Hardware Verification using ANSI-C Programs as a Reference*

Edmund Clarke

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213 USA
Tel: +1-412-268-2628
Fax: +1-412-621-5473
e-mail: emc@cs.cmu.edu

Daniel Kroening

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213 USA
Tel: +1-412-268-5409
Fax: +1-412-268-5576
e-mail kroening@cs.cmu.edu

ABSTRACT

We describe an algorithm to verify a hardware design given in Verilog using an ANSI-C program as a specification. We use SAT based Bounded Model Checking [1] in order to reduce the equivalence problem to a bit vector logic decision problem. As a case study, we describe experimental results on a hardware and a software implementation of the data encryption standard (DES) algorithm.

I. INTRODUCTION

A common hardware design approach employed by many companies is to first write a quick prototype that behaves like the planned circuit in a language like ANSI-C. This program is then used for extensive testing and debugging, in particular of any embedded software that will later on be shipped with the circuit. An example is the hardware of a cell phone and its software. After testing and debugging of the program, the actual hardware design is written using hardware description languages like VHDL or Verilog.

Thus, there are two implementations of the same design: one written in ANSI-C, which is written for simulation, and one written in register transfer level HDL, which is the actual product. The ANSI-C implementation is usually thoroughly tested and debugged.

Due to market constraints, companies aim to sell the chip as soon as possible, i.e., shortly after the HDL implementation is designed. There is usually little time for additional debugging and testing of the HDL implementation. Thus, an automated, or nearly automated way of establishing the consistency of the HDL implementation is highly desirable.

This motivates the verification problem: we want to verify the consistency of the HDL implementation, i.e., the product, using the ANSI-C implementation as a reference [2]. Establishing the consistency does not require a formal specification. However, formal methods to verify either the hardware or software design are still desirable.

Related Work There have been several attempts in the past to tackle the problem. In [3], a tool for verifying the combinatorial 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. The System C standard, among others, defines a subset of C++ that can be used for synthesis [4].

The previous work focuses on a small subset of ANSI-C that is particularly close to register transfer language. Thus, the designer is often required to rewrite the C program manually in order to comply with these constraints. We extend the methodology to handle the full set of ANSI-C language features. This is a challenge in the presence of complex, dynamic data structures and pointers that may dynamically point to multiple objects. However, the approach is currently limited to functional equivalence, i.e., the function computed by the circuit and the program is compared. Reactive systems are not supported.

SAT based Bounded Model Checking (BMC) [1, 5] was introduced several years ago as a complementary technique for the more traditional BDD-based symbolic model checking. The basic idea of BMC is to search for a counterexample in traces whose length is bounded by some integer n. If no bug is found then the bound n is increased until either a bug is found, the problem becomes intractable, or some upper bound is reached. The BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved by standard SAT methods rather than BDDs.

Outline In section II, we describe how the ANSI-C program is transformed into a bit vector equation. In section III, we present the algorithm that handles pointers. Section IV contains the algorithm for the transformation of the Verilog design into a bit vector equation, and section V shows the transforma-

*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. 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.

tion into a SAT instance and we report how the technique is applied to two implementations of DES.

II. Transforming ANSI-C

A. Preparing the Translation

This section describes how we formalize the semantics of the ANSI-C language and reduce the Model Checking Problem to determining the validity of a bit vector equation. We assume that the ANSI-C program is already preprocessed. The program is then prepared for translation:

1. The instructions `break`, `continue`, and `return` are replaced by semantically equivalent goto instructions as described in the ANSI-C standard [6]. The `switch` and `case` instructions are replaced by semantically equivalent code using `if` and `goto` instructions.
2. The `for` and `do while` instructions are replaced by equivalent `while` instructions.
3. Side effects, i.e., post- and pre-increment operators, and function calls, are removed by introducing new temporary variables (section III describes how pointer dereferences are handled). For example,

   ```
   x=5+(i++);
   ```

   is transformed to

   ```
   tmp=i; i=i+1; x=5+tmp;
   ```

   where `tmp` is a new variable of the same type as `i`. In case of function calls, appropriate variable renaming is applied to preserve locality. As the ANSI-C standard allows multiple evaluation orderings, all allowed orderings have to be verified.

B. Unwinding the Program

After the preparation phase, loop constructs are unwound. Loop constructs can be expressed using `while` statements, (recursive) function calls, and `goto` statements. The `while` loops are unwound using the following transformation n times:

$$\texttt{while(}e\texttt{)}\ \textit{instr}$$
$$\longrightarrow \texttt{if(}e\texttt{)}\ \{\ \textit{instr}\texttt{;}\ \texttt{while(}e\texttt{)}\ \textit{instr}\ \}$$

The `if` statement is added for premature termination of the execution of the loop body, since the actual number of iterations can depend on the inputs. Any statements generated for side effects in e are copied as well. The remaining `while` loop is replaced by an assertion that assures that the program never does more iterations. This assertion is called an *unwinding assertion*.

$$\texttt{while(}e\texttt{)}\ \textit{instr} \longrightarrow \texttt{assert(!}e\texttt{);}$$

These unwinding assertions are a crucial part of our approach in order to assert that the unwinding bound is actually great enough. We formally verify that the assertion holds. If the assertion fails for any possible execution, then we increase the number of iterations for the particular loop until the bound is big enough. Note that this bound just has to be an upper bound and does not have to match the number of iterations exactly.

Function calls are expanded. Recursive function calls and backward `goto` statements are unwound in a manner similar to `while` loops.

C. Final Transformation with Variable Renaming

The program resulting from the preceding steps only consists of (nested) `if` instructions, assignments, assertions, labels, and `goto` instructions with branch targets that are defined after the `goto` instruction (forward jumps). It is now transformed into a bit-vector equation C that forms the set of constraints and into a bit-vector equation P that represents the set of properties, i.e., the assertions. During this process, the variables are renamed.

Let the program refer to variable v at a given program location. Let α denote the number of assignments made to variable v prior to the location. The variable v is then renamed to v_α. Within assignments to variable v, the expression on the right hand side is considered to be before the assignment. The variable that is assigned to on left the hand side is considered to be after the assignment. Let e denote an expression. Then $\rho(e)$ denotes the expression after renaming.

This transformation is done iteratively as follows: We start with an empty set of constraints and properties, i.e., $C = true$ and $P = true$. Let C and P denote the equations before one step and C' and P' denote the equations after the step. The algorithm terminates if the program is empty. If not so, let p denote the remainder of the program. Let c, p' be parts of p, and I be a single statement such that the concatenation of c, I, and p' is p. Furthermore, let c contain only `if` statements, and I any other statement.

$$p \quad = \quad c\, I\, p'$$

Thus, I is the first statement of the remaining program p that is not an `if` statement. Let n denote the number of `if` statements in c. Let $e_1, \ldots, e_n$ denote the expressions that are conditions of the `if` statements in c.

Let g denote the conjunction of the conditions of the `if` statements. In case there is no `if` statement, i.e., $n = 0$, then g is true. The conjunction g is called *guard* of I.

$$g \quad := \quad \begin{cases} \bigwedge_{i=1}^{n} e_i & : \quad n \geq 1 \\ \text{true} & : \quad \text{otherwise} \end{cases}$$

The algorithm proceeds by a case split on *I*.

1. Let I be a `goto` statement. The `goto` statement is removed. The target label l of the `goto` instruction must be after the `goto`, i.e., within p'. Let x denote the part of p' before the label and y denote the part of p' after the label l:

   $$p' \quad = \quad x\, l : y$$

An `if` statement is added as guard to all statements in x. The condition of the `if` statement is $!g$. The transformations does not work for `goto` instructions that jump inside a guarded block.

2. Let I be an assertion, i.e., `assert(`a`)`. In this case, the assertion statement is removed, the assertion a is renamed and added as a constraint to the bit vector equation P. Since the assertion is only executed if g holds, then the renamed guard $\rho(g)$ must imply the renamed assertion $\rho(a)$. Remember that $\rho(e)$ denotes the expression e after renaming.

$$P' \quad := \quad P \wedge (\rho(g) \Longrightarrow \rho(a))$$

3. Let I be the skip statement. If the "then" block of the last `if` statement in c contains other instructions, the skip statement is just removed. If not so, and the last `if` statement of c has an `else` part, the condition of the last `if` statement is negated and the "else" block becomes the "then" block. If the `if` statement does not have an `else` part, the last `if` statement in c is replaced by a skip statement.

4. Let I be an assignment. Let v be the variable on the left hand side, let e be the expression on the right hand side.

   Let the value of the variable after the assignment be v_α. The value before the assignment then is $v_{\alpha-1}$. If v is a simple variable, i.e., not of an array or struct type, i.e., the assignment is $v = e$, we add the following constraint: The new value of the variable v_α has to be equal to the renamed right hand side if the guard holds, and equal to the old value of v otherwise.

$$C' \quad := \quad C \wedge v_\alpha = \begin{cases} \rho(e) & : \quad \rho(g) \\ v_{\alpha-1} & : \quad \text{otherwise} \end{cases}$$

   If v is of an array type, let a be the array index address, i.e., the assignment is $v[a] = e$. The new value of the array v_α at index i has to be equal to the renamed right hand side if the guard holds and if i is equal to a, and equal to the old value of $v[i]$ otherwise.

$$C' \quad := \quad C \wedge v_\alpha = \lambda i : \begin{cases} \rho(e) & : \quad \rho(g) \wedge i = \rho(a) \\ v_{\alpha-1}[i] & : \quad \text{otherwise} \end{cases}$$

   We add an assertion that $\rho(a)$ is greater or equal zero and that it is smaller than the number of elements of a. Assignment to variables with struct types are handled similar to assignments to variables with array type.

After the computation of C and P using the algorithm above, we verify that

$$C \quad \Longrightarrow \quad P$$

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.

III. POINTERS

All pointer dereferences are removed recursively as follows: 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, o the offset. The 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 the star operator and the array index operator. Both are replaced by the expression provided by ϕ. The star operator uses offset zero.

$$\begin{aligned} *e \quad &\longrightarrow \quad \phi(e, g, 0) \\ e[o] \quad &\longrightarrow \quad \phi(e, g, o) \end{aligned}$$

The function ϕ is defined using a case split on e:

1. Let e be 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 arbitrary expression. The pointer p is then dereferenced by dereferencing e'. Otherwise, proceed as in case 7.

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

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

3. Let e 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.

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

4. If e is an address of array element expression, 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.

5. Let e be 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.

$$\phi(c?e' : e'', g, o) := c \,?\, \phi(e', g \wedge c, o) \;:\; \phi(e'', g \wedge \overline{c}, o)$$

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

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

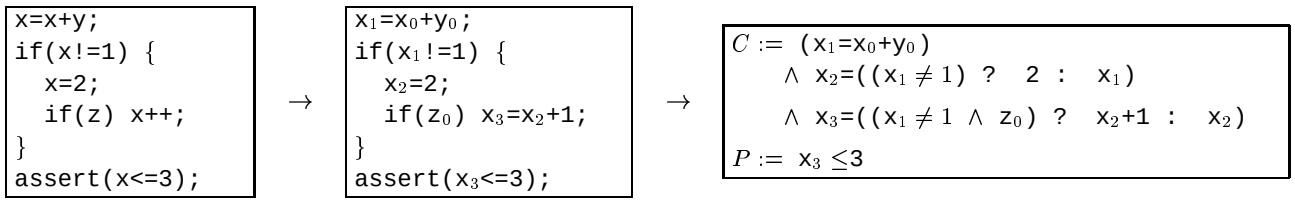

Fig. 1. Example: Renaming and transformation. The first box on the left contains the unwound program with assertions. Each variable is a bit vector. The first step is to rename the variables. Then the program is transformed into a into bit vector equation as described in section C.

7. In any other case, the ANSI-C standard does not define semantics (e might be the `NULL` or a pointer variable that is uninitialized). We use a free variable in this case and we assert that this dereferencing is never done 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.

Example: Consider the code fragment

```
int a, *p; p=&a;
if(x) p=NULL;
if(p!=NULL && *p==1);
```

The first two statements are transformed into:

$$p_1 = \&a \quad \wedge \quad p_2 = (x_0?\texttt{NULL} : p_1)$$

The star operator in the `if` statement is removed as follows:

$$\begin{aligned} *p &= \phi(p, p \neq \texttt{NULL}, 0) \\ &= \phi(x_0?p_1 : \texttt{NULL}, p \neq \texttt{NULL}, 0) \\ &= x_0?\phi(p_1, p \neq \texttt{NULL} \wedge x_0, 0) : \\ &\qquad \phi(\texttt{NULL}, p \neq \texttt{NULL} \wedge \overline{x_0}, 0) \\ &= x_0?\phi(\&a, p \neq \texttt{NULL} \wedge x_0, 0) : v \\ &= x_0?a : v \end{aligned}$$

It is asserted that $p_2 \neq \texttt{NULL} \wedge \overline{x_0}$ does not hold.

The algorithm permits dynamic memory allocation, which we omit due to lack of space.

IV. Transforming Verilog

We only consider a very restricted subset of the Verilog language. Delay or event specifiers are ignored and only register data transfers are converted. Such a language is called synchronous register transfer language (RTL). The process of translating verilog into a bit vector equation closely matches the synthesis process. As result from synthesis, we obtain a transition relation and an initial state predicate. This transition relation is then unwound as usually done by a bounded model checker. In contrast to the unwinding done for ANSI-C, the number of times the unwinding must be specified manually.

V. SAT Instance Generation and Experiments

The bit vector equations obtained from the ANSI-C program and the Verilog circuit are translated into CNF by generating circuits for bitwise operators such as equality, shifting, and multiplication. Due to lack of space, we have to omit the details of the simplification and translation process.

As a case study, we are using a software and hardware implementation of the DES encryption standard. The software implementation is used in most Unix systems. On a 1.5 GHZ machine the translation of the program into a SAT instance takes 59 seconds. The SAT checker Chaff [7] detects it to be unsatisfiable within seconds. The hardware implementation is written in synchronous Verilog, is cost optimized, and about 1900 lines long. It requires 16 unwinding steps.

VI. Conclusion and Future Work

We have described the translation of ANSI-C programs into a SAT instance using Bounded Model Checking. We have performed experiments using a hardware and software implementation of the DES algorithm.

We plan to add support for reactive models and to optimize the generation of the SAT instance using specialized bit vector decision procedures and abstraction techniques.

References

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

[2] C. Pixley. Guest Editor's Introduction: Formal Verification of Commercial Integrated Circuits. *IEEE Design & Test of Computers*, 18(4):4–5, 2001.

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

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

[5] 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.

[6] International Organization for Standardization. *ISO/IEC 9899:1999: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 1999.

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