A Tool for Checking ANSI-C Programs

Edmund Clarke, Daniel Kroening, and Flavio Lerda

Carnegie Mellon University

Abstract. We present a tool for the formal verification of ANSI-C programs using Bounded Model Checking (BMC). The emphasis is on usability: the tool supports almost all ANSI-C language features, including pointer constructs, dynamic memory allocation, recursion, and the `float` and `double` data types. From the perspective of the user, the verification is highly automated: the only input required is the BMC bound. The tool is integrated into a graphical user interface. This is essential for presenting long counterexample traces: the tool allows stepping through the trace in the same way a debugger allows stepping through a program.

1 Introduction

We present a tool that uses Bounded Model Checking to reason about low-level ANSI-C programs. There are two applications of the tool: 1) the tool checks safety properties such as the correctness of pointer constructs, and 2) the tool can compare an ANSI-C program with another design, such as a circuit given in Verilog.

Many safety-critical software systems are legacy designs, i.e., written in a low level language such as ANSI-C or even assembly language, or at least contain components that are written in this manner. Furthermore, very often performance requirements enforce the use of these languages. These systems are a bigger security and safety problem than programs written in high level languages. The high level languages are usually easier to verify, as they enforce type-safety, for example. The verification of low level ANSI-C code is challenging due to the extensive use of arithmetic, pointers, pointer arithmetic, and bit-wise operators.

We describe a tool that formally verifies ANSI-C programs. The properties checked include pointer safety, array bounds, and user-provided assertions. The tool implements a technique called Bounded Model Checking (BMC) [1]. In BMC, the transition relation for a complex state machine and its specification are jointly unwound to obtain a Boolean formula that is satisfiable if there exists an error trace. The formula is then checked by using a SAT procedure. If the formula is satisfiable, a counterexample is extracted from the output of the SAT procedure. The tool checks that sufficient unwinding is done to ensure that no longer counterexample can exist by means of *unwinding assertions*.

The tool comes with a graphical user interface (GUI) that hides the implementation details from the user. It resembles tools already well-known to software engineers. If a counterexample is found, the GUI allows stepping through the trace like a debugger. We hope to make formal verification tools accessible to non-expert users this way.

Hardware Verification using ANSI-C as a Reference. 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. After testing and debugging the program, the actual hardware design is written using hardware description languages like Verilog. The Verilog description is then synthesized into a circuit.

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 with respect to the ANSI-C model 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.

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. Our tool supports the full set of ANSI-C language features, which makes it easier to use software written for simulation and testing as a reference model. Details of the various programming styles permitted by our tool are described in [3]. A short version is in [4].

In order to verify the consistency of the two implementations, we unwind both the C program and the circuit in tandem. The unwinding of the circuit is done as conventionally done by any Bounded Model Checker.

2 Bounded Model Checking for ANSI-C Programs

2.1 Generating the Formula

We reduce the Model Checking Problem to determining the validity of a bit vector equation. The full details of the transformation are described in [3]. The process has five steps:

1. We assume that the ANSI-C program is already preprocessed, e.g., all the `#define` directives are expanded. We then replace side effects by equivalent assignments using auxiliary variables, `break` and `continue` by equivalent `goto` statements, and `for` and `do while` loops by equivalent `while` loops.
2. The loop constructs are unwound. Loop constructs can be expressed using `while` statements, (recursive) function calls, and `goto` statements. The

`while` loops are unwound by duplicating the loop body n times. Each copy is guarded using an `if` statement that uses the same condition as the loop statement. The `if` statement is added for the case that the loop requires less than n iterations. After the last copy, an assertion is added that assures that the program never requires more iterations. The assertion uses the negated loop condition. We call this assertion an *unwinding assertion.*
These unwinding assertions are crucial for our approach: they assert that the unwinding bound is actually large enough. If the unwinding assertion of a loop fails for any possible execution, then we increase the bound n for that particular loop until the bound is large enough.
3. Backward `goto` statements are unwound in a manner similar to `while` loops.
4. Function calls are expanded. Recursive function calls are handled in a manner similar to while loops: the recursion is unwound up to a bound. It is then asserted that the recursion never goes deeper. The `return` statement is replaced by an assignment (if the function returns a value) and a `goto` statement to the end of the function.
5. The program resulting from the preceding steps only consists of (possibly nested) `if` instructions, assignments, assertions, labels, and `goto` instructions with branch targets that are defined after the `goto` instruction (forward jumps). This program is then transformed into static single assignment (SSA) form, which requires a pointer analysis. We omit the full details of this process. Here is a simple example of the transformation:

```
x=x+y;
if(x!=1)
  x=2;
else
  x++;

assert(x<=3);
```

$\rightarrow$

$$\begin{array}{l}
\texttt{x}_1\texttt{=x}_0\texttt{+y}_0\texttt{;}\\
\texttt{if(x}_1\texttt{!=1)}\\
\quad\texttt{x}_2\texttt{=2;}\\
\texttt{else}\\
\quad\texttt{x}_3\texttt{=x}_1\texttt{+1;}\\
\\
\texttt{x}_4\texttt{=(x}_1\texttt{!=1)?x}_2\texttt{:x}_3\texttt{;}\\
\texttt{assert(x}_4\texttt{<=3);}
\end{array}$$

$\rightarrow$

$$\begin{array}{ll}
C := & \texttt{x}_1\texttt{=x}_0\texttt{+y}_0 \ \wedge\\
& \texttt{x}_2\texttt{=2} \ \wedge\\
& \texttt{x}_3\texttt{=x}_1\texttt{+1} \ \wedge\\
& \texttt{x}_4\texttt{=(x}_1\texttt{!=1)?x}_2\texttt{:x}_3\\
P := & \texttt{x}_4 \leq \texttt{3}
\end{array}$$

The procedure above produces two bit-vector equations: C (for the constraints) and P (for the property). In order to check the property, we convert $C \wedge \neg P$ into CNF by adding intermediate variables and pass it to a SAT solver such as Chaff [5]. If the equation is satisfiable, we found a violation of the property. If it is unsatisfiable, the property holds.

2.2 Converting the Formula to CNF

The conversion of most operators into CNF is straight-forward, and resembles the generation of appropriate arithmetic circuits. The tool can also output the bit-vector equation before it is flattened down to CNF, for the benefit of circuit-level SAT solvers.

`CBMC` allows programs that make use of dynamic memory allocation, e.g., for dynamically sized arrays or data structures such as lists or graphs. As an example, the following fragment allocates a variable number of integers using `malloc`, writes one value into the last array element, and then deallocates the array:

```
void f(unsigned int n) {
  int *p;

  p=malloc(sizeof(int)*n);

  p[n-1]=0;

  free(p);
}
```

While the integer n is still bounded, its maximum value requires to reserve far too many literals in order to build a CNF for the fragment above. Thus, dynamically allocated arrays are not translated into CNF by allocating literals for each potential array element. Instead, arrays with variable size are implemented by means of uninterpreted functions.

3 A Graphical User Interface

The command line version of the tool cannot be used easily by its intended users, i.e., system designers, software engineers and programmers. Such users are not likely to have a deep knowledge of formal verification tools. Therefore, to increase the usability of our tool, we have designed a user interface meant to be more familiar.

The tool has two main possible applications: the verification of properties of C programs and checking consistency of Verilog designs against a C implementation. The former is mostly addressed to software engineers and programmers, and the latter is mostly meant for hardware designers.

In order to make the tool appealing to software designers and programmers, we organized the interface in a way similar to an IDE (Integrated Development Environment). The main window allows accessing source files. Source files can be organized into projects, for which a set of options is maintained. The options allow the user to configure the parameters that are passed to CBMC on the command line. If CBMC generates an error trace, the "Watches" window, which contains the current values of the program variables, is opened. At the beginning, this window will show the initial values of the variables. Then, it is possible to step forward in the trace: the line that is "executed" is highlighted as the user steps through the trace, and the values of the variables in the Watches window are updated. This is done similarly to the way a programmer steps through a program execution during debugging. The main difference is that the trace corresponds to an erroneous execution, while during debugging this is not necessarily the case. Moreover, we allow stepping through a trace not only forward, in the way a debugger usually does, but also backward, giving the user the ability to determine more easily the causes of the error.

When trying to verify the consistency of a Verilog design and a C program, it is still possible to step through the trace generated for the C program as before, but this is not true for the Verilog design. Moreover, hardware designer are used to tools like simulators, which display waveform diagrams. In order to make our tool more suitable to hardware designers, the interface displays the traces that have been generated for a Verilog design using a waveform diagram. Therefore, while stepping through the C program, it is possible to analyze the waveform corresponding to the signals in the Verilog design.

4 Conclusion and Future Work

We described a tool that formally verifies ANSI-C programs using Bounded Model Checking (BMC). The tool supports all ANSI-C operators and pointer constructs allowed by the ANSI-C standard, including dynamic memory allocation, pointer arithmetic, and pointer type casts. The user interface is meant to appeal to system designers, software engineers, programmers and hardware designers, offering an interface that resembles the interface of tools that the users are familiar with.

When a counterexample is generated, the line number reported by CBMC is usually not pointing to the line that contains the actual bug. A version of CBMC modified by Alex Groce addresses the problem of error localization [6]: the tool displays which statements or input values are important for the fact that the property is violated.

References

1. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.
2. Carl Pixley. Guest Editor's Introduction: Formal Verification of Commercial Integrated Circuits. *IEEE Design & Test of Computers*, 18(4):4–5, 2001.
3. 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.
4. Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.
5. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
6. Alex Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

Appendix

A Availability

The tool is available at:

`http://www.cs.cmu.edu/~modelcheck/cbmc/`

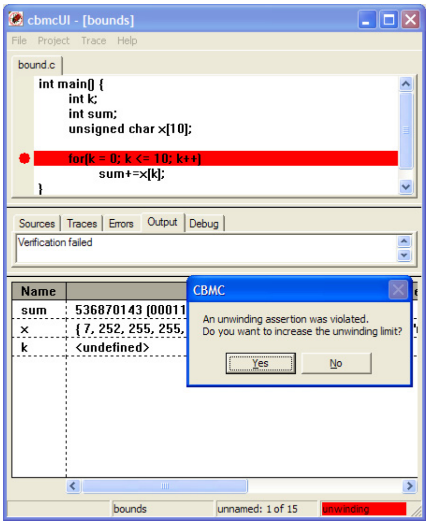
The web-page provides binaries for Linux, Windows, and Solaris that are ready to install. It also offers a detailed manual with an introductory tutorial.

B ANSI-C Language Features

The tables 1 and 2 summarize the supported ANSI-C language features and the properties that are checked automatically. A more detailed description with examples is available in the manual.

Table 1. Supported language features and implicit properties

Supported Language Features		Properties checked
Basic Data Types	All scalar data types `float` and `double` using fixed-point arithmetic. The bit-width can be adjusted using a command line option.	
Integer Operators	All integer operators, including division and bit-wise operators Only the basic floating-point operators	Division by zero Overflow for signed data types
Type casts	All type casts, including conversion between integer and floating-point types	Overflow for signed data types
Side effects	`CBMC` allows all compound operators	Side effects are checked not to affect variables that are evaluated elsewhere, and thus, that the ordering of evaluation does not affect the result.
Function calls	Supported by inlining. The locality of parameters and non-static local variables is preserved.	1. Unwinding bound for recursive functions 2. Functions with a non-void return type must return a value by means of the return statement.
Control flow statements	`goto`, `return`, `break`, `continue`, `switch` ("fall-through" is not supported)	
Non-Determinism	User-input is modeled by means of non-deterministic choice functions	
Assumptions and Assertions	Only standard ANSI-C expressions are allowed as assertions.	Assertions are verified to be true for all possible non-deterministic choices given that any assumption executed prior to the assertion is true.
Arrays	Multi-dimensional arrays and dynamically-sized arrays are supported	Lower and upper bound of arrays, even for arrays with dynamic size



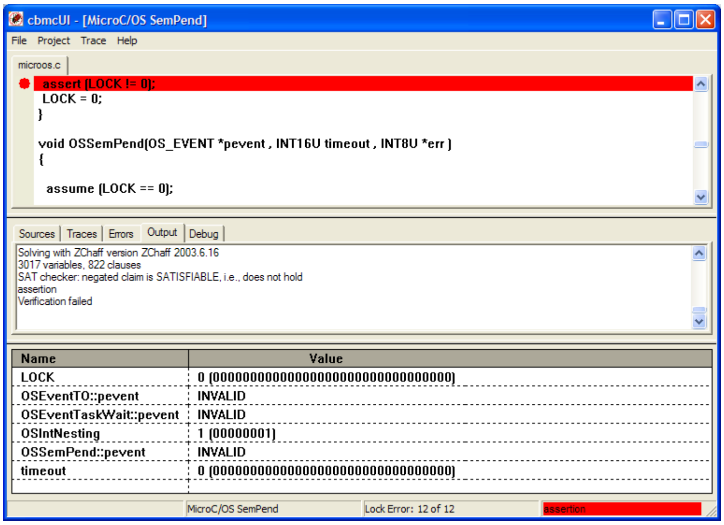
```
int main() {
        int k;
        int sum;
        unsigned char x[10];

        for(k = 0; k <= 10; k++)
              sum+=x[k];
}
```

```
Verification failed
```

Name	
sum	536870143 (00011
x	{ 7, 252, 255, 255,
k	<undefined>

An unwinding assertion was violated.
Do you want to increase the unwinding limit?

Fig. 1. The tool is able to automatically check the bound selected by the user for the unwinding of loops. If the given bound is not sufficient, the tool suggests to provide a larger bound.

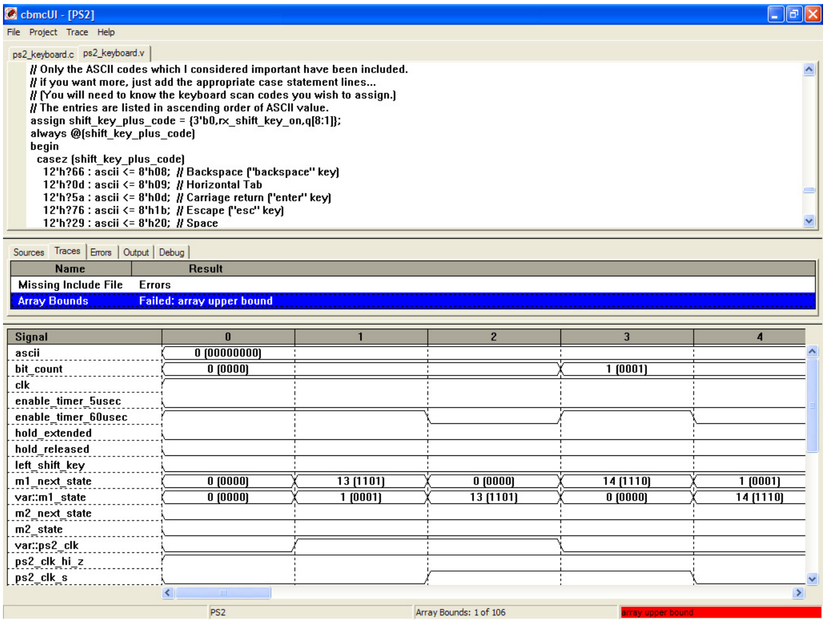


```
 assert (LOCK != 0);
 LOCK = 0;
}

void OSSemPend(OS_EVENT *pevent , INT16U timeout , INT8U *err )
{

  assume (LOCK == 0);
```

```
Solving with ZChaff version ZChaff 2003.6.16
3017 variables, 822 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
assertion
Verification failed
```

Name	Value
LOCK	0 (00000000000000000000000000000000)
OSEventTO::pevent	INVALID
OSEventTaskWait::pevent	INVALID
OSIntNesting	1 (00000001)
OSSemPend::pevent	INVALID
timeout	0 (00000000000000000000000000000000)

Fig. 2. The Watches windows allows keeping track of the current values of the program variables. In this case, the assertion failed because the variable LOCK has value 0.



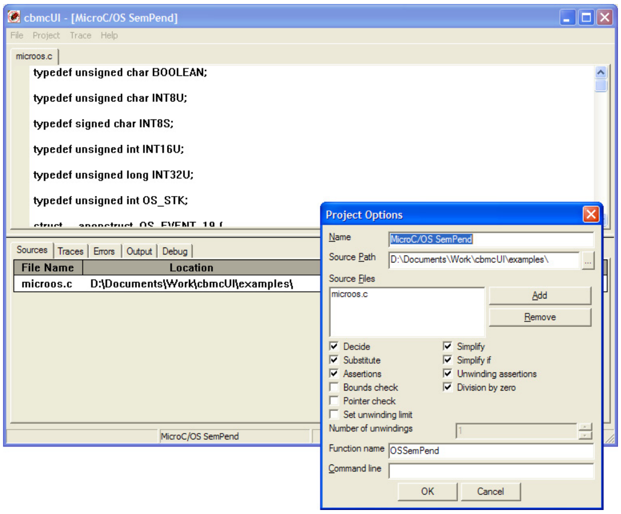
```
// Only the ASCII codes which I considered important have been included.
// if you want more, just add the appropriate case statement lines...
// (You will need to know the keyboard scan codes you wish to assign.)
// The entries are listed in ascending order of ASCII value.
assign shift_key_plus_code = {3'b0,rx_shift_key_on,q[8:1]};
always @(shift_key_plus_code)
begin
  casez (shift_key_plus_code)
    12'h?66 : ascii <= 8'h08;  // Backspace ("backspace" key)
    12'h?0d : ascii <= 8'h09;  // Horizontal Tab
    12'h?5a : ascii <= 8'h0d;  // Carriage return ("enter" key)
    12'h?76 : ascii <= 8'h1b;  // Escape ("esc" key)
    12'h?29 : ascii <= 8'h20;  // Space
```

Name	Result
Missing Include File	Errors
Array Bounds	Failed: array upper bound

Signal	0	1	2	3	4
ascii	0 (00000000)				
bit_count	0 (0000)			1 (0001)	
clk					
enable_timer_5usec					
enable_timer_60usec					
hold_extended					
hold_released					
left_shift_key					
m1_next_state	0 (0000)	13 (1101)	0 (0000)	14 (1110)	1 (0001)
var::m1_state	0 (0000)	1 (0001)	13 (1101)	0 (0000)	14 (1110)
m2_next_state					
m2_state					
var::ps2_clk					
ps2_clk_hi_z					
ps2_clk_s					

PS2 | Array Bounds: 1 of 106 | array upper bound

Fig. 3. The Signals window shows the values of the variables in a Verilog design using a waveform representation.



```
typedef unsigned char BOOLEAN;

typedef unsigned char INT8U;

typedef signed char INT8S;

typedef unsigned int INT16U;

typedef unsigned long INT32U;

typedef unsigned int OS_STK;

struct anonstruct_OS_EVENT_19 {
```

File Name	Location
microos.c	D:\Documents\Work\cbmcUI\examples\

MicroC/OS SemPend

Name: MicroC/OS SemPend
Source Path: D:\Documents\Work\cbmcUI\examples\
Source Files: microos.c

- [✓] Decide
- [✓] Substitute
- [✓] Assertions
- [ ] Bounds check
- [ ] Pointer check
- [ ] Set unwinding limit
- [✓] Simplify
- [✓] Simplify if
- [✓] Unwinding assertions
- [✓] Division by zero

Number of unwindings: 1
Function name: OSSemPend
Command line:

Fig. 4. The Project Options dialog allows setting up the parameters.

Table 2. Supported language features and implicit properties

Supported Language Features		Properties checked
Structures	Arbitrary, nested structure types; may be recursive by means of pointers; incomplete arrays as last element of structure are allowed	
Unions	Support for named unions, anonymous union members are currently not supported	`CBMC` checks that unions are not used for type conversion, i.e., that the member used for reading is the same as used for writing last time.
Pointers	Dereferencing	When a pointer is dereferenced, `CBMC` checks that the object pointed to is still alive and of matching type. If the object is an array, the array bounds are checked.
	Pointer arithmetic	
	Relational operators on pointers	`CBMC` checks that the two operands point to the same object.
	Pointer Type Casts	Upon dereferencing, the type of the object and the expression are checked to match
	Pointers to Functions	The offset within the object is checked to be zero
Dynamic Memory	`malloc` and `free` are supported. The argument of malloc may be a nondeterministically chosen, arbitrarily large value.	Upon dereferencing, the object pointed to must still be alive. The pointer passed to `free` is checked to point to an object that is still alive. `CBMC` can check that all dynamically allocated memory is deallocated before exiting the program ("memory leaks").