programs are unwound in tandem and converted to a Boolean formula that represents behavioral consistency. The formula is checked using Chaff [5]. If the two programs are inconsistent, a counterexample demonstrating this is generated, unless the tool exceeds its time or memory bounds.

The tool allows the user to customize the concept of "consistency". Besides simple consistency criteria, such as cycle accuracy or functional equivalence, other complex criteria can be realized. The value of each Verilog signal at every clock cycle is visible to the C program. This enables the programmer to monitor the behavior of the design, and assert correct values at certain points. Both cycle accurate and non cycle accurate specifications are easy to produce. Due to space limitations, we are not able to present this here. The interested reader may refer to [6].

Although converting Verilog code to a Boolean formula is relatively straightforward, ANSI-C programs are difficult to convert to Boolean formulas for many reasons including side effects and pointer usage. We give a procedure for this translation which addresses the subtleties of the language.

Related Work In [7], a tool for verifying the combinational equivalence of RTL-C and an HDL is described. They translate the C code into HDL and use standard equivalence checkers to establish the equivalence. The C code has to be very close to a hardware description (RTL level), which implies that the source and target have to be implemented in a very similar way. There are also variants of C specifically for this purpose, for example Spec C and Handel C, as well as the System C standard, which defines a subset of C++ that can be used for synthesis [8].

The concept of verifying the equivalence of a software implementation and a synchronous transition system was introduced by Pnueli, Siegel, and Shtrichman [9]. The C program is required to be in a very specific form, since a mechanical translation is assumed.

The methodology presented in [10] provides a large set of ANSI-C language features, including arbitrary loop constructs using unwinding assertions. However, it is limited to comparing the function that the program and circuit compute. This paper extends this methodology to allow fully reactive programs and circuits. This is illustrated using clearly reactive circuits such as a microprocessor. Furthermore, we present optimizations for nested loops and add support for pointer manipulation.

2. TRANSFORMING ANSI-C

We briefly show how to reduce the Model Checking Problem of C programs to determining the validity of a bit vector equation. A detailed explanation can be found in [6]. We pre-process the program into an equivalent program that uses only `while, if, goto`, and assignments. Next, all `while` loops are unwound using the following transformation n times:

`while (e) inst` $\rightarrow$ `if (e) { inst; while(e) inst}`

The last `while` loop is replaced by an assertion of $!e$, which assures that the program never performs more iterations. This *unwinding assertion* is verified along with the user defined assertions. If it fails we increase the number of iterations for this loop until the bound is big enough. Note that this bound is an upper bound and does not have to match the number of iterations.

The program now consists of only (nested) `if` instructions, assignments, assertions, and forward `goto` statements. It is next transformed into a bit-vector equation C that forms the set of constraints and a bit-vector equation P that represents the set of assertions. During this process, the program variables are renamed so that each (renamed) variable is assigned only once. Here is an example of the renaming function ρ:

```
if (b)                 if (b0)
  x = x + 1;             x1 = x0 + 1;
else                   else
  x = x + 2;             x2 = x1 + 2;
y = y + x;             y1 = y0 + x2;
x = y + 1;             x3 = y1 + 1;
```

$\xrightarrow{\rho}$

At this point forward `goto` statements are changed into equivalent `if` statements, as explained in [10].

The final transformation is done by the functions $\mathcal{C}(p, g)$, which computes the constraints (assumptions), and $\mathcal{P}(p, g)$, which computes the properties (assertions). Both take a program p and a guard g as argument and map this to an equation. Both are defined by a case split on p:

Skip. If p is empty or skip, both $\mathcal{C}$ and $\mathcal{P}$ are true.

$$\mathcal{C}(\texttt{skip}, g) := \mathsf{true} \quad \mathcal{P}(\texttt{skip}, g) := \mathsf{true}$$

Conditional. Let p be an `if` statement with condition c, and code blocks I and I'. Then: The functions are used recursively for both code blocks. For I, $\rho(c)$ is added to the guard, and for I', $\neg\rho(c)$ is added to the guard. The resulting constraints and claims are conjoined.

$$\mathcal{C}(\texttt{if(c) } I \texttt{ else } I', g) := \mathcal{C}(I, g \wedge \rho(c)) \wedge \mathcal{C}(I', g \wedge \neg\rho(c))$$
$$\mathcal{P}(\texttt{if(c) } I \texttt{ else } I', g) := \mathcal{P}(I, g \wedge \rho(c)) \wedge \mathcal{P}(I', g \wedge \neg\rho(c))$$

Sequential Composition. Let p be a sequential composition of I and I'. Then: As above, the functions are used recursively for both code blocks, but for this case with the same guard g.

$$\mathcal{C}(\texttt{I; I'}, g) := \mathcal{C}(I, g) \wedge \mathcal{C}(I', g)$$
$$\mathcal{P}(\texttt{I; I'}, g) := \mathcal{P}(I, g) \wedge \mathcal{P}(I', g)$$

```
while(B1) {
  S1;
  while(B2) {
   S2;
  }
}
```

$\longrightarrow$

```
while (vpc<=2) {
  switch (vpc) {
   case 1:
    if (B1)
     { S1; vpc = 2; }
    else
     vpc=3;
    break;
   case 2:
    if (B2) S2 else vpc=1;
    break; } }
```

Figure 2: Example of nested loop transformation

Assertion. Let p be an as assertion with argument a. The argument is renamed, guarded by g, and then returned as a property.

$$\mathcal{P}(\texttt{assert}(a), g) := g \Longrightarrow \rho(a)$$

$\mathcal{C}$ of an assertion is true.

Assignment $v = e$. The assignment is returned as an equality constraint, after renaming of variables. Let the value of the variable after the assignment be v_α. The value before the assignment is then $v_{\alpha-1}$. If v is a simple variable we add the following constraint: The value of v_α is equal to the renamed right hand side if the guard holds, and equal to $v_{\alpha-1}$ otherwise.

$$\mathcal{C}(v = e, g) := v_\alpha = g ? \rho(e) : v_{\alpha-1}$$

Note that the case split on g cannot be evaluated at translation time but is instead added as part of the constraint.

For $v[a] = e$ we add a constraint as follows: The new value of the array v_α at index i is equal to the renamed right hand side if the guard holds and i is equal to a, and equal to $v_{\alpha-1}[i]$ otherwise. We model arrays as functions and use lambda notation.

$$\mathcal{C}(v = e, g) := v_\alpha = \lambda i : (g \wedge i = \rho(a)) ? \rho(e) : v_{\alpha-1}[i]$$

If bounds checking is desired, we assert (by defining $\mathcal{P}$) that $\rho(a)$ is greater than or equal to zero and smaller than the number of elements of a. Assignment to variables with `struct` types are handled in a similar manner.

After computing $\mathcal{C}$ and $\mathcal{P}$ we verify that $\mathcal{C} \Longrightarrow \mathcal{P}$ is valid. This proves that no unwinding assertions have been violated and that all array bounds are obeyed. Figure 1 shows a simple example of the transformation process.

3. NESTED LOOPS

Nested loops within the ANSI-C code can result in extremely large CNF formulas since for every unwinding of an outer loop we unwind each inner loop in full. To alleviate this problem we transform nested loop constructs into equivalent single loops.

We partition the body of the nested loop construct into subprograms. We then add a virtual program counter variable that keeps track of which sub-program should be executed next. The result is a monolithic loop that executes the correct subprogram according to the virtual program counter. Figure 2 demonstrates this process.

The above transformation reduces the complexity of loop unwindings in cases where the number of times each sub-program is executed is bounded. This situation is extremely common when writing programs that specify synchronous hardware designs. Usually, such programs use an integer variable to index the arrays that contain signal values. The natural way of using this variable is to increment it within each loop in the program. Note that this variable can be incremented by any value, and of course one can always refer to any cycle at any point in the program. However, since the

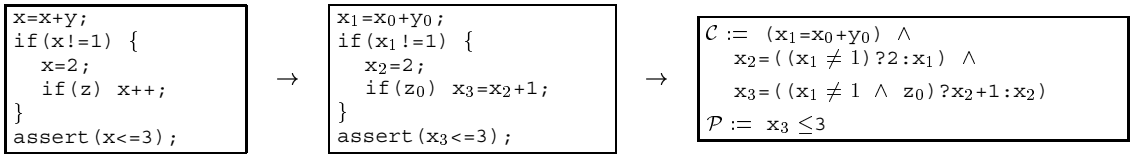

Figure 1: The first box on the left contains the unwound program with assertions. Each variable is a bit vector. The first step is to rename the variables. Then the program is transformed into a into bit vector equation as described in section 2.

C program is not allowed to access the value of a design signal at a clock cycle that is greater than CBMC_bound, we expect the programmer to insert a condition on the cycle variable of always being less than or equal to CBMC_bound. These observations lead us to expect a variable that is increased within every loop body and is checked to be below the bound in every loop condition. We perform our transformation only on such loops.

Let n be the bound of the Bounded Model Checking algorithm (CBMC_bound). As mentioned in Section 2, our tool will unwind each loop at least the maximum number of times that it might be iterated through. Assume we have a double-nested loop for which we perform our transformation. Under these conditions, the body of the inner loop will be replicated $O(n)$ times for the first unwinding of the outer loop, $O(n-1)$ times for the second unwinding of the outer loop, and so on. All together, the inner loop is replicated $O(n^2/2)$. However, after the transformation we have a single loop that is guaranteed not to need more than n unwindings.

4. POINTERS

Dereferencing Pointers Pointers are commonly used in ANSI-C programs, in particular for call by reference and for arrays. This applies even to programs that specify circuits.

During the unwinding phase, and before the variable renaming, all pointer dereferences are removed recursively as follows. First we simplify &*p expressions to p. This allows ANSI-C constructs such as p=&*NULL (it is guaranteed not to cause an exception).

In the second step, the remaining dereferencing operators are removed. Let e denote the sub-expression that is to be dereferenced. We remove dereferencing operators bottom-up, i.e., all sub-expressions of e are already free of dereferencing operators or other side effects. Let g denote the guard as described above, and o the offset. Dereferencing is done by a recursive function that is denoted by $\phi(e, g, o)$. The function maps a pointer expression to the dereferenced expression.

ANSI-C offers two dereferencing operators: The star operator and the array index operator. Both are replaced by the expression provided by ϕ. The star operator uses offset zero.

$$*e \longrightarrow \phi(e, g, 0) \qquad e[o] \longrightarrow \phi(e, g, o)$$

The pointer or array e has a type $*T$. This type T can be determined syntactically. The function ϕ is defined by a case split on e:

e **is a symbol of pointer type**. Let p be that pointer. The equation generated so far or the guard g must contain an equality of the form $\rho(p) = e'$ where e' is an expression. The pointer p is then dereferenced by applying ϕ to e'.

$$\phi(p, g, o) \quad := \quad \phi(e', g, o)$$

e **is a symbol of array type**. Let a be that array, i.e., $e = a$. We treat this case as syntactic sugar for $e = \&a[0]$.

e **is an "address of symbol"**, i.e., $e = \&s$ where s is a symbol. In this case, $\phi(e, g, o)$ is just s and we assert that the offset is zero. The variable is then renamed according to the rules above.

$$\phi(\&s, g, 0) \quad := \quad s$$

In addition to that, we check type consistency: the type of s has to match the type T (this can be determined syntactically). If T is a struct type and a prefix of the type of s, this is considered a match. In any other case, we generate an assertion that g is false.

e **is an "address of array element"**, i.e., $e = \&a[i]$, we add the offset to the index:

$$\phi(\&a[i], g, o) \quad := \quad a[i + o]$$

The array access is then done according to the rules above. As above, we check type consistency: the type of the array elements of the array a has to match the type T.

e **is a conditional expression**. The function ϕ is applied recursively for both cases. The condition c is added to the guard. The condition is free of side effects and pointer dereferences.

$$\phi(c?e' : e'', g, o) \quad := \quad c\,?\,\phi(e', g \wedge c, o) \,:\, \phi(e'', g \wedge \neg c, o)$$

e **is a pointer arithmetic expression**. A pointer arithmetic expression is a sum of a pointer and an integer . Let e' denote the pointer part, and i denote the integer part. The function ϕ is applied recursively to e', part while i is added to the offset.

$$\phi(e' + i, g, o) \quad := \quad \phi(e', g, o + i)$$

In order to prevent exposure of architecture properties, such as endianess, we assert that $*T$ matches the type of e'. This also prevents arithmetic on (void *) or incomplete type pointers.

e **is a pointer typecast**, i.e., $e = (Q*)e'$, where Q is an arbitrary type. The recursion proceeds with e'.

$$\phi((Q*)e', g, o) \quad := \quad \phi(e', g, o)$$

All other cases. For other cases the ANSI-C standard does not define semantics. For example, e might be the NULL pointer or a pointer variable that is uninitialized. We use an error value in this case and we assert that this dereferencing is never executed by the program. This is implemented by adding an assertion that $\rho(g)$ does not hold. The algorithm for the difference of two pointers $p - q$ is similar. We assert that p and q point to the same object, as required by the ANSI-C standard.

Consider the following example:

```
int a, b, *p;
if(x) p=&a; else p=&b;
*p=1;
```

The first statement is transformed into:

$$p_1 = (x_0\,?\,\&a : p_0) \quad \wedge \quad p_2 = (x_0\,?\,p_1 : \&b)$$

The variable p in the assignment is renamed to p_2. The star operator in the assignment statement is removed as follows:

$$\begin{array}{rcl}
*p & = & \phi(p, \mathsf{true}, 0) \\
& = & \phi(x_0?p_1 : \&b, \mathsf{true}, 0) \quad \text{because of } \rho(p) = p_2 \\
& = & x_0\,?\,\phi(p_1, x_0, 0) : \phi(\&b, \neg x_0, 0) \\
& = & x_0\,?\,\phi(x_0?\&a : p_0, x_0, 0) : b \\
& & \quad \text{because of } p_1 = (x_0\,?\,\&a : p_0) \\
& = & x_0\,?\,(x_0?\phi(\&a, x_0, 0) : \phi(p_0, x_0 \wedge \neg x_0, 0)) : b \\
& = & x_0\,?\,(x_0?a : \phi(p_0, x_0 \wedge \neg x_0, 0)) : b
\end{array}$$

This simplifies to $x?a : b$. After renaming, this is $x_0?a_0 : b_0$. This simplification is done by the program.

Dynamic Memory Allocation We allow programs that make use of dynamic memory allocation, e.g., for dynamically sized arrays or data structures such as lists or graphs. This is realized by adding two variables for each dynamic object: an active bit, and the object size. This allows bounds checks, and checking whether the object is accessed after its lifetime ended. The absence of memory holes can be verified by asserting that the active bits are false at program termination. We provide the full details in [6].

5. FINAL TRANSFORMATION

In order to create the final SAT instance we first need to unwind the Verilog circuit and transform it into a bit vector equation as well. We only consider the subset of the Verilog language known as the synchronous register transfer level (RTL). The process of translating the Verilog design closely resembles the process of synthesis of behavioral Verilog into a netlist.

The transition relation obtained from the Verilog file is then unwound. In contrast to the unwinding done for ANSI-C, the number of times the Verilog design is unwound must be specified manually (this is the `CBMC_bound` variable mentioned above).

We now have bit vector equations for both the ANSI-C program and the Verilog design. In order to compare them, we translate them into a SAT instance. For lack of space, the details of these transformations can not be included here.

6. EXPERIMENTS

We have run our tool on several examples. Following is a description of each example including running times. No optimization techniques, such as Bounded Cone of Influence, were applied.

Instruction Fetch Unit This example is the Instruction Fetch Module for the Torch Microprocessor, taken from [11]. The clock signals had to be identified manually. The specification implements the instruction fetch state machine and specifies a few invariants. For an unwinding bound of 100, the total runtime is 380s, using 1.4 GB of memory.

DRAM Arbiter This example is courtesy of Galileo Technology [12]. It features a rather small and yet non-trivial 4 to 1 arbiter that arbitrates between four client units requesting the services of one DRAM unit. For this example we created two different C specifications, the first uses a a list of invariants and the second gives a full (cycle accurate) functional specification. For sufficiently large bounds, we hit the memory limit of 2 GB.

PS/2 Interface The interface implements a serial bus standard that connects the keyboard and mouse to a PC. The Verilog keyboard controller has 67 latches and is about 700 lines. The ANSI-C code we wrote for it does not try to reproduce the behavior of the Verilog, but it nondeterministically picks a key and generates appropriate PS/2 clock and data signals, which are fed to the Verilog module. Thus, it plays the role of the keyboard, while the Verilog module is the computer. After sending the packet, it waits for the circuit to decode the packet and then compares the decoded key with the key that was sent. For an unwinding bound of 48, the total run time is 51 seconds.

DLX We compare a hardware and a software implementation of the DLX microprocessor, which is a load/store architecture with a RISC instruction set similar to the MIPS instruction set. The ANSI-C code implements the ISA only and is not a cycle-accurate simulation. In particular, it does not make use of a state machine but rather uses ANSI-C flow control to distinguish between individual instructions. It is approximately 550 lines of code. The hardware implementation contains a total of 1219 latches.

The software implementation is derived from a DLX simulator `dlxsim`. `CBMC` found a bug in the code that decodes the instruction word. This bug was not previously known. Using an unwinding bound of 5 cycles, `CBMC` generates a counterexample within 1 minute and 37 seconds. This shows how each implementation processes the instruction affected by the bug differently.

7. CONCLUSION AND FUTURE WORK

We have described the translation of ANSI-C programs and Verilog designs into a bit-vector equation using Bounded Model Checking techniques. This method allows us to specify the behavior of complex, reactive hardware designs using a well-known and easy to use sequential programming language like ANSI-C. The algorithm formally verifies behavioral consistency using an efficient SAT solver. We have performed experiments using hardware and software implementations of several reactive designs.

As future work, we intend to further optimize the generation of the SAT instance using specialized bit vector decision procedures and abstraction techniques.

8. REFERENCES

[1] C. Pixley. Formal verification of commercial integrated circuits. *IEEE Design & Test of Computers*, 18(4), 2001.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Yhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[3] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC'99)*, 1999.

[4] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In et al. Zuck, editor, *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *LNCS*, pages 298–309. Springer Verlag, 2003.

[5] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[6] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using Bounded Model Checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, School of Computer Science, 2003.

[7] L. Séméria, A. Seawright, R. Mehra, D. Ng, A. Ekanayake, and B. Pangrle. RTL C-based methodology for designing and verifying a multi-threaded processor. In *Proc. of the 39th Design Automation Conference*. ACM Press, 2002.

[8] http://www.systemc.org.

[9] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Int. Journal of Software Tools for Technology Transfer (STTT)*, 2(2):192–201, 1998.

[10] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, 2003.

[11] Texas 97 benchmarks, http://vlsi.colorado.edu/ vis/texas-97/.

[12] Galileo Technology, A Marvell Company, http://www.marvell.com.