TACAS, 2004.

# A Tool for Checking ANSI-C Programs

Edmund Clarke,
Daniel Kroening,
Flavio Lerda Carnegie Mellon University

JUNBEOM YOO

Dependable Software Laboratory
KONKUK University

http://dslab.konkuk.ac.kr

2010.11.17

## 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.

## Contents

## 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
  2) the tool can compare an ANSI-C program with another design, such as a circuit given in Verilog.

## 1. Introduction

- We describe a tool (CBMC) that formally verifies ANSI-C programs.
  - Checked include pointer safety, array bounds, and user-provided assertions.
  - Implements a technique called Bounded Model Checking (BMC) [1].
  - + GUI
- 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 (CBMC) checks that sufficient unwinding is done to ensure that no longer counterexample can exist by means of *unwinding assertions.*

## 1. Introduction

- Hardware Verification using ANSI-C as a Reference

- There are two implementations of the same design:
  (1) One written in ANSIC, which is written for simulation,
  (2) One written in register transfer level HDL, which is the actual product.

- The ANSI-C implementation is usually thoroughly tested and debugged. 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.

- Due to market constraints,
- An automated, or nearly automated way of establishing the consistency of the HDL implementation with respect to the ANSI-C model is highly desirable.

## 1. Introduction

- 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.

- In order to verify the consistency of the two implementations, we unwind both the C program and the circuit in tandem.

## 2. Bounded Model Checking for ANSI-C Programs

- We reduce the Model Checking Problem to determining the validity of a bit vector equation.

- The process has five steps:
  1. We assume that the ANSI-C program is already preprocessed, e.g., all the #define directives are expanded.
  2. The loop constructs are unwound using assertions.
  3. Backward goto statements are unwound in a manner similar to while loops.
  4. Function calls are expanded.
  5. The program is then transformed into static single assignment (SSA) form, which requires a pointer analysis.

## 2.1 Generating the Formula

- 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 ∧ ¬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.

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

assert(x<=3);
```

$$\rightarrow$$

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

$$\rightarrow$$

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

## 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.

## 3. A Graphical User Interface

- 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:
  - Verification of properties of C programs
  - Checking consistency of Verilog designs against a C implementation
- 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.

## 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.

# ANSI-C Language Features

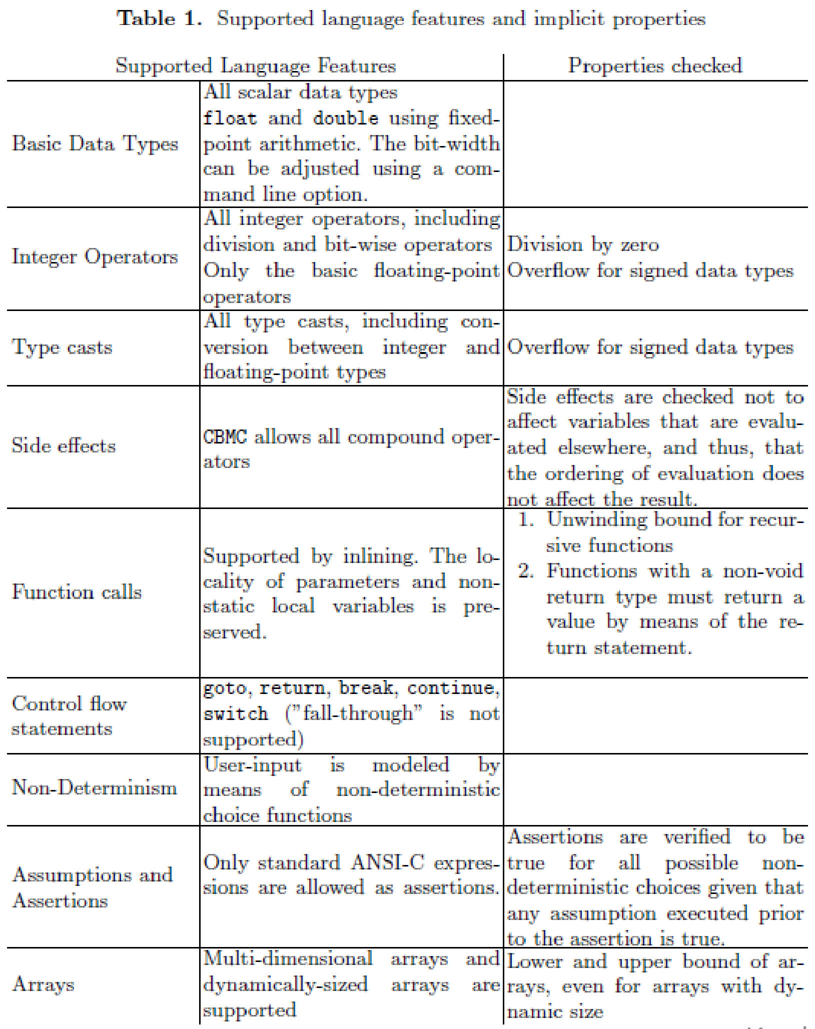


**Table 1.** Supported language features and implicit properties

| 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 |

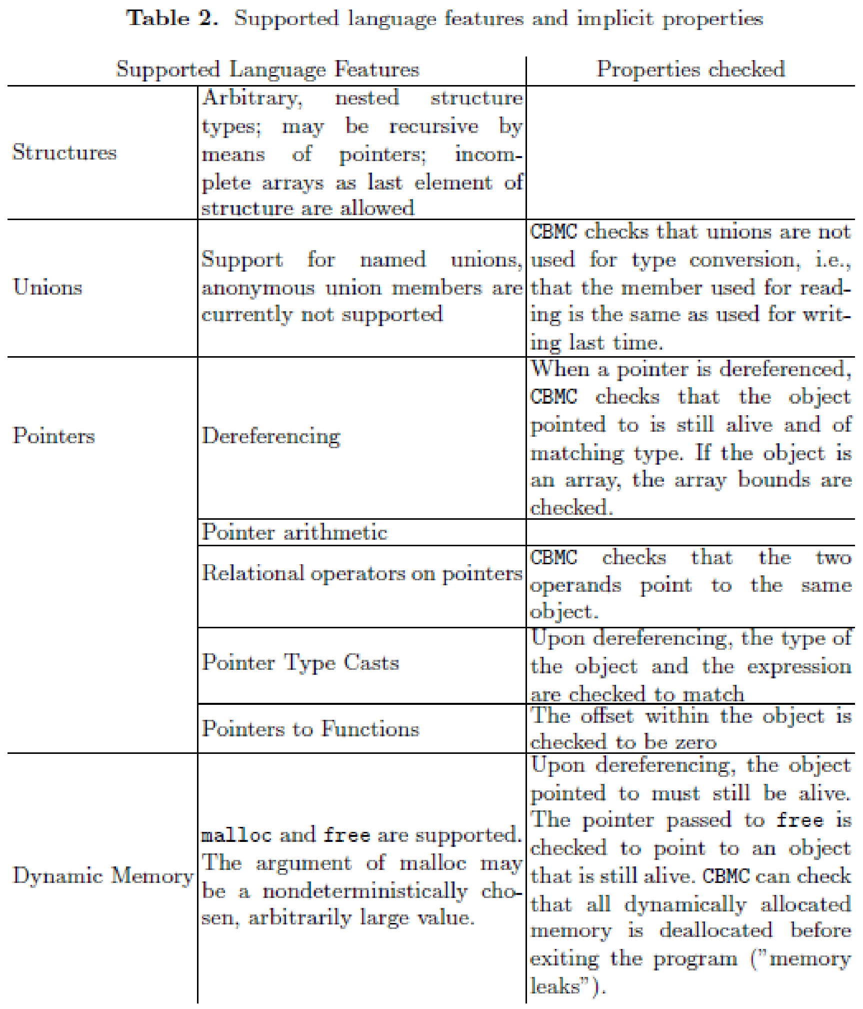
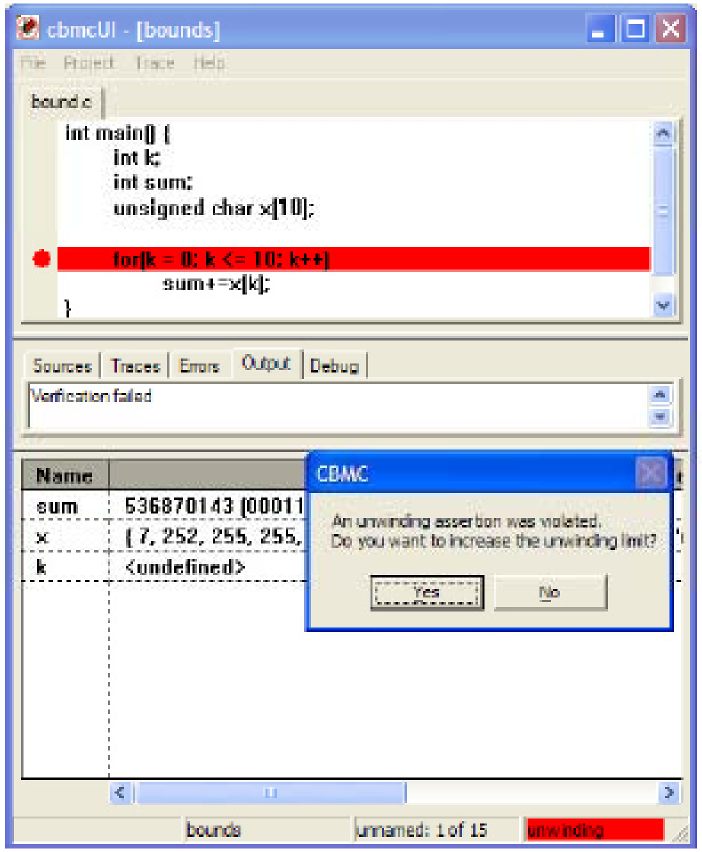


**Table 2.** Supported language features and implicit properties

| 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"). |

## CBMC



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

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

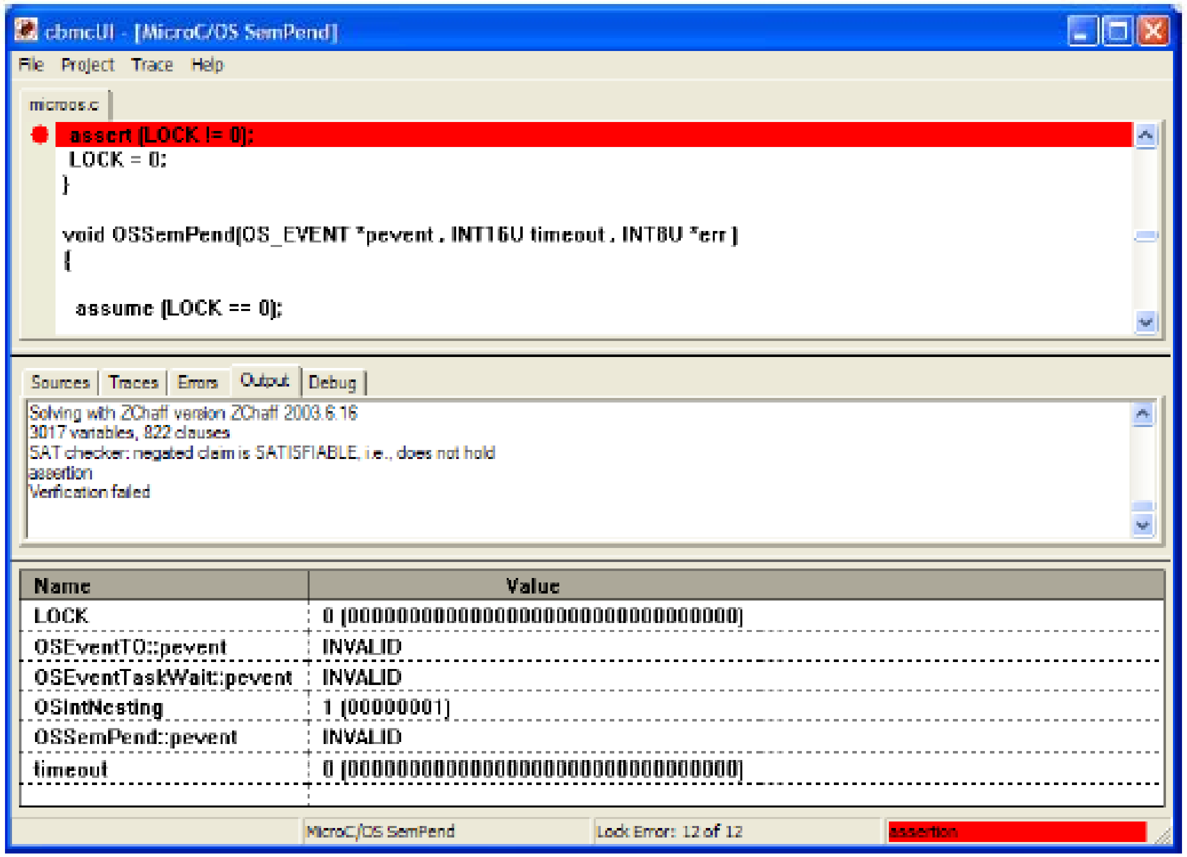
```
Verification failed
```

```
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.

## CBMC


cbmcUI - [MicroC/OS SemPend]
File Project Trace Help
micros.c
Sources Traces Errors Output Debug

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

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

  assume (LOCK == 0);
```

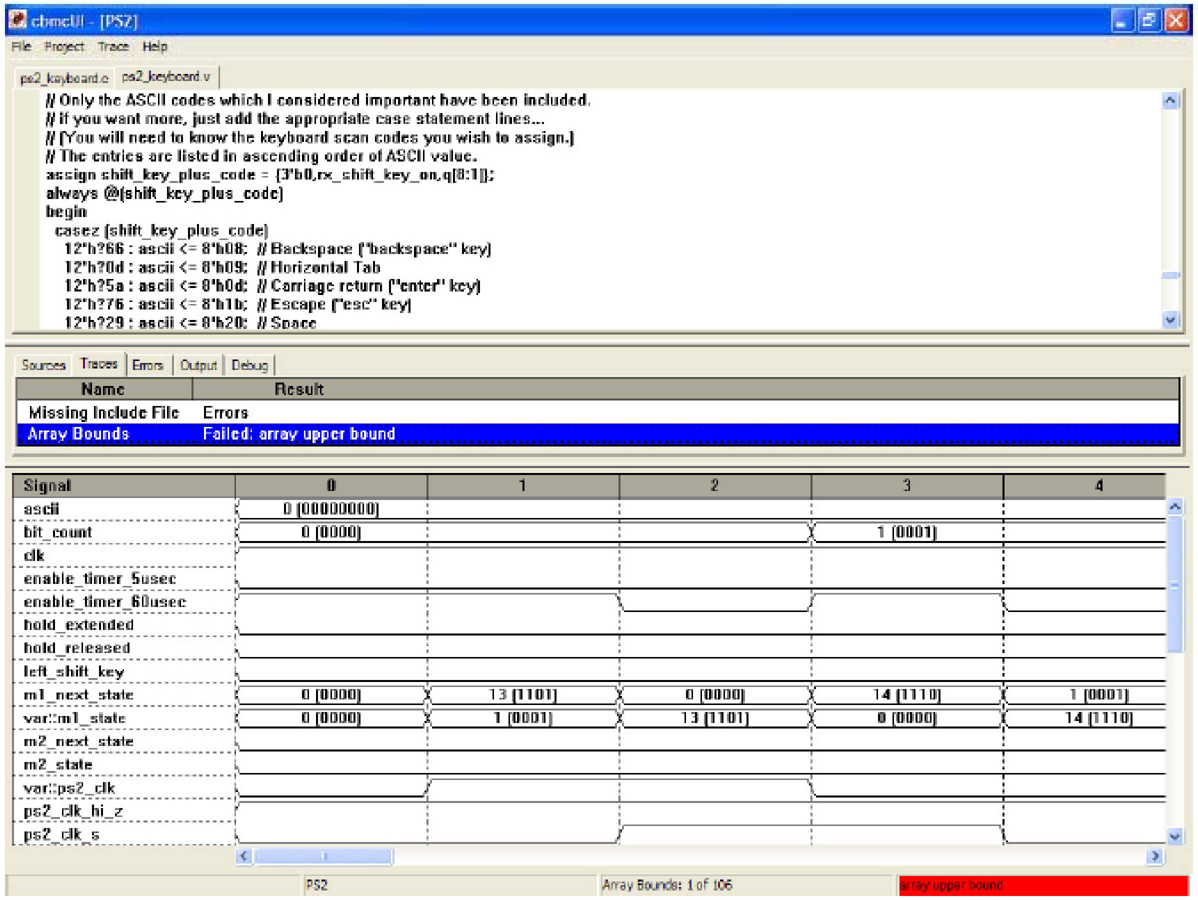
```
Solving with ZChaff version ZChaff 2003.6.16
3017 variables, 822 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
assertion
Verification failed
```

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

MicroC/OS SemPend | Lock Error: 12 of 12 | assertion

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.

# CBMC

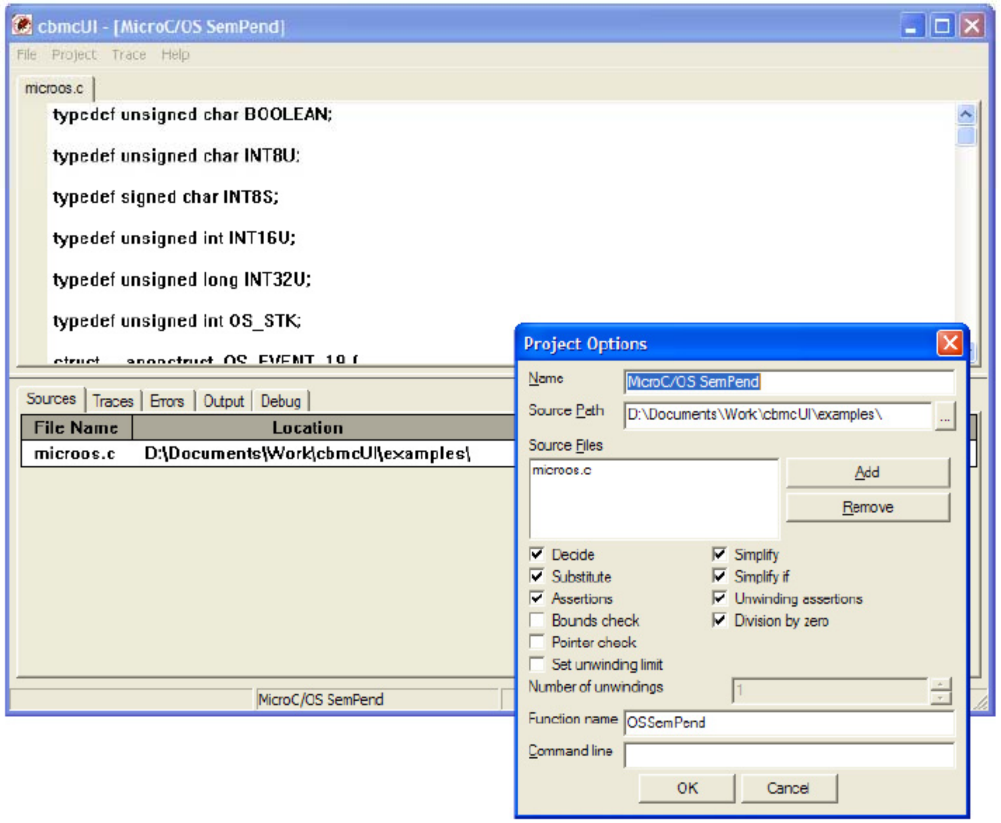


```
// Only the ASCII codes which I considered important have been included.
// If you want more, just add the appropriate case statement lines...
// (You will need to know the keyboard scan codes you wish to assign.)
// The entries are listed in ascending order of ASCII value.
assign shift_key_plus_code = {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 |

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

# CBMC



```
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\ |

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.