ANSI-C Bounded Model Checker
User Manual

Edmund Clarke
Daniel Kroening
August 2, 2006

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We describe a tool that formally verifies ANSI-C programs. The tool implements a technique called Bounded Model Checking (BMC). In BMC, the transition relation for a complex state machine and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability by using a SAT procedure. The tool supports all ANSI-C integer operators and all pointer constructs allowed by the ANSI-C standard, including dynamic memory allocation, pointer arithmetic, and pointer type casts.

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.

Keywords: ANSI-C, Bounded Model Checking, Pointers

Chapter 1

Introduction

1.1 Bounded Model Checking for ANSI-C

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. 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 verification of low level ANSI-C code is challenging due to extensive use of arithmetic, pointers and pointer arithmetic, and bit-wise operators.

We describe a tool that formally verifies ANSI-C programs. The properties checked include pointer safety and array bounds, and user-provided assertions. The tool implements a technique called Bounded Model Checking (BMC). In BMC, the transition relation for a complex state machine and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability by using a SAT procedure. If the formula is satisfiable, a counterexample is extracted from the output of the SAT procedure. If the formula is not satisfiable, the program can be unwound more to determine if a longer counterexample exists. The tool checks that sufficient unwinding is done to ensure that no longer counterexample can exist by means of *unwinding assertions*.

While one feature of the tool is to check conformance of the program with the ANSI-C standard [1], many architecture specific extensions of C are supported if enabled by the user. As example, `CBMC` supports both little-endian and big-endian machine word layouts.

1.2 A Short Tutorial

1.2.1 First Steps

Like a compiler, `CBMC` takes the name of `.c` files on the command line. `CBMC` then translates the program and merges the function definitions from the various `.c` files,

just like a linker. However, instead of producing a binary for execution, CBMC performs symbolic simulation on the program.

As an example, consider the following simple program, named file1.c:

```
int puts(const char *s) { }

int main(int argc, char **argv) {
  int i;

  if(argc>=1)
    puts(argv[2]);
}
```

Of course, this program is faulty, as the argv array might have only two elements (argv[0] followed by NULL), and then the array access argv[2] is out of bounds. Now, run CBMC as follows:

```
cbmc --show-vcc file1.c
```

CBMC prints the verification conditions it generates for the program, formatted as a Gentzen sequent. Note that it prints a comment together with each sequent, in this case "array argv upper bound" together with the location of the faulty array access. CBMC prints the equation since the built-in simplifier is not able to determine its validity. Now run CBMC without the --show-vcc option, which invokes a decision procedure to solve the formula:

```
cbmc file1.c
```

CBMC transforms the equation you have seen before into CNF and passes it to a SAT solver. It can now detect that the equation is actually not valid, and thus, there is a bug in the program. It prints a counterexample trace, i.e., a program trace that ends in a state which violates the property. In case of the example, it ends in the faulty array access. It also shows the values the input variables must have for the bug to occur. In this example, argc must be one or larger to trigger the out-of-bounds array access. If you change the condition in the example to argc>=2, the bug is fixed and CBMC will report a successful verification run.

1.2.2 Verifying Modules

In the example above, we used a program that starts with a main function. However, CBMC is aimed at embedded software, and these kinds of programs usually have different entry points. Furthermore, CBMC is also useful for verifying program modules. Consider the following example, called file2.c:

```
int array[10];

int sum() {
  unsigned i, sum;

  sum=0;
```

```
  for(i=0; i<10; i++)
    sum+=array[i];
}
```

In order to set the entry point to the `sum` function, run

```
cbmc file2.c --function sum
```

Note that global variables are initialized before the execution of the given function, which might result in a state that is different from the state the function is called within the program itself. Also, note that CBMC does not attempt to guess pre-conditions for the arguments of the function.

1.2.3 Loop Unwinding

You will note that `CBMC` unwinds the `for` loops in the program. As `CBMC` performs Bounded Model Checking, all loops have to have a finite upper run-time bound in order to guarantee that all bugs are found. `CBMC` actually checks that enough unwinding is performed. As an example, consider `binsearch.c`:

```
int binsearch(int x) {
  int a[16];
  signed low=0, high=16;

  while(low<high) {
    signed middle=low+((high-low)>>1);

    if(a[middle]<x)
      high=middle;
    else if(a[middle]>x)
      low=middle+1;
    else // a[middle]=x !
      return middle;
  }

  return -1;
}
```

If you run `CBMC` on this function, you will notice that the unwinding does not stop. The built-in simplifier is not able to determine a run time bound for this loop. The unwinding bound has to be given as a command line argument:

```
cbmc binsearch.c --function binsearch --unwind 6 --decide
```

`CBMC` not only verifies the array bounds (note that this actually depends on the result of the right shift), but also checks that enough unwinding is done, i.e., it proves a run-time bound. For any lower unwinding bound, there are traces that require more loop iterations. Thus, `CBMC` will produce an appropriate counterexample.

1.2.4 Unbounded Loops

However, CBMC can also be used for programs with unbounded loops. In this case, CBMC is used for bug hunting only; CBMC does not attempt to find all bugs. Consider the following program:

```
_Bool nondet_bool();
_Bool LOCK = 0;

_Bool lock() {
  if(nondet_bool()) {
    assert(!LOCK);
    LOCK=1;
    return 1; }

  return 0;
}

void unlock() {
  assert(LOCK);
  LOCK=0;
}

int main() {
  unsigned got_lock = 0;
  int times;

  while(times > 0) {
    if(lock()) {
      got_lock++;
      /* critical section */
    }

    if(got_lock!=0)
      unlock();

    got_lock--;
    times--;
}  }
```

The `while` loop in the `main` function has no (useful) run-time bound. Thus, the –unwind parameter has to be used in order to prevent infinite unwinding. However, you will note that CBMC will detect that not enough unwinding is done and aborts with an unwinding assertion violation.

In order to disable this test, run CBMC with the following option:

```
--no-unwinding-assertions
```

For an unwinding bound of one, no bug is found. However, already for a bound of two, CBMC detects a trace that violates an assertion. Without unwinding assertions, CBMC does not prove the program correct, but it can be helpful to find program bugs.

1.2.5 A Note About the ANSI-C Library

The ANSI-C library that comes with any C compiler may not be usable for `CBMC`. It usually comes as binary and not as source code, and furthermore, usually contains architecture dependent code. In case of the GNU C Library, even the prototypes in the `.h` file contain code that is not compliant with the ANSI-C standard. `CBMC` ships with prototypes that are standard compliant. You need to define this path as include path for the ANSI-C pre-processor.

E.g., assuming the `bash` shell is used and `gcc` is the pre-processor, this statement adjusts the path accordingly:

```
export C_INCLUDE_PATH=/path-to-cprover/examples/ansi-c-lib/
```

This directory only contains the prototypes, no implementation of the library functions is currently provided.

1.3 Outline

In chapter 2, we describe the ANSI-C language features supported by `CBMC`. In chapter 3, we describe how `CBMC` can be used to check conformance of hardware designs with an ANSI-C specification. Chapter 4 describes the command line interface.

1.4 Further Reading

The first version of `CBMC` was introduced in [2] in the context of equivalence checking. Improved versions are described in [3] and [4].

Chapter 2

ANSI-C Language Features

2.1 Basic Datatypes

CBMC supports the scalar data types as defined by the ANSI-C standard, including _Bool. By default int is 32 bits wide, short int is 16 bits wide, and char is 8 bits wide. Using a command line option, these default widths can be changed. By default, char is signed. Since some architectures use an unsigned char type, a command line option allows to change this setting.

There is also support for the floating point data types float, double, and long double. By default, CBMC uses fixed-point arithmetic for these types. Variables of type float have by default 32 bits (16 bits integer part, 16 bits fractional part), variables of type double and long double have 64 bits.

In addition to the types defined by the ANSI-C standard, CBMC supports the following Microsoft C extensions: __int8, __int16, __int32, __int64. These types define a bit vector with the given number of bits.

2.2 Operators

2.2.1 Boolean Operators

CBMC supports all ANSI-C Boolean operators on scalar variables a, b:

Operator	Description
!a	negation
a && b	and
a || b	or

2.2.2 Integer Arithmetic Operators

CBMC supports all integer arithmetic operators on scalar variables a, b:

Operator	Description
`-a`	unary minus, negation
`a+b`	sum
`a-b`	subtraction
`a*b`	multiplication
`a/b`	division
`a%b`	remainder
`a<<b`	bit-wise left shift
`a>>b`	bit-wise right shift
`a&b`	bit-wise and
`a|b`	bit-wise or
`a^b`	bit-wise xor
`a< b`	relation
`a<=b`	relation
`a> b`	relation
`a>=b`	relation

Note that the multiplication, division, and reminder operators are very expensive with respect to the size of the equation that is passed to the SAT solver. Furthermore, the equations are hard to solve for all SAT solvers known to us.

As an example, consider the following program:

```
unsigned char nondet_uchar();

int main() {
  unsigned char a, b;
  unsigned int result=0, i;

  a=nondet_uchar();
  b=nondet_uchar();

  for(i=0; i<8; i++)
    if((b>>i)&1)
      result+=(a<<i);

  assert(result==a*b);
}
```

The program non-deterministically selects two 8-bit unsigned values, and then uses shift-and-add to multiply them. It then asserts that the result (i.e., the sum) matches `a*b`. Although the resulting SAT instance has only about 2400 variables, it takes about 3 minutes to solve using MiniSAT.

Properties Checked Optionally, `CBMC` allows checking for arithmetic overflow in case of signed operands. In case of the division and the remainder operator, `CBMC` checks for division by zero. This check can be disabled using a command line option.

As an example, the following program non-deterministically selects two unsigned integers `a` and `b`. It then checks that either of them is non-zero and then computes the inverse of `a+b`:

```
int main() {
  unsigned int a, b, c;

  a=nondet_uint();
  b=nondet_uint();

  if(a>0 || b>0)
    c=1/(a+b);
}
```

However, due to arithmetic overflow when computing the sum, the division can turn out to be a division by zero. CBMC generates a counterexample as follows for the program above:

```
Initial State
-----------------------------------------------------
  c=0 (00000000000000000000000000000000)

State 1 file div_by_zero.c line 4 function main
-----------------------------------------------------
  a=4294967295 (11111111111111111111111111111111)

State 2 file div_by_zero.c line 5 function main
-----------------------------------------------------
  b=1 (00000000000000000000000000000001)

Failed assertion: division by zero file div_by_zero.c
line 8 function main
```

2.2.3 Floating Point Arithmetic Operators

CBMC supports the following operators on variables of the types `float`, `double`, and `long double`:

Operator	Description
`-a`	unary minus, negation
`a+b`	sum
`a-b`	subtraction
`a*b`	multiplication
`a< b`	relation
`a<=b`	relation
`a> b`	relation
`a>=b`	relation

Division is currently not supported. Note that the multiplication operator is very expensive with respect to the size of the equation that is passed to the SAT solver. Furthermore, the equations are hard to solve for all SAT solvers known to us. CBMC also supports type conversions to and from integer types.

2.2.4 The Comma Operator

CBMC supports the comma operator `a, b`. The operands are evaluated for potential side effects. The result of the operator is the right operand.

2.2.5 Type Casts

CBMC has full support for arithmetic type casts. As an example, the expression

```
(unsigned char)i
```

for an integer `i` is guaranteed to be between 0 and 255 in case of an eight bit character type.

Properties Checked For the unsigned data types, the ANSI-C standard requires modulo semantics, i.e., that no overflow exception occurs. Thus, overflow is not checked. For signed data types, an overflow exception is permitted. Optionally, CBMC checks for such arithmetic overflows.

2.2.6 Side Effects

CBMC allows all side effect operators with their respective semantics. This includes the assignment operators (=, +=, etc.), and the pre- and post- increment and decrement operators.

As an example, consider the following program fragment:

```
unsigned int i, j;

i=j++;
```

After the execution of the program, the variable `i` will contain the initial value of `j`, and `j` will contain the initial value of `j` plus one. CBMC generates the following equation from the program:

$$\begin{aligned} i_1 &= j_0 \\ j_1 &= j_0 + 1 \end{aligned}$$

CBMC performs the implicit type cast as required by the ANSI-C standard. As an example, consider the following program fragment:

```
char c;
int i;
long l;

l = c = i;
```

The value of `i` is converted to the type of the assignment expression `c=i`, that is, `char` type. The value of this expression is then converted to the type of the outer assignment expression, that is, `long int` type.

Ordering of Evaluation The ANSI-C standard allows arbitrary orderings for the evaluations of expressions and the time the side-effect becomes visible. The only exceptions are the operators `&&`, `||`, and the trinary operator `?:`. For the Boolean operators, the standard requires strict evaluation from left to right, and that the evaluation aborts once the result is known. The operands of the expression `c ? x : y` must be evaluated as follows: first, `c` is evaluated. If `c` is true, `x` is evaluated, and `y` otherwise.

As an example, assume that a pointer `p` in the following fragment may point to either `NULL` or a valid, active object. Then, an `if` statement as follows is valid, since the evaluation must be done from left to right, and if `p` points to NULL, the result of the Boolean AND is known to be false and the evaluation aborts before `p` is dereferenced.

```
if(p!=NULL && *p==5) {
  . . .
```

For other operators, such as addition, no such fixed ordering exists. As an example, consider the following fragment:

```
int g;

int f() {
  g=1;
  ...
}

...
  g=2;

  if(f()+g==1) {
    ...
```

In this fragment, a global variable `g` is assigned to by a function `f()`, and just before an `if` statement. Furthermore, `g` is used in an addition expression in the condition of the `if` statement together with a call to `f()`. If `f()` is evaluated first, the value of `g` in the sum will be one, while it is two if `g` is evaluated first. The actual result is architecture dependent.

Properties Checked `CBMC` models this problem as follows: One option allows setting a fixed ordering of evaluation for all operators. The other option allows checking for such artifacts: `CBMC` asserts that no side-effect affects the value of any variable that is evaluated with equal priority. This includes changes made indirectly by means of pointers. In the example, this is realized by write-protecting the variable `g` during the execution of `f`. This rules out programs that show architecture dependent behavior due to the ordering of evaluation. While such programs are still valid ANSI-C programs, we do not believe that programs showing architecture dependent behavior are desirable.

2.2.7 Function calls

CBMC supports functions by inlining. No modular approach is done. CBMC preserves the locality of the parameters and the non-static local variables by renaming.

As an example, the following program calls the functions `f()` and `g()` twice. While `f()` uses a static variable, which is not renamed between calls, `g()` uses a true local variable, which gets a new value for each call.

```
int f() {
  static int s=0;

  s++;

  return s;
}

int g() {
  int l=0;

  l++;

  return l;
}

int main() {
  assert(f()==1); // first call to f
  assert(f()==2); // second call to f
  assert(g()==1); // first call to g
  assert(g()==1); // second call to g
}
```

Recursion is implemented by finite unwinding, as done for `while` loops. CBMC checks that enough unwinding is done by means of an *unwinding assertion* (section 2.3.6 provides more details).

2.3 Control Flow Statements

2.3.1 Conditional Statement

CBMC allows the use of the conditional statement as described in the ANSI-C standard.

Properties Checked CBMC generates a warning if the assignment operator is used as condition of a control flow statement such as `if` or `while`.

2.3.2 `return`

The `return` statement without value is transformed into an equivalent `goto` statement. The target is the end of the function. The `return` statement with value is

transformed into an assignment of the value returned and the `goto` statement to the end of the function.

Properties Checked `CBMC` enforces that functions with a non-void return type return a value by means of the return statement. The execution of the function must not end by reaching the end of the function. This is realized by inserting `assert(FALSE);` at the end of the function. `CBMC` reports an error trace if this location is reachable.

As an example, consider the following fragment:

```
int f() {
  int c=nondet_int();

  if(c!=1)
    return c;
}

int main() {
  int i;
  i=f();
}
```

In this fragment, `f()` may exit without returning a value. `CBMC` produces the following counterexample:

```
State 1 file no-return.c line 2 function f
------------------------------------------------------
  c=1 (00000000000000000000000000000001)

Failed assertion: end-of-function assertion
file no-return.c line 6 function f
```

2.3.3 `goto`

While only few C programs make use of `goto` statements, `CBMC` provides support for such programs. We distinguish forward and backward jumps. In case of backward jumps, the same technique used for loops is applied: the loop is unwound a given number of times, and then we check that this number of times is sufficient by replacing the `goto` statement by `assert(FALSE);`.

2.3.4 `break` and `continue`

The `break` and `continue` statements are replaced by equivalent `goto` statements as described in the ANSI-C standard.

2.3.5 `switch`

`CBMC` provides full support for `switch` statements, including ”fall-through”, as in the following example:

```
switch(i) {
  case 1: j++;
  case 2: j++; break;
}
```

2.3.6 Loops

In Bounded Model Checking, the transition system is unwound up to a finite depth. In case of C programs, this means that `for` and `while` are unwound up to a certain depth. In many cases, `CBMC` is able to automatically detect the maximum number of times a loop can be executed. This includes `while` loops and loops with modifications to the loop counter inside the loop body, even when done indirectly using a pointer.

However, in case of loops that have no pre-set bound, e.g., loops iterating on dynamic data structures, the user must specify a bound by means of the `--unwind` command line argument. `CBMC` will then unwind the loops up to that bound and check that the number is large enough by means of an *unwinding assertion*.

2.4 Non-Determinism

`CBMC` allows to model user-input by means of non-deterministic choice functions. The names of these functions have the prefix `nondet_`. The value range generated is determined by the return type of the function. As an example,

```
int nondet_int();
```

returns a non-deterministically chosen value of type `int`. The functions are built-in, i.e., the prototype is sufficient. `CBMC` will evaluate all traces arising from all possible choices.

2.5 Assumptions and Assertions

`CBMC` checks assertions as defined by the ANSI-C standard: The `assert` statement takes a Boolean condition, and `CBMC` checks that this condition is true for all runs of the program. The logic for assertions is the usual ANSI-C expression logic.

In addition to the `assert` statement, `CBMC` provides the `__CPROVER_assume` statement. The `__CPROVER_assume` statement restricts the program traces that are considered and allows assume-guarantee reasoning. As an assertion, an assumption takes a Boolean expression. Intuitively, one can consider the `__CPROVER_assume` statement to abort the program *successfully* if the condition is false. If the condition is true, the execution continues.

As an example, the following function first non-deterministically picks an integer value. It then assumes that the integer is in a specific range and returns the value.

```
int one_to_ten() {
  int value=nondet_int();
  __CPROVER_assume(value>=1 && value<=10);
```

```
  return value;
}
```

Note that the assume statement is not retro-active with respect to assertions. E.g.,

```
  assert(value<10);
  __CPROVER_assume(value==0);
```

may fail, while

```
  __CPROVER_assume(value==0);
  assert(value<10);
```

passes.

When using the `__CPROVER_assume` statement, it must be ensured that there still exists a program trace that satisfies the condition. Otherwise, any property will pass vacuously. This should be checked by replacing the property by false. If no counterexample is produced, the assumptions eliminate all program paths.

2.6 Arrays

`CBMC` allows arrays as defined by the ANSI-C standard. This includes multi-dimensional arrays and dynamically-sized arrays.

Dynamic Arrays The ANSI-C standard allows arrays with non-constant size as long as the array does not have static storage duration, i.e., is a non-static local variable. Even though such a construct has a potentially huge state space, `CBMC` provides full support for arrays with non-constant size. The size of the Boolean equation that is generated does not depend on the array size, but rather on the number of read or write accesses to the array.

Properties Checked `CBMC` checks both lower and upper bound of arrays, even for arrays with dynamic size. As an example, consider the following fragment:

```
  unsigned size=nondet_uint();
  char a[size];

  a[10]=0;
```

In this fragment, an array `a` is defined, which has a non-deterministically chosen size. The code then accesses the array element with index 10. `CBMC` produces a counterexample with an upper array bound error on array `a`. The trace shows a value for `size` less then 10.

Furthermore, `CBMC` checks that the size of arrays with dynamic size is non-negative. As an example, consider the following fragment:

```
  signed size=nondet_int();
  char a[size];
```

For this fragment, CBMC produces a counterexample in which the size of the array a is negative.

2.7 Structures

CBMC allows arbitrary structure types. The structures may be nested, and may contain arrays.

The sizeof operator applied to a structure type yields the sum of the sizes of the components. However, the ANSI-C standard allows arbitrary padding between components. In order to reflect this padding, the sizeof operator should return the sum of the sizes of the components *plus* a non-deterministically chosen non-negative value.

Recursive Structures Structures may be recursive by means of pointers to the same structure. As an example, consider the following fragment:

```
struct nodet {
  struct nodet *n;
  int payload;
};

int main() {
  unsigned i;
  struct nodet *list=(void *)0;
  struct nodet *new_node;

  for(i=0; i<10; i++) {
    new_node=malloc(sizeof(*new_node));
    new_node->n=list;
    list=new_node;
  }
}
```

The fragment builds a linked list with ten dynamically allocated elements.

Structures with Dynamic Array The last component of an ANSI-C structure may be an incomplete array (an array without size). This incomplete array is used for dynamic allocation. This is described in section 2.10.

2.8 Unions

CBMC allows the use of unions to use the same storage for multiple data types. Internally, CBMC actually shares the literals used to represent the variables values among

the union members.

Properties Checked CBMC does not permit the use of unions for type conversion, as this would result in architecture dependent behavior. Specifically, if a member is read, the same member must have been used for writing to the union the last time.

2.9 Pointers

2.9.1 The Pointer Data Type

Pointers are commonly used in ANSI-C programs. In particular, pointers are required for call by reference and for dynamic data structures. CBMC provides extensive support for programs that use pointers according to rules set by the ANSI-C standard, including pointer type casts and pointer arithmetic.

The size of a pointer, e.g., `sizeof(void *)` is by default 4 bytes. This can be adjusted using a command line option.

Conversion of pointers from and to integers The ANSI-C standard does not provide any guarantees for the conversion of pointers into integers. However, CBMC ensures that the conversion of the same address into an integer yields the same integer. The ANSI-C standard does not guarantee that the conversion of a pointer into an integer and then back yields a valid pointer. CBMC does not allow this construct.

2.9.2 Pointer Arithmetic

CBMC supports the ANSI-C pointer arithmetic operators. As an example, consider the following fragment:

```
int array[10], *p;

int main() {
        array[1] = 1;
        p = &array[0];
        p++;

        assert(*p == 1);
}
```

2.9.3 The Relational Operators on Pointers

The ANSI-C standard allows comparing to pointers using the relational operators <=, <, >=, <.

Properties Checked The standard restricts the use of these operators to pointers that point to the same object. CBMC enforces this restriction by means of an automatically generated assertion.

2.9.4 Pointer Type Casts

CBMC provides full support for pointer type casts as described by the ANSI-C standard. As an example, it is a common practice to convert a pointer to, e.g., an integer into a pointer to `void` and then back:

```
int i;
void *p;

p=&i;
. . .
*((int *)p)=5;
```

Note that pointer type casts are frequently used for architecture specific type conversions, e.g., to write an integer byte-wise into a file or to send it over a socket:

```
int i;
char *p;

p=(char *)&i;

for(j=0; j<4; j++) {
  /* write *p */
  p++;
}
```

The result is architecture-dependent. In particular, it exposes the endianess of the architecture. CBMC allows these constructs if the endianess is specified on the command line with one of the following two options:

```
--little-endian
--big-endian
```

Properties Checked CBMC checks that the type of the object being accessed matches the type of the dereferencing expression. For example, the following fragment uses a `void *` pointer to store the addresses of both `char` and `int` type objects:

```
int nondet_int();

void *p;
int i;
char c;

int main() {
  int input1, input2, z;

  input1=nondet_int();
  input2=nondet_int();

  p=input1? (void *)&i : (void *)&c;
```

```
  if(input2)
    z=*(int *)p;
  else
    z=*(char *)p;
}
```

CBMC produces the following counterexample:

```
Initial State
-----------------------------------------------------
  c=0 (00000000)
  i=0 (00000000000000000000000000000000)
  p=NULL

State 1 file line 10 function main
-----------------------------------------------------
  input1=0 (00000000000000000000000000000000)

State 2 file line 11 function main
-----------------------------------------------------
  input2=1 (00000000000000000000000000000001)

State 3 line 13 function main
-----------------------------------------------------
  p=&c

Failed assertion: dereference failure (wrong object type)
line 16 function main
```

Note that the ANSI-C standard allows the conversion of pointers to structures to another pointer to a prefix of the same structure. As an example, the following program performs a valid pointer conversion:

```
typedef struct {
  int i;
  char j;
} s;

typedef struct {
  int i;
} prefix;

int main() {
  s x;
  prefix *p;

  p=(prefix *)&x;

  p->i=1;
}
```

2.9.5 String Constants

ANSI-C implements strings of characters as an array. Strings are then often represented by means of a pointer pointing to the array. Array bounds violations of string arrays are the leading cause of security holes in Internet software such as servers or web browsers.

CBMC provides full support for string constants, usable either in initializers or as a constant. As an example, the following fragment contains a string array `s`, which is initialized using a string constant. Then, a pointer `p` is initialized with the address of `s`, and the second character of `s` is modified indirectly by dereferencing `p`. The program then asserts this change to `s`.

```
char s[]="abc";

int main() {
  char *p=s;

  /* write to p[1] */
  p[1]='y';

  assert(s[1]=='y');
}
```

Properties Checked CBMC performs bounds checking for string constants as well as for normal arrays. In the following fragment, a pointer `p` is pointing to a string constant of length three. Then, an input `i` is used as address of an array index operation. CBMC asserts that the input `i` is not greater than four (the string constant ends with an implicit zero character).

```
char *p="abc";

void f(unsigned int i) {
  char ch;

  /* results in bounds violation with i>4 */
  ch=p[i];
}
```

In addition to that, CBMC checks that string constants are never written into by means of pointers pointing to them.

2.9.6 Pointers to Functions

CBMC allows pointers to functions, and calls through such a pointer. The function pointed to may depend on non-deterministically chosen inputs. As an example, the following fragment contains a table of three function pointers. The program uses a function argument to index the table and then calls the function. It then asserts that the right function was called.

```
int global;

int f() { global=0; }
int g() { global=1; }
int h() { global=2; }

typedef int (*fptr)();
fptr table[] = { f, g, h };

void select(unsigned x) {
  if(x<=2) {
    table[x]();
    assert(global==x);
  }
}
```

2.10 Dynamic Memory

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);
}
```

Properties Checked Optionally, CBMC checks array bounds of dynamically allocated arrays, and it checks that a pointer pointing to a dynamic object is pointing to an active object (i.e., that the object has not yet been freed and that it is not a static object). Furthermore, CBMC checks that an object is not freed more than once.

In addition to that, CBMC can check that all dynamically allocated memory is deallocated before exiting the program, i.e., CBMC can prove the absence of ”memory leaks”.

As an example, the following fragment dynamically allocates memory, and stores the address of that memory in a pointer `p`. Depending on an input `i`, this pointer is redirected to a local variable `y`. The memory pointed to by `p` is then deallocated using `free`. CBMC detects that there is an illegal execution trace in case that the input `i` is true.

```
void f(_Bool i) {
  int *p;
```

```
  int y;

  p=malloc(sizeof(int)*10);

  if(i) p=&y;

  /* error if p points to y */
  free(p);
}
```

Chapter 3

Hardware Verification using ANSI-C as a Reference

3.1 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 is later on 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 [5]. 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 [6], 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. The System C standard defines a subset of C++ that can be used for synthesis [7]. Other variants of ANSI-C for specifying hardware are Spec C and Handel C, among others.

The concept of verifying the equivalence of a software implementation and a synchronous transition system was introduced by Pnueli, Siegel, and Shtrichman [8]. The C program is required to be in a very specific form, since a mechanical translation is assumed.

In [9], Currie, Hu, and Rajan transform DSP assembly language into an equation for the Stanford Validity Checker. However, problems involving bit vector overflow are not detected and while loops are not supported. The symbolic execution of programs for comparison with RTL is common practice [10, 11].

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. Furthermore, our methodology allows arbitrary loop constructs.

3.2 A small Tutorial

The following Verilog module implements a 4-bit counter:

```
module main(clk);

input clk;
reg [3:0] counter;

initial counter=0;

always @(posedge clk)
  counter=counter+1;

endmodule
```

`CBMC` can take Verilog modules as the one above as additional input. The trace produced by the Verilog module is provided to the C program by means of arrays. For the example above, the following C fragment is the declaration of the array that corresponds to the trace values of the `counter` register:

```
extern const unsigned int counter[];
```

Note that the array has no size specification. However, as `CBMC` performs Bounded Model Checking, the size of the array must be bounded. As it is desirable to change the bound to adjust it to the available computing capacity, the bound is given on the command line and not as part of the C program. This makes it easy to use only one C program for arbitrary bounds. The actual bound is available in the C program using the following declaration:

```
extern const unsigned int bound;
```

Also note that the array declaration above declares an array with constant members. Thus, the C program can only read the trace values and not modify them. We later on describe how to drive inputs of the Verilog module within the C program.

As described in previous chapters, assertions can be used to verify properties of the Verilog trace. As an example, the following program checks two values of the trace of the counter module:

```
extern const unsigned int bound;
extern const unsigned int counter[];

int main() {
  assert(counter[5]==5);
  assert(counter[16]==0);
}
```

The following `CBMC` command line checks these assertions with a bound of 20:

```
hw-cbmc counter.c counter.v --module main --bound 20 --decide
```

Note that `hw-cbmc` must be used instead of `cbmc`. The module name given must match the name of the top module in the Verilog file. Multiple Verilog files can be given on the command line.

The `--bound` parameter is not to be confused with the `--unwind` parameter. While the `--unwind` parameter specifies the maximum unwinding depth for loops within the C program, the `--bound` parameter specifies the number of times the transition relation of the Verilog module is to be unwound.

Counterexamples For the given example, the verification is successful. If the first assertion is changed to

```
  assert(counter[5]==0);
```

and the bound on the command line is changed to 6, `CBMC` produces a counterexample. `CBMC` produces two traces: One for the C program, which matches the traces described earlier, and a separate trace for the Verilog module. The values of the registers in the Verilog module are also shown in the C trace as part of the initial state.

```
Initial State
----------------------------------------------------
  bound=6 (00000000000000000000000000000110)
  counter= 0, 1, 2, 3, 4, 5, 6

Failed assertion: assertion line 6 function main

Transition system state 0
----------------------------------------------------
  counter=0 (0000)

Transition system state 1
----------------------------------------------------
```

```
  counter=1 (0001)

Transition system state 2
----------------------------------------------------
  counter=2 (0010)

Transition system state 3
----------------------------------------------------
  counter=3 (0011)

Transition system state 4
----------------------------------------------------
  counter=4 (0100)

Transition system state 5
----------------------------------------------------
  counter=5 (0101)

Transition system state 6
----------------------------------------------------
  counter=6 (0110)
```

Using the Bound The following program is using the bound variable to check the counter value in all cycles:

```
extern const unsigned int bound;
extern const unsigned int counter[];

int main() {
  unsigned cycle;

  for(cycle=0; cycle<=bound; cycle++)
    assert(counter[cycle]==(cycle & 15));
}
```

CBMC performs array bounds checking on the trace arrays. Thus, care must be taken to prevent access to the trace arrays beyond the bound.

Synchronizing Inputs The example above is trivial as there is only one possible trace. The initial state is deterministic, and there is only one possible transition, so the verification problem can be solved by mere testing. Consider the following Verilog module:

```
module main(clk, i);

input clk;
input i;
reg [3:0] counter;

initial counter=0;

always @(posedge clk)
```

```
  if(i)
    counter=counter+1;

endmodule
```

Using the C program above fails, as the Verilog module is free to use zero as value for the input `i`. This implies that the counter is not incremented. The C program has to read the value of the input `i` in order to be able to get the correct counter value:

```
extern const unsigned int bound;
extern const unsigned int counter[];
extern const unsigned _Bool i[];

int main() {
  unsigned cycle;
  unsigned C_counter=0;

  for(cycle=0; cycle<=bound; cycle++) {
    assert(counter[cycle]==(C_counter & 15));
    if(i[cycle]) C_counter++;
  }
}
```

Similarly, the C model has to synchronize on the choice of the initial value of registers if the Verilog module does not perform initialization.

Restricting the Choice of Inputs The C program can also restrict the choice of inputs of the Verilog module. This is useful for adding environment constraints. As an example, consider a Verilog module that has a signal `reset` as an input, which is active-low. The following C fragment drives this input to be active in the first cycle, and not active in any subsequent cycle:

```
  assume(resetn[0]==0);

  for(i=1; i<=bound; i++)
    __CPROVER_assume(resetn[i]);
```

Mapping Variables within the Module Hierarchy Verilog modules are hierarchical. The `extern` declarations shown above only allow reading the values of signals and registers that are in the top module. In order to read values from sub-modules, `CBMC` uses structures.

As an example, consider the following Verilog file:

```
module counter(clk, increment);

  input clk;
  input [7:0] increment;
  reg [7:0] counter;
```

```
  initial counter=0;

  always @(posedge clk)
    counter=counter+increment;

endmodule

module main(clk);

  input clk;

  counter c1(clk, 1);
  counter c2(clk, 2);

endmodule
```

The file has two modules: a main module and a counter module. The counter module is instantiated twice within the main module. A reference to the register `counter` within the C program would be ambiguous, as the two module instances have separate instances of the register. `CBMC` uses the following data structures for this example:

```
/* unwinding bound */

extern const unsigned int bound;

/*
  Module verilog::main
*/

extern const _Bool          clk[];

/*
  Module verilog::counter
*/

struct counter {
  _Bool          clk;
  unsigned char  increment;
  unsigned char  counter;
};

extern const struct counter c1[];
extern const struct counter c2[];

int main() {
  assert(c1[5].counter==5);
  assert(c2[5].counter==10);
}
```

The `main` function reads both counter values for cycle 5. A deeper hierarchy (modules in modules) is realized by using structure members. Writing these data structures

for large Verilog designs is error prone, and thus, CBMC can automatically generate them. The declarations above are generated using the command line

```
cbmc --gen-interface --module main hierarchy.v
```

Mapping Verilog Vectors to Arrays or Scalars In Verilog, a definition such as

```
wire [31:0] x;
```

can be used for arithmetic (e.g., `x+10`) and as array of Booleans (e.g., `x[2]`). ANSI-C does not allow both, so when mapping variables from Verilog to C, the user has to choose one option for each such variable. As an example, the C declaration

```
extern const unsigned int x[];
```

allows using `x` in arithmetic expressions, while the C declaration

```
extern const _Bool x[][32];
```

allows accessing the individual bits of `x` using the syntax `x[cycle][bit]`. The `--gen-interface` option of CBMC generates the first variant if the vector has the same size as one of the standard integer types, and the second option if not so. This choice can be changed by adjusting the declaration accordingly.

Chapter 4

Command Line Interface

This chapter describes the command line interface of CBMC. Like a C compiler, CBMC takes the names of the .c source files as arguments. Additional options allow to customize the behavior of CBMC.

Option	Description
--program-only	only show program expression
--function name	set main function name
--no-simplify	do not simplify
--all-claims	keep all claims
--unwind nr	unwind nr times
--unwindset nr	unwind given loop nr times
--claims-only	only show claims
--decide	run decision procedure
--dimacs	generate CNF in DIMACS format
--document-subgoals	generate subgoals documentation
--remove-assignments	remove unrelated assignments
--no-substitution	do not perform substitution
--no-simplify-if	do not simplify ?:
--no-assertions	ignore assertions
--no-unwinding-assertions	do not generate unwinding assertions
--no-bounds-check	do not do array bounds check
--no-div-by-zero-check	do not do division by zero check
--no-pointer-check	do not do pointer check
--bound nr	number of transitions
--module name	module to unwind
--counterexample file	write counterexample to file

Bibliography

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

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

[3] 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.

[4] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

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

[6] Luc Séméria, Andrew Seawright, Renu Mehra, Daniel Ng, Arjuna Ekanayake, and Barry Pangrle. RTL C-based methodology for designing and verifying a multi-threaded processor. In *Proc. of the 39th Design Automation Conference*, pages 123–128. ACM Press, 2002.

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

[8] 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.

[9] David W. Currie, Alan J. Hu, and Sreeranga Rajan. Automatic formal verification of DSP software. In *Proceedings of the 37th Design Automation Conference (DAC 2000)*, pages 130–135. ACM Press, 2000.

[10] Kiyoharu Hamaguchi. Symbolic simulation heuristics for high-level design descriptions with uninterpreted functions. In *International Workshop on High-Level Design, Validation, and Test*, pages 25–30. IEEE, 2001.

[11] C. Blank, H. Eveking, J. Levihn, and G. Ritter. Symbolic simulation techniques — state-of-the-art and applications. In *International Workshop on High-Level Design, Validation, and Test*, pages 45–50. IEEE, 2001.