Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking

Edmund Clarke Daniel Kroening Karen Yorav

May 2003
CMU-CS-03-126

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

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

This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU.

Keywords: Hardware Verification, ANSI-C, Bounded Model Checking

1 Introduction

When a new device is designed, a ”golden model” is often written in a programming language like ANSI-C. This model together with any embedded software that will run on the device is extensively simulated to insure both correct functionality and performance. Later, the device is implemented in a hardware description language like Verilog. It is essential to determine if the C and Verilog programs are consistent [1].

We show how the consistency test can be automated by using a formal verification technique called Bounded Model Checking (BMC) [2, 3, 4]. 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 such as GRASP [5] or Chaff [6]. If the formula is satisfiable, a counterexample can be extracted from the output of the SAT procedure. If the formula is not satisfiable, the state machine and its specification can be unwound more to determine if a longer counterexample exists. This process terminates when the length of the potential counterexample exceeds its completeness threshold (i.e., is sufficiently long to ensure that no counterexample exists [7]) or when the SAT procedure exceeds its time or memory bounds. BMC has been successfully used to find subtle errors in very large circuits [8, 9, 10].

The tool that we have developed, called `CBMC`, takes as input a C program and its Verilog implementation. The two programs are unwound in tandem and converted to a Boolean formula that is satisfiable if and only if the circuit and the code disagree. The formula is checked using a fast SAT procedure. If the two programs are inconsistent, a counterexample, which demonstrates the inconsistency, is generated or the tool exceeds its time or memory bounds. Multiple inconsistencies can be eliminated by running the tool several times.

The tool enables the user to customize the concept of ”consistency”. It enables cycle-accurate and non-cycle-accurate functional specifications, as well as more complex specifications that are realized by having the C code reference Verilog variables.

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

Related Work In [11], 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 [12]. Other variants of ANSI-C for specifying hardware are SpecC [13] and Handel-C [14].

The concept of verifying the equivalence of a software implementation and a synchronous transition system was introduced by Pnueli, Siegel, and Shtrichman [15]. The C program is required to be in a very specific form, since a mechanical translation is assumed.

The methodology presented in this report handles a large set of ANSI-C language features, including arbitrary loop constructs, and allows fully reactive programs and circuits. We also present optimizations for nested loops and add support for pointer type casts. We conclude with a number of examples and explain how the tool was used to verify them.

2 ANSI-C Programs for Hardware Specification

In this section we show how ANSI-C programs are used to specify the correct behavior of hardware designs. Our tool supports ANSI-C programs, defined according to the ANSI-C standard [16]. However, since we know that the programs are going to be used as (synchronous) hardware specifications we have also added a few functions that especially useful for non-cycle-accurate functional specification. These functions are implemented efficiently within the tool instead of being implemented by the user. We stress that the specifying program is written in standard ANSI-C, which means that it can also be executed. This is particularly useful for checking performance issues.

2.1 Connecting C and Verilog

The signals of the Verilog design exposed to the C program using name matching. A signal sig is made visible in the C program by declaring it as an external, constant, unbounded array. The ith element of this array represents the value of sig at clock cycle i. For now we assume a single clock, but the same ideas apply for multiple clock systems as well. The extension of the tool to multiple clock domains is currently under development.

The next step is to specify the values that signals should have. This is done using the "assert" statement, which is a part of the ANSI-C standard. Each assert statement translates into a formula to be proven. In Figure 1, the C code asserts that the signal sig toggles with each clock cycle.

This example shows a common feature of hardware specifying programs, where an integer variable is used to track the clock cycle being referred to (in our example it is called `cycle`). Such a variable appears in both cycle-accurate and non-cycle-accurate specifications. This is a normal C variable - it can be incremented or decremented by any number and can be used on the right hand side of any assignment. We discuss a few of the many possible specification styles in section 2.2.

The `CBMC` tool performs Bounded Model Checking, which means that we prove correctness for all possible test vectors up to a given bound. The bound is provided in a special variable called `CBMC_bound`. All the design signals, although declared as unbounded arrays, are in fact valid only up to cycle `CBMC_bound`. The example in Figure 1 shows how this variable is used to bound the loop that checks sig.

We now describe our extensions to the ANSI-C standard. Each of these functions has an equivalent MACRO definition that can be used when the program should be executed. However, for verification purposes our internal implementations are more efficient.

```
extern unsigned CBMC_bound;
extern const _Bool sigA[];

for (cycle=0; cycle < CBMC_bound; cycle++)
    assert (sigA[cycle] = !sigA[cycle+1]);
```

Figure 1: A program specifying that `sig` toggles with each clock cycle

`WAITFOR(cycle,property)` (where `cycle` is the name of an integer variable and `property` is an expression that uses the variable `cycle` to refer to clock cycles). This function returns the next value for `cycle` that makes `property` true. If `CBMC_bound` is reached before `property` holds `CBMC_bound+1` is returned. For example, assume that the program uses the variable cc as a clock cycle index. The line: `i = WAITFOR(cc, sig[cc] && !sig[cc - 1])` assigns to i the next clock cycle in which sig rises (changes from 0 to 1). The function starts with the current value of cc (any number in the range 0 to `CBMC_bound`) and checks the property on clock cycles $cc, cc+1$, etc. It returns the first value that makes the property true without changing cc.

`POSEDGE(cycle,property)` This function returns the cycle of the next positive edge of property. It is similar to `WAITFOR`, only it waits for the property to be false for at least one clock cycle before it becomes true.

`NEGEDGE(cycle,property)` This is the dual of `POSEDGE`, it waits for a negative edge on `property`.

`ASSERT_RANGE(cycle, i, j, property)` This function asserts a given property on a range of values for cycle. This is efficient shorthand for a loop that asserts `property` for `cycle=i` through `cycle=j`.

2.2 Specification Styles

We now discuss a few of the different specification styles that are possible using C programs, demonstrating the versatility of our tool. Throughout this section we use a running example of an imaginary Client-Server application. In this example there are two clients that communicate with a single server. The protocol between each client and the server includes a request for service ($c2s\_req\#$), a grant from the server ($s2c\_grant\#$), and an indication that the server has completed this request ($s2c\_done\#$). Figure 2 gives a block diagram of the example.

2.2.1 Property Assertions

We can use C to describe a simple assertion that is an invariant of the design. This is usually done using a "monitor". The C program monitors the design's progress and watches out for a specific type of error. An assertion is then used to claim that an error

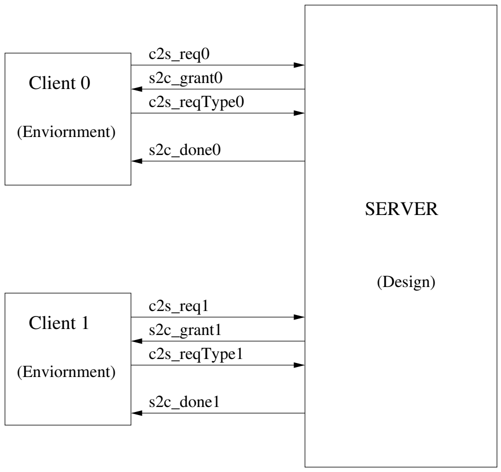

Figure 2: A Block Diagram for the Client-Server example

is never encountered. In our example, we may want to ensure that the server never grants two requests at the same clock cycle. This is checked using a single assertion, as follows:

```
for (cycle=0; cycle<=CBMC_bound; cycle++)
    assert(!(s2c_grant0[cycle] && s2c_grant1[cycle]));
```

A more interesting monitor would be one that asserts that the server will not grant a second request before it reports the previous request being done:

```
int cc=0;
_Bool busy_flag=0;

while (cc <= CBMC_bound) {
  cc = WAITFOR(cc, s2c_grant0 || s2c_grant1);
  if (cc < CBMC_bound) {
    i = WAITFOR(cc, s2c_done0 || s2c_done1);
    if (i < CBMC_bound)
      ASSERT_RANGE(cc, cc, i,
          !s2c_grant0[cc] && !s2c_grant1[cc]);
    cc = i;
  }
}
```

A more advanced specification is one that tracks temporal properties [17] of the design. One of the common criticisms of the use of temporal logic for specifications is that they can be difficult to understand and to write correctly. In the following we give a few examples of real life temporal properties and how they can be modeled using C. These examples were taken from the property database of the Accellera formal verification committee [18].

- **In English:** "If the "boff" signal is asserted, then if the first request which is accepted after the assertion of "boff" is not a snoop request, then it is a write request".

  In LTL: **G** (boff -> **X** (!accepted **W** (accepted & (!snoop_req -> write_req))))

  C Specification:

  ```
  for (cc=0; cc<=CBMC_bound ; cc++) {
   if (boff[cc]) {
    accept_c = WAITFOR(cc, accepted[cc])
    assert(!snoop_req[accept_c] =>
                  write_req[accept_c]);
   }
  }
  ```

  The for loop scans all clock cycles within the model checking bound. Whenever there is a `boff` signal at clock cycle *cc* the variable `accept_c` is assigned the next clock cycle in which `accepted` is 1 using a `WAITFOR` function call. We then assert on the clock cycle `accept_c` that *if* the `snoop_req` variable is 0 then the `write_req` variable must be 1.

- **In English:**"If a write command starts and size=N (N=1 through 8), then N assertions of signal "gx_start" should occur before the LAST bit goes active".

  In LTL:

  G ((start && cmd=write && size=1) ->
  !last **W** (gx_start && !last && **X** (!gx_start **W** last))) &&
  G ((start && cmd=write && size=2) ->
  !last **W** (gx_start && !last &&
  X (!last **W** (gx_start && !last && X (!gx_start **W** last)))))
  ... (for size=3 through size=8)

  C Specification The C specification uses the same code, no matter what `size` is. A counter variable called `counter` is initialized with N, and decremented every time `gx_start` is active. While `count` is greater than zero the program asserts that `last` is not active. Once it reaches zero, the program asserts that there will be no more `gx_start`'s before the next `last`.

```
cc = 0;
while (cc <= CBMC_bound) {
    cc = WAITFOR(cc, start[cc] && cmd[cc]==write);
    if (cc <= CBMC_bound) {
        count = size[cc];
        // correct number of gx_start before last
        for (i=cc+1; i<=CBMC_bound && count>0; i++) {
            assert(!last[i]);
            if (gx_start[i]) count--;
        }
        // no more gx_start before last
        for (; i<=CBMC_bound && !last[i]; i++)
            assert(!gx_startp[i]);
    }
}
```

In each pass through the while loop we jump to the next time there is `start` with the command being a write (using a `WAITFOR` function call). If the result is greater than `CBMC_bound` it means we reached the end of the bounded trace that is checked. Otherwise, we assign the variable `count` with the number of times `gx_start` needs to be 1 before we allow the signal `last` to be asserted. The first for loop scans the following clock cycles counting down the occurrences of `gx_start`, while asserting that `last` should be 0. After the correct number of `gx_start`'s, the second for loop makes sure that there are no extra `gx_start`'s before `last` is asserted.

- This example shows a bounded liveness specification. The CTL specification is a liveness formula and the C specification interprets this as a requirement that the eventuality will occur within the bound.

  In LTL: G (req -> (busy **U** ack))

  C Specification This examples shows a typical use of WAITFOR. The first WAITFOR brings us to the next time a request is made. The second one finds the first acknowledge after this request. If the aknowledge is not found within the bound the program reports an error, using `assert(false)`. Otherwise, the program asserts that between these two clock cycles the signal `busy` must be active.

```
cc = 0;
while (cc<=CBMC_bound) {
    cc = WAITFOR(cc, req[cc]);
    ack_c = WAITFOR(cc, ack[cc])
```

```
    if ((cc <= CBMC_bound) && (ack_c > CBMC_bound))
        assert(false); // Liveness error detected
    else
        ASSERT_RANGE(cc, cc, ack_c-1, busy[cc]);
    c++;
    }
}
```

In each iteration through the while loop `cc` is assigned the next clock cycle in which there is a request, and `ack_c` is assigned the next clock cycle after that in which there is an acknowledge. Both of these are done using `WAITFOR`, and both variables will get the value `CBMC_bound` if no such event occurs. If a request is found (`cc <= CBMC_bound`) but an acknowledge does not occur until the end of the trace (`ack_c > CBMC_bound`) an error is detected. Otherwise, we assert that between the request and acknowledge the `busy` signal must be 1.

2.2.2 Functional Specification

A functional specification is one in which the C program specifies the full functionality expected from the design, as opposed to having assertions that only track inputs and specify constraints on the outputs. In functional specification we create a software implementation of the design. Since we are using ANSI-C, the program can actually be executed for performance evaluation and other simulation techniques.

One of the advantages of using ANSI-C for specification is that we can exploit the flow of control to differentiate between different tasks that the design needs to perform, instead of relying on a "state" variable. An example of this is shown in Figure 3. This code is a skeleton of a program that is used to give a cycle accurate specification of the Server in our Client-Server application from figure 2.

Variable names that start with `my_` are the local C versions of the output signals of the design. They are defined as arrays with length `CBMC_bound` so that each entry is assigned the proper value at that clock cycle. In a subsequent part of the program, not seen in the figure, we assert that the values of the local C version and the verilog version are identical for each clock cycle. This approach is particularly suited when we have a cycle accurate specification.

If we do not have a cycle accurate specification we can change our program so that instead of setting a specific cycle in which an event must occur, it will wait for events to happen in the system and check subsequent behavior.

In the example above we used arrays to store the value for each output at each clock cycle. This turns out to be rather wasteful. Instead, we could use a single boolean variable for each boolean output of the design, assign it the correct values, and then compare the values whenever the cycle variable is incremented. For this scheme we use the following function:

```
int inc_and_assert(cycle, /* signals to compare /*){
    assert(my_sig == sig[cycle]);
```

```
while (cycle <= CBMC_bound) {

 // Look for the next request
 cycle = WAITFOR(cycle,c2s_req0[cycle]||c2s_req1[cycle]);

 // Choose a client if both of them request at the same time
 granted_client = arbitration_policy(c2s_req0[cycle],c2s_req1[cycle]);

 // set the proper value to the ``granted'' outputs
 if (granted_client=0) {

   my_s2c_grant0[cycle] = 1;
   service_type = c2s_reqType0[cycle];
   cycle++;
   my_s2c_grant0[cycle] = 0;     // grant is a pulse

 } else {

   my_s2c_grant1[cycle] = 1;
   service_type = c2s_reqType1[cycle];
   cycle++;
   my_s2c_grant1[cycle] = 0;     // grant is a pulse
 };

 // Proceed according to the type of service requested.
 switch (service_type) {
   case READ : // implementation of READ command that
               // may span many clock cycles
                 ...
               break;

   case WRITE : // implementation of WRITE command
                 ...
                break;
 }

 // Terminate the service using the ``done'' output
 if (granted_client=0) {

   my_s2c_done0[cycle] = 1;
   cycle++;
   my_s2c_done0[cycle] = 0; // done is a pulse

 } else {

   my_s2c_done1[cycle] = 1;
   cycle++;
   my_s2c_done1[cycle] = 0; // done is a pulse
 };
}
```

Figure 3: Skeleton of a cycle-accurate specification of the Server from Figure 2

```
        // do this for each signal
    return(cycle+1);
}
```

2.3 Assumptions and Assume-Guarantee Reasoning

By default, all inputs to the design are assumed to have non-deterministic values, and the tool checks the assertions for all possible combinations of input values. An `assume` statement constrains the values of the inputs, enabling the user to create an environment for the design using ANSI-C. The "assume" statement limits the state space to only those computations that adhere to the given constraint. For example, to assume that `sig` is a pulse, i.e., it cannot be 1 for two consecutive clock cycles, we use the following line within a loop that increments `cycle`:

```
assume(sig[cycle-1] => !sig[cycle]);
```

Assume statements can also be used to abstract away parts of the design and thus support an assume-guarantee style reasoning [19]. When the implementation of a module within the design is removed, its outputs are considered inputs to the design and are thus non-deterministic. A small C program can be written to create assumptions that constrain these signals to provide properties on the rest of the design. Later, the "assume" and "assert" statements are interchanged to check the assumptions on the module that was abstracted away. This is most useful when the full design is too large to be checked, or when parts of the design are missing.

3 Transforming ANSI-C into a Bit Vector Equation

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 model the behavior of ANSI-C programs according to the ANSI/ISO C 99 standard [16]. We assume that the ANSI-C program is already preprocessed, i.e., all `#define` directives are expanded. We then perform a series of transformations on the program so that in the end we have a single assignment program that uses only branching and assignment statements. These transformations are inter-dependent, one transformation may result in the need to use the other, so we perform them iteratively until no more transformations are needed. Sections 3.1 through 3.5 describe all of our transformations. Finally, we use the resulting program to create a set of bit-vector equations. This process is described in Section 3.6.

One of the most challenging features we need to deal with is the use of pointers and dynamic memory allocation. Because of the complexity of handling these features we ignore them in this section and devote the whole of Section 4 to them.

3.1 Preparing the Translation

We first perform a series of transformations that remove several ANSI-C commands by transforming them into equivalent if, goto, and while commands.

1. The instructions `break` and `continue` are replaced by semantically equivalent goto instructions as described in the ANSI-C standard [16]. The `switch` and `case` instructions are replaced by semantically equivalent code using `if` and `goto` instructions.

2. The `for` instructions are replaced by `while` instructions as follows (I is a single statement or a block):

$$\texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ I \quad\longrightarrow\quad e_1\texttt{;}\ \texttt{while(}e_2\texttt{)}\ \{\ I\texttt{;}\ e_3\texttt{;}\ \}$$

3. The `do while` instructions are replaced by `while` instructions as follows:

$$\texttt{do}\ I\texttt{;}\ \texttt{while(}e\texttt{)} \quad\longrightarrow\quad I\texttt{;}\ \texttt{while(}e\texttt{)}\ I\texttt{;}$$

3.2 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. These three cases are handled as follows:

1. The `while` loops are unwound using the following transformation n times:

$$\texttt{while(}e\texttt{)}\ I\texttt{;} \quad\longrightarrow\quad \texttt{if(}e\texttt{)}\ \{\ I\texttt{;}\ \texttt{while(}e\texttt{)}\ I\texttt{;}\ \}$$

   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. 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*.

$$\texttt{while(}e\texttt{)}\ I\texttt{;} \quad\longrightarrow\quad \texttt{assert(!}e\texttt{);}$$

   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.

2. Function calls are expanded. Recursive function calls are handled in a manner similar to while loops: the recursion is unwound up to a certain 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. Further details about function call expansion are given in Section 3.4.

3. Backward `goto` statements are unwound in a manner similar to `while` loops.

```
if (a <= 0) {
  a = a + 1;
  b = b - 1;
}
c = a + b;
```

$\xrightarrow{\rho}$

```
if (a0 <= 0) {
  a1 = a0 + 1;
  b1 = b0 - 1;
}
c1 = a1 + b1;
```

Figure 4: Example: Renaming program variables in order to remove duplicate assignments. The result is a program in SSA form.

3.3 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). To make the program a *single assignment* program we use variable renaming. 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 into 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. Figure 4 shows an example of variable renaming. The result is similar to a Static Single Assignment (SSA) program [20] without ϕ-functions.

3.4 Side effects

Side effects, i.e., pre- and post-increment operators, the assignment operators, and function calls, are removed by introducing new temporary variables (Section 4 describes how pointer dereferences are handled). This is done recursively, starting with the inner most side effects. Let e denote a side effect expression, e.g., `x++`. By $\sigma(e)$ we denote the expression after the removal of the side effect. Furthermore, all side effects require additional code that is to be inserted before the statement that contains the side effect. We denote this additional code by $\Sigma(e)$. Note that $\Sigma(e)$ might again contain side effects. Thus, it might be necessary to perform the side effect removal multiple times. Also, the created code may require further renaming, as described in Section 3.3.

The functions $\sigma(e)$ and $\Sigma(e)$ are defined by a case split on the type of the side effect in e.

- Let the side effect be a pre-increment or pre-decrement operator, i.e.,

$$e = op\ e'$$

where *op* is one of ++ or −−, and e' is a (side effect free) expression. In this case, the expression *op* e' is simply replaced by e'.

$$\sigma(op\ e') := e'$$

The side effect is performed using Σ:

$$\Sigma(op\ e') := \{\ \texttt{e' = (e' op' 1);}\ \}$$

where op' is $+$ in case of $op = $ ++, and $-$ otherwise.

For example,

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

is transformed to

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

- Let the side effect be a post-increment or post-decrement operator, i.e.,

$$e = e'\ op$$

where *op* is one of ++ or −−, and e' is a (side effect free) expression. In this case, the expression e' *op* is replaced by a new variable of the same type. Let t denote this variable.

$$\sigma(e'\ op) := t$$

The code inserted before the statement with the expression, as defined by Σ, initializes the variable t with the value of the expression and then performs the side effect:

$$\Sigma(e'\ op) := \{\ \texttt{t = e; e' = (e' op' 1);}\ \}$$

where op' is $+$ in case of $op = $ ++, and $-$ otherwise.

For example,

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

is transformed to

$\texttt{t}_1\texttt{=i}_0\texttt{;}$
$\texttt{i}_1\texttt{=i}_0\texttt{+1;}$
$\texttt{x}_1\texttt{=5+t}_1\texttt{;}$

where `t` is a new variable of the same type as `i`.

- In case of function calls, a new variable is introduced that has the same type as the return type of the function. Let t denote this variable. Let f denote the function expression, and $a_1, \ldots, a_n$ denote the arguments. Both f and the arguments are side effect free (all side effects within f have already been removed).

$$\sigma(f(a_1, \ldots, a_n)) := t$$

$\Sigma(f(a_1,\ldots,a_n))$ is defined to be the function body of f where appropriate variable renaming is applied in order to preserve locality. Furthermore, every return statement is replaced by an assignment to t and a `goto` to the end of the function. Note that the function body itself might contain further side effects, including recursive calls to the same function. The recursion depth is limited using unwinding assertions, as done for `while` loops.

Side effect operators that were not mentioned above are the assignment operators `+=` , `-=` , `*=` ,.... These are actually shorthands for an increment and assignment and they are opened according to the ANSI-C standard. For example, `a += 1` is transformed into `a = a + 1`.

The code Σ, which is inserted before the statement with the side effect expression, must be guarded in case the expression makes use of the operators `?:`, `&&`, or `||`. For example,

```
x=5+c?(++i):0;
```

is transformed into

```
if(c) i=i+1;
x=5+c?i:0;
```

The algorithm presented above is used in conjunction with the renaming process described in Section 3.3. The side effect removal algorithm and the renaming algorithm are used iteratively until there is no more to be done. To see why this is needed, consider a similar example to the one above, in which `++i` is changed to `++c`:

```
x=5+c?(++c):0;
```

A naive transformation results in:

```
if(c) c=c+1;
x=5+c?c:0;
```

which obviously is incorrect. The renaming process will ensure that the two uses of c are transformed using different variables:

$$\begin{aligned} c_1 = c_0?c_0+1:c_0 \quad &\wedge \\ x_1 = 5+c_0?c_1:0 \end{aligned}$$

This is justified as follows: the evaluation of `c` in the condition is done *before* the side effect, and thus is renamed to c_0. The value of `++c` is the value after the assignment, and thus, is renamed to c_1.

Note that the ANSI-C standard allows multiple evaluation orderings for side effects. Thus, all allowed orderings have to be verified. This is done by generating a version of the program for all possible orderings. We then compare the generated equations for equivalence. The number of orderings is potentially exponential.

```
if(a0) {
 if(b0)
   goto label;

 b1=z0;
}
y1=z0;
label:;
```

$\longrightarrow$

```
if(a0) {
 if(b0) ;

 if(!(a0 && b0))
   b1=z0;
}
if(!(a0 && b0)
 y1=z0;
label:;
```

Figure 5: Example: Transforming `goto` to `if`

3.5 Eliminating Goto Commands

After renaming, forward `goto` statements are changed into equivalent `if` statements. Let x denote the part of the program p before the label and y denote the part of p after the label l, i.e., $p = x\,l : y$. An `if` statement is added as guard to all statements in x. The condition of the `if` statement is the conjunction of the conditions guarding the `goto` statement. Figure 5 shows an example of this transformation. Note that this does not allow `goto` statements with a target inside a guarded block.

3.6 Creating Bit Vector Equations

At this point we have a program that is in single assignment form and consists of only assignment statements and conditionals. We create a a bit-vector equation $\mathcal{C}$ that forms the set of constraints and a bit-vector equation $\mathcal{P}$ that represents the set of properties, i.e., the assertions.

The final transformation is done using the functions $\mathcal{C}(p, g)$ and $\mathcal{P}(p, g)$. Both take a program p and a guard g as argument and map this to an equation. The first function $\mathcal{C}$ computes the constraints (assumptions), and the second function $\mathcal{P}$ computes the properties (assertions). Both functions are defined by induction on the syntax of the statement p.

Skip. If p is empty or skip, both the constraint and property are true.

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

Conditional. Let p be an `if` statement with condition c, and code blocks I and I'. The functions are used recursively for both code blocks. For I, $\rho(c)$ is added to the guard, and for I', $\neg\rho(c)$ is added to the guard. The two constraints and claims provided by $\mathcal{T}$ are conjoined.

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

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

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

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

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

The two sub-programs I and I' are processed with the same guard because they both run under the same conditions, i.e., I will be executed iff I' is executed (since there are no more `goto` statements in the program). Consider for example the short program from Figure 4 (after renaming, page 11). Let P be the `if` statement in this program (without the last assignment into c). Processing the `if` statement gives us:

$$\mathcal{C}(P, \mathsf{true}) \quad = \quad \mathcal{C}(\texttt{"a}_1 \texttt{ = a}_0 \texttt{ + 1; b}_1 \texttt{ = b}_0 \texttt{ - 1",}\ a_0 \leq 0)$$

The program is a single assignment program, so the body of the "if" statement cannot influence the value of variables that appear in the guard (a_0 in this case). Also, the first assignment and second assignment are always run under the same conditions - either they are both run, or they are both not run. This is why we process them using the same guard:

$$\begin{gathered}\mathcal{C}(\texttt{"a}_1 \texttt{ = a}_0 \texttt{ + 1; b}_1 \texttt{ = b}_0 \texttt{ - 1"}, a_0 \leq 0) \equiv \\ \mathcal{C}(\texttt{"a}_1 \texttt{ = a}_0 \texttt{ + 1"}, a_0 \leq 0) \quad \wedge \quad \mathcal{C}(\texttt{"b}_1 \texttt{ = b}_0 \texttt{ - 1"}, a_0 \leq 0) \equiv \\ a_1 = (a_0 \leq 0 ? a_0 + 1 : a_0) \quad \wedge \quad b_1 = (a_0 \leq 0 ? b_0 - 1 : b_0)\end{gathered}$$

The last assignment $\texttt{c}_1 \texttt{ = a}_1 \texttt{ + b}_1$ generates the following additional constraint:

$$c_1 = a_1 + b_1$$

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

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

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

Assignment $v = e$. The assignment is returned as an equality constraint. Note that the variables in the constraint are renamed as described above. Let the value of the variable after the assignment be v_α. The value before the assignment is then $v_{\alpha-1}$. If v is a simple variable, i.e., not of an array or struct type, 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.

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

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

If v is of an array type, let a be the array index address, i.e., the assignment is $v[a] = e$. We add a constraint as follows: 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. We model arrays as functions and use lambda notation.

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

```
x=x+y;
if(x!=1) {
  x=2;
  if(z) x++;
}
assert(x<=3);
```

$\rightarrow$

```
x1=x0+y0;
if(x1!=1) {
  x2=2;
  if(z0) x3=x2+1;
}
assert(x3<=3);
```

$$\rightarrow \quad \begin{array}{lll} \mathcal{C} & := & x_1 = x_0 + y_0 \;\wedge \\ & & x_2 = ((x_1 \neq 1)?2 : x_1) \;\wedge \\ & & x_3 = ((x_1 \neq 1 \wedge z_0)?x_2 + 1 : x_2) \\ \mathcal{P} & := & x_3 \leq 3 \end{array}$$

Figure 6: 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 3.6.

Note that we use the lambda notation in a very restricted way. The variable used has always a simple type (the index type), and is never instantiated using another function.

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

After the computation of $\mathcal{C}$ and $\mathcal{P}$ using the algorithm above, we verify that $\mathcal{C} \Longrightarrow \mathcal{P}$ is valid. This proves that no unwinding assertions have been violated and that all array bounds are obeyed. Figure 6 shows a simple example of the transformation process.

4 Pointers

4.1 Dereferencing Pointers

Pointers are commonly used in ANSI-C programs. This even applies to ANSI-C programs that are a representation of a circuit. In particular, pointers are required for call by reference and for arrays.

During the unwinding phase, but before the variable renaming, all pointer dereferences are removed recursively as follows: The first step is to simplify expressions of the form `&*p` to `p`. Note that this allows ANSI-C constructs such as `p=&*NULL` (this is guaranteed not to cause an exception).

In the second step, the remaining dereferencing operators are removed. Let e denote the sub-expression that is to be dereferenced. We remove dereferencing operators bottom-up, i.e., all sub-expressions of e are already free of dereferencing operators or other side effects. Let g denote the guard as described above, and o the offset. 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 two dereferencing operators: The star operator and the array index operator. Both are replaced by the expression provided by ϕ. The star operator uses offset zero.

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

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

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 8.

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

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

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

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

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. As above, we check type consistency: the type of the array elements of the array a has to match the type T.

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.

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

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.

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

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

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

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

8. In any other case, the ANSI-C standard does not define semantics. As example, e might be the `NULL` pointer or a pointer variable that is uninitialized. We use an error value in this case and we assert that this dereferencing is never done by the program. This is implemented by adding an assertion that $\rho(g)$ does not hold. Let $\perp$ be the error value.

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

The algorithms for the difference of two pointers $p-q$ or the relation between two pointers, e.g., p¿=q, are similar. We assert that p and q point to the same object, as required by the ANSI-C standard, and then use the difference between the offsets.

Example 1: Consider the code fragment

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

The first statement is transformed into:

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

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

$$\begin{array}{rcll}
*p & = & \phi(p,\textsf{true},0) & \\
 & = & \phi(x_0?p_1:\&b,\textsf{true},0) & \text{because of } \rho(p)=p_2 \text{ (case 1)} \\
 & = & x_0\,?\,\phi(p_1,x_0,0):\phi(\&b,\neg x_0,0) & \text{(case 5)} \\
 & = & x_0\,?\,\phi(x_0?\&a:p_0,x_0,0):b & \text{because of } p_1=(x_0\,?\,\&a:p_0) \\
 & = & x_0\,?\,(x_0?\phi(\&a,x_0,0):\phi(p_0,x_0\wedge\neg x_0,0)):b & \text{(case 5)} \\
 & = & x_0\,?\,(x_0?a:\phi(p_0,x_0\wedge\neg x_0,0)):b & \text{(case 3)}
\end{array}$$

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

Example 2: It is common practice to use the fixed evaluation ordering of the Boolean operators to guard pointer dereferences. 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 the following bit-vector equations:

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

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

$$\begin{array}{rcll} *p & = & \phi(p, p \neq \texttt{NULL}, 0) & \\ & = & \phi(x_0?p_1 : \texttt{NULL}, p \neq \texttt{NULL}, 0) & \text{because of } \rho(p) = p_2 \text{ (case 1)} \\ & = & x_0?\phi(p_1, p \neq \texttt{NULL} \wedge x_0, 0) : \phi(\texttt{NULL}, p \neq \texttt{NULL} \wedge \neg x_0, 0) & \text{(case 5)} \\ & = & x_0?\phi(\&a, p \neq \texttt{NULL} \wedge x_0, 0) : \bot & \text{(case 1, case 7)} \\ & = & x_0?a : \bot & \text{(case 3)} \end{array}$$

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

Example 3: Consider the code fragment

```
int a[n], *p;
p=&a[5];
p[1]=1;
```

The first assignment statement is transformed into $p_1 = \&a[5]$. The index operator in the second assignment statement is removed as follows:

$$\begin{array}{rcll} p[1] & = & \phi(p, \mathsf{true}, 1) & \\ & = & \phi(\&a[5], \mathsf{true}, 1) & \text{because of } \rho(p) = p_1 \\ & = & a[5+1] & \text{(case 4)} \end{array}$$

This is then transformed into a λ constraint as described above.

4.2 Dynamic Memory Allocation

We allow programs that make use of dynamic memory allocation, e.g., for dynamically sized arrays or data structures such as lists or graphs. This is realized by replacing every call to `malloc` or `calloc` by the address of a new variable of the desired type and size. For this, we assume that the type of the new variable is given by either an explicit or implicit type cast to a pointer that points to a variable of type t. In case of `malloc`, let x be a new variable of an array type with elements of type t and size s divided by `sizeof(`t`)`. We assert that s is an integer multiple of `sizeof(`t`)`; thus the result of the division is always an integer.

$$(t\ *)\ \texttt{malloc}(s) \quad \longrightarrow \quad \&x$$

```
while (B1) {
  S1
  while (B2)
{
      S2
  }
  S3
}
```

→

```
if (B1) {
  S1
  Full unwinding of inner loop
  S3

  if (B1) {
    S1
    Full unwinding of inner loop
    S3
    ...
  }
}
```

Figure 7: A double nested loop and its unwinding. S_1, S_2, and S_3 are arbitrary loop free sub-programs.

In case of `calloc`, let n denote the number of elements to be allocated and s denote the size of each element. We add the assertion that $s = \texttt{sizeof}(t)$ holds. Let x be a new variable of an array type with elements of type t and size n.

$$(t\ \texttt{*})\ \texttt{calloc}(n,\ s) \quad \longrightarrow \quad \texttt{\&}x$$

In order to prohibit access to a dynamically allocated object after deallocation, a single bit variable is added for each `malloc` statement. The `malloc` statement sets this variable to true, while the `free` statement sets it to false. The `free` statement determines the object pointed to by the pointer using the dereferencing algorithm described above. The bit is used to check whether the object created by `malloc` has exceeded its lifetime. This also allows verifying the absence of memory holes by asserting that all these variables are false at the end of the program.

5 Nested Loops

Nested loops within the ANSI-C code can result in extremely large CNF formulas. This is because of the unwinding process, where for every unwinding of an outer loop we unwind the inner loop in full. Figure 7 shows the resulting program after unwinding two nested loops. If we had three nested loops, which is not that common but can occur, matters would be even worse.

We attempt to alleviate this problem by taking advantage of the fact that our programs are hardware specifications and that we are performing *bounded* model checking. We notice that many C programs that specify synchronous hardware designs contain a "`cycle`" variable that is used to refer to the clock cycle in which an event occurs. This variable is typically incremented within each while loop, to signify the passing of one or more clock cycles. Since our tool performs bounded model checking, the user has access to a variable called "`CBMC_bound`" that holds the bound used for a particular run. The C program is not allowed to access the value of a design signal at a clock

```
while ((vpc != 1) || B1) {
  if (vpc == 1) {
    S1
    vpc = 2
  } else if (vpc == 2) {
    if (B2) {
      S2
    } else {
      vpc = 3
    }
  } else if (vpc == 3) {
    S3
    vpc = 1
  }
}
```

Figure 8: The double nested loop from Figure 7 after transformation

cycle that is greater than this bound. Therefore, we expect the programmer to insert a condition on the cycle variable of always being less than or equal to `CBMC_bound`. These two observations lead us to expect that in many cases there will be a constant bound on the number of iterations each loop can be executed, and that this bound would be in the order of the bounded model checking constant `CBMC_bound`.

We use a program transformation that transforms a nested loop construct into a single loop. We do this by partitioning the body of the outermost while loop into subprograms, labelling the sub-programs, and then adding a virtual program counter variable that keeps track of which sub-program should be executed next. This process is demonstrated in Figure 8 for the program from Figure 7. The program variable `vpc` is added to indicate which part of the original program should be executed next. If `vpc==1` then the first part of the outer loop is executed (S_1), if `vpc==2` the nested loop is executed, and if `vpc==3` the third part of the outer loop is executed (S_3). This transformation can easily be extended to a nested loop construct with more than two loops.

As mentioned above, this transformation will only be useful in specific cases. We apply it only if there exists an integer variable in the program that is incremented in every loop body and is checked to be less than or equal to some bound within every loop condition. In the following we give an analysis that shows how the transformation can significantly improve performance.

Let n be the bound of the bounded model checking algorithm, i.e., our tool is requested to compare the specifying program, with the unwinding of n clock cycles of the Verilog design. To analyze the complexity of a program, we compare the size of the unwound program with the model checking bound. The reason is that we expect the program to be able to assume/assert on every clock cycle within the bound, thus the most efficient program would be one in which the process of unwinding loops will replicate each statement $O(n)$ times.

Figure 7 gives a general structure for two nested loops. We evaluate the length of the unwound program by counting the number of times S_1 and S_2 are replicated (the number for S_3 is identical as for S_1). As mentioned in Section 3, our tool will unwind each loop at least the maximum number of times that it might be iterated. If we choose a number that is too low the assertion associated with this loop will fail, and the loop will be further unwound. Let k be the maximal number of times that the outer loop can be executed, and let l_i ($i = 1, \ldots, k$) be the maximal number of times the inner loop can be executed in the ith iteration through the outer loop. The unwinding algorithm is guaranteed to unwind the outer loop *at least* k times, and unwind the ith copy of the inner loop *at least* l_i times. In this unwinding S_1 is replicated k times, and S_2 is replicated $\sum_{i=1}^{k} l_i$ times. Now, assuming that the `cycle` variable is incremented (by at least 1) in both S_1 (or S_3) and S_2, we get that $l_i \leq n - i$, so S_1 is replicated $O(n)$ times and S_2 is replicated $O(n^2/2)$ times.

In the transformed program we have a single loop. The body of this loop is larger than the body of the original outer loop by some constant number of statements. However, under the same assumption that both S_1 (or S_3) and S_2 increment `cycle` we get that the whole loop needs only to be unwound $O(n)$ times, so each sub-program is replicated $O(n)$ times. This reduction from n^2 to n can significantly effect the performance of the tool. However, it also entails a larger multiplicative constant, so it is not recommended for extremely small programs. Applying the transformation to a triplly nested loop construct will achieve an even greater performance boost, although these cases are more rare.

Obviously, the question arises of whether the conditions we impose on nested loops are too strict. We suggest that C programs that are written for specifying hardware designs will naturally use a `cycle` variable, since without it, it is very difficult to assert properties on the design. If such programs are to be used in a bounded model checking setting, it is natural to assume that the user will insert a check on the bound to each loop condition, or otherwise an error may be generated during model checking. Finally, we note that in all the examples in which we wrote the C specification ourselves all of our loops satisfied the conditions for making the transformations, including specifications that were *not* cycle-accurate.

6 Transforming Verilog

6.1 Single Clock Designs

We only consider a restricted subset of the Verilog language [21]. Delay or event specifiers are ignored and only register data transfers are converted. Such a language is called synchronous register transfer language (RTL). The abstract data types real and time are not allowed. The process of translating the Verilog design closely resembles the process of synthesis of behavioral Verilog into a netlist.

The first step of the translation is to determine the variables that form the state of the circuit, i.e., the latches. It is a common design practice to specify registers in Verilog that are not intended to become part of the state. This allows us to define combinatorial behavior using assignment statements. The tool uses the following heuristic to distinguish a latch from a register that is only used for syntactic reasons: It is required that all assignments are guarded using an event guard. A Verilog register is translated into a latch if this event guard contains a clock event (`posedge` or `negedge`). Otherwise, the logic is treated like combinatorial logic. This allows defining the state of the circuit.

Example Consider the following Verilog example:

```
module main(clk);

input clk;

reg [31:0] latch;
reg [31:0] pseudo_latch;

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

always @(latch)
  if(latch & 1)
    pseudo_latch=0;
  else
    pseudo_latch=latch >> 1;

endmodule
```

This example contains two register declarations: `latch` and `pseudo_latch`. However, only `latch` will actually be part of the state. The register `pseudo_latch` will be considered combinatorial logic.

Let s and s' denote states of the circuit. The Verilog design is then translated into an initial state predicate and a transition relation. Synthesis tools usually do not convert initial state predicates. The initial state predicate $\mathcal{I}(s)$ holds if s is a valid initial state. The transition relation $\mathcal{R}(s, s')$ is a bit vector equation that holds if a transition from state s to state s' is allowed. Let e denote an expression. The value of an expression e using variable values in a given state s is denoted by $s(e)$.

The second step of the translation is to unwind all repetition statements. These are `for`, `while`, and `repeat`. In contrast to the unwinding done for ANSI-C, we assume that the truth value of the loop condition (and thus the number of iterations) can be determined statically. Any `case` statements are translated into equivalent `if` statements. Module instantiations are expanded using variable renaming to maintain locality.

The third step is to translate the remaining program into a set of constraints. This transformation is done by a graph traversal on the parse tree. The top level of the program only consists of `always` and `initial` blocks (i.e., behavioral constructs) and continuous assignments. The algorithm maintains a substitution map ϱ that maps a variable name to an expression that represents the current value of the variable. By $e[\varrho]$

we denote the expression e with all substitutions applied that the mapping function ϱ specifies. Initially, we start with $\varrho(v) = v$ for all variables v.

Let I denote the current assignment during the parse tree traversal. The algorithm proceeds by induction on the structure of the program I.

Continuous Assignment Let I be a continuous assignment. Let v be the variable on the left hand side, let e be the expression on the right hand side. We add the assignment as an equality constraint to the transition relation and the initial state predicate.

$$\mathcal{R}'(s,s') \quad := \quad \mathcal{R}(s,s') \wedge s(v) = s(e) \wedge s'(v) = s'(e)$$

Behavioral constructs Let I be an `always` or `initial` block. Let I be the current assignment in the block, and let g be the guard of I (i.e., the conjunction of the conditions in the `if` statements).

- Let I be a blocking `always` assignment to a latch v. The variable on the left hand side of the assignment is a variable of the *next state* s'. The variables on the right hand side are replaced using the current value function. The current value of v is changed to the right hand side.

$$\begin{aligned} \mathcal{R}'(s,s') \quad &:= \quad \mathcal{R}(s,s') \wedge (s(g) \Longrightarrow s'(v) = s(e[\varrho])) \\ \varrho'(x) \quad &:= \quad \begin{cases} e[\varrho] & : \quad x = v \\ \varrho(x) & : \quad \text{otherwise} \end{cases} \end{aligned}$$

- Let I be a non-blocking `always` assignment to a latch v. The variable on the left hand side of the assignment is a variable of the *next state* s', the variables on the right hand side need to be adjusted using the current value function ϱ, as above. In contrast to blocking assignments, non-blocking assignments do not adjust the current value mapping function ϱ.

$$\mathcal{R}'(s,s') \quad := \quad \mathcal{R}(s,s') \wedge (s(g) \Longrightarrow s'(v) = s(e[\varrho]))$$

- Let I be a blocking or non-blocking `initial` assignment to a latch. The variables on the left hand side of the assignment are variables of the initial state. On the right hand side, we expect a constant expression.

$$\mathcal{I}'(s) \quad := \quad \mathcal{I}(s) \wedge (s(g) \Longrightarrow s'(v) = e)$$

For Bounded Model Checking, the transition relation $\mathcal{R}$ obtained from the Verilog file is then unwound. In contrast to the unwinding done for ANSI-C, the number of times the unwinding must be specified manually. Let n be this number. Let $s_0, \ldots, s_n$ denote the states such that

$$\mathcal{I}(s_0) \wedge \bigwedge_{i=0,\ldots,n-1} \mathcal{R}(s_i, s_{i+1})$$

holds.

6.2 Multiple Clock Designs

The translation described above assumes that the design is governed by a single clock. Verilog provides extensive support to model designs that utilize multiple clocks. The translation described above would merge these clocks and therefore hide possible behavior. CBMC supports a safe abstraction that adds behavior in the case of multiple clocks rather than hiding it. The clock that is used for a latch is assumed to be given as an event guard. Consider the following Verilog module:

```
module main(clk1, clk2);

input clk1, clk2;

reg [31:0] latch1;
reg [31:0] latch2;

initial latch1=0;
initial latch2=0;

always @(posedge clk1) latch1=latch1+1;
always @(posedge clk2) latch2=latch2+1;

endmodule
```

It takes two clock signals as input. Upon a positive edge of clk1, latch1 is incremented, and upon a positive edge of clk2, latch2 is incremented. Assuming no further knowledge on the clock signals, the Verilog standard allows all interleavings. However, the translation above would synchronize the two processes; the incrementing would always be done synchronously. Since the latches are both initialized with zero, they will always have the same value. This is hiding possible behavior.

This problem is mended by adding the event guards to the guard of the assignments. Since the clocks are free and unconstrained inputs, this will allow all possible interleavings. For the example above, the transition relation is:

$$\begin{aligned}\mathcal{R}(s,s') \quad :\Longleftrightarrow \quad & s(clk1) \Longrightarrow s'(latch1) = s(latch1) + 1 \wedge \\ & s(clk2) \Longrightarrow s'(latch2) = s(latch2) + 1 \wedge \\ & /s(clk1) \Longrightarrow s'(latch1) = s(latch1) \wedge \\ & /s(clk2) \Longrightarrow s'(latch2) = s(latch2)\end{aligned}$$

This approach also allows clock signals that are derived from external clocks (i.e., inputs).

7 Translation to SAT Instance

Both the ANSI-C program and the Verilog circuit are unwound. This results in a bit vector equation for both the circuit and the program. In order to compare them, we translate them into a SAT instance.

As preparation for the translation, λ expressions are simplified. Consider the following code fragment:

```
int a[n], x, y, z;
a[x]=1;
z=a[y];
```

This is translated into:

$$a_1 \;=\; \lambda i : \, i = x_0 \,?\, 1 : a_0[i] \;\wedge\; z_1 \;=\; a_1[y_0]$$

A simplistic translation of this equation into CNF allocates literals for the whole lambda expression and then selects the bits that correspond to the element y_0. In order to reduce the size of the CNF, we simplify this equation using the following rule:

$$(\lambda i : e)[x] \quad \longrightarrow \quad e[\text{substitute } i/x]$$

The example expression is simplified to:

$$z_1 \;=\; (y_0 = x_0) \,?\, 1 : a_0[x_0]$$

The translation of the bit vector equation for the basic Boolean operators is done by adding new variables rather than using the law of distribution. The other bit vector operators are translated as follows:

- Bit vector addition, subtraction, and the relational operators $<$, $>$, $\leq$, $\geq$ are transformed into a Boolean equation using a carry chain adder.
- Bit vector multiplication is translated into a cost optimized multiplication circuit.
- The shifting expressions are translated using shifting circuits.
- All remaining lambda expressions (for arrays) are expanded.
- The array index operator for a variable index is replaced by a vector of new literals. Let v denote this vector. Let a denote the bit vector for the array, and x denote the index expression. Let s denote the number of elements in the array. We add constraints as follows:

$$\bigwedge_{i=0,\ldots,s-1} (x = i) \Longrightarrow (v = a[i])$$

Since we are interested in validity rather than satisfiability, the equation is negated as last step.

8 Experiments

We have run our tool on several examples. It should be noted that no optimization techniques, such as Bounded Cone of Influence, have been applied. We describe our examples in different levels of detail, and give performance information.

8.1 DES Encryption Standard

This example includes a software and hardware implementation of the DES encryption standard. The software implementation is taken from `libdes`, which is used in most Unix systems and in popular security software such as `ssh`. It is written in ordinary ANSI-C and is hand-optimized for performance. It therefore makes extensive use of macros, bit vector arithmetic, table lookups, pointers, and data structures that mix structures and arrays. It is 930 lines long.

On a 1.5 GHZ AMD Athlon machine the translation of the ANSI-C program into a SAT instance takes 1 minute and 49 seconds, 321 assertions are generated, 256 remain in the SAT instance after simplification. The exact number of iterations of all loops is determined statically by the tool. The full SAT instance including all bounds checks consists of 300,000 variables and 1,4 million clauses. The SAT checker Chaff [6] detects it to be unsatisfiable within 26 seconds. This proves validity of the original equation.

The hardware implementation is written in synchronous Verilog. There is a sequential (cost optimized) and a pipelined (speed optimized) version of the circuit. Currently, we only verify the sequential version. It is 1900 lines long. In order to unwind the hardware implementation, the number of steps has to be specified manually. For this example, 16 unwinding steps are required. The variable names have to be mapped manually to the corresponding variables in the C program.

8.2 Instruction Fetch Unit

This example is the Instruction Fetch Module for the Torch Microprocessor, taken from []. The specification implements the instruction fetch state machine and specifies a few invariants that hold in certain states.

8.3 PS/2 Interface

The PS/2 interface was introduced by IBM as an interface standard in order to connect the keyboard and mouse to an IBM PS/2 PC. It is still found in allmost all IBM compatible PCs. The PS/2 interface is a serial bus. The communication is bidirectional. The bus uses a single data signal and a clock signal. The PS/2 Verilog implementation provides an interface to this bus. Besides the basic protocol, the modules also decode the packets that are received. For example, the keyboard controller keeps track of the state of the shift keys and computes an ASCII equivalent of the key pressed. If a key is pressed, generates an event and waits for an acknowledgement. The mouse interface keeps track of the position of the mouse and the mouse buttons.

The keyboard controller has 67 latches and is about 700 lines of Verilog. The ANSI-C code we wrote for it does not try to reproduce the behavior of the Verilog in a cycle-accurate way. Instead, the ANSI-C code communicates with the Verilog using the module interface. The ANSI-C non-deterministically picks a key and code generates an appropriate PS/2 clock and data signal, which is fed to the Verilog module. Thus, it plays the role of the keyboard, while the Verilog module is the computer. After sending the packet, it waits for the circuit to decode the packet using `WAITFOR`. The decoded key is then compared to the key that was sent. Due to the length of the packets and the fact that the Verilog module is oversampling the data on the PS/2 bus, a successfull run requires a bound of at least 48. The overall runtime, including unwinding and generation of the CNF, is 51 seconds.

8.4 DLX

The DLX Implementations

The DLX architecture [22, 23] is a load/store architecture with a RISC instruction set that is similar to the MIPS instruction set. The general purpose (GPR) register file of the DLX architecture consists of 32 integer registers (R0,...,R31), each of which is 32 bits wide. The register R0 is defined to be always zero. The general purpose registers are used for all integer operations and memory addressing purposes.

We compare a hardware and a software implementation of the DLX. The hardware implementation is a sequential implementation. The control has five states fetch, decide, execute, memory, and writeback, following the implementation suggested in [22]. The implementation is given in synthesizeable Verilog and consists of the register file, the ALU, and the main control. Thus, it does not contain a model of the main memory but rather a memory bus interface. Including the register file, the implementation contains a total of 1219 latches.

The ANSI-C software implementation is derived from a DLX simulator `dlxsim`. It implements the ISA only and is not a cycle-accurate simulation. In particular, it does not make use of a state machine but rather uses ANSI-C flow control to distinguish between individual instructions. In order to verify equivalence between the hardware and software implementation the data read from memory by both machines must be the same. This is achieved by changing the software implementation. Instead of reading the memory contents from an array, the software implementation watches the hardware implementation and copies the data the hardware implementation reads from the memory bus. In order to do so, the software implementation needs to know the cycle the data word is on the bus, i.e., the read accesses must be synchronized. The synchronization between the two machines is done using the WAITFOR construct. The software implementation waits until the software implementation is in the appropriate state and the memory bus becomes active. This is detected by watching the `MEM_BUSY` signal. This signal is active if the memory data is unavailable for any reason.

There are two different types of memory accesses: the instruction fetch and the load/store instructions. The hardware implementation stores its state in a register called `state`. In order to get the instruction word, the software implementation waits until the hardware implementation is in the instruction fetch state (state 0) and the memory bus is not busy:

```
cycle=WAITFOR(cycle,!MEM_BUSY[cycle] && state[cycle]==0);
```

Once the hardware implementation is in the instruction fetch state, and the memory data is available, the data word on the bus (as given by the `MEM_IN signal`) is read into the `ir` variable:

```
if(cycle<=bound) ir=MEM_IN[cycle];
```

In case of a load instruction, the software implementation waits until the hardware implementation is in the instruction memory state (state 3), and the memory data is available. The data word is then read from `MEM_IN`:

```
cycle=WAITFOR(cycle,state[cycle]==3 && !MEM_BUSY[cycle]);
if(cycle<=bound) result=MEM_IN[cycle];
```

The equivalence of the two implementations is specified using an assertion that establishes the equality of the value written into the respective register files. In the software implementation, this value is denoted by `result`. In the hardware implementation, the result is stored in the register `C`.

```
assert(C[cycle]==result);
```

Results

Instances with Bugs As described above, the software implementation is derived from a DLX simulator `dlxsim`. This implementation contains a bug in the code that decodes the instruction word. The DLX architecture provides control instructions (conditional branch and jump), ALU instructions such as add and compare, and the memory instructions load and store. The instruction that is to be executed is encoded in a 32-bit instruction word.

	6	5	5	16
I–type	Opcode	RS1	RD	Immediate

	6	5	5	5	5	6
R–type	Opcode	RS1	RS2	RD	SA	Function

	6	26
J–type	Opcode	PC Offset

Figure 9: Integer instruction formats of the DLX

There are three instruction formats for integer instructions (figure 9): the I-type format provides a 16-bit immediate constant and two register addresses, the R-type format provides three register addresses, a 5-bit immediate constant and an additional 6-bit function code. The J-type format provides a 26-bit immediate constant, which is used as PC offset for jump instructions.

Using the original instruction word decoding code from `dlxsim` and an unwinding bound of 5 cycles, `CBMC`generates a counterexample within 1 minute and 37 seconds.

The time includes the time for unwinding the ANSI-C code, synthesizing and unwinding the Verilog code, generating the CNF, and running Chaff on the generated CNF. The counterexample provides the instruction word of the instruction that is executed: the instruction is a `jal` (jump and link) instruction, which is a J-type instruction. Furthermore, the counterexample shows how both implementation process this instruction. The machines disagree on the new value of the PC (program counter). The hardware implementation computes the correct new value, as defined by the specification. However, the software implementation only extracts 25 bits of the PC offset rather than 26. This is indicated by the fact that the top bit of the offset of the immediate constant in the instruction word given in the counterexample is set.

In contrast to the bug described above, the bug described in the following is inserted artificially to test the tool. It is not part of the original code. In order to obtain a counterexample that requires more than one instruction, we change the write back enable signal in the hardware implementation such that it is no longer active in case of an `ALUi` instruction. Thus, the result of the `ALUi` instruction is no longer written into the register file. The instructions following the `ALUi` instruction therefore potentially read the wrong value from the register file. As expected, with an unwinding bound of 10 cycles `CBMC`generates a counterexample within 34 minutes and 51 seconds. The first instruction in the trace is an `ALUi` instruction. The second instruction is a `jr` instruction that reads the result of the first instruction. Since this is wrong, the `jr` instruction computes a wrong value for the new PC.

Runtime without Bugs The unsatisfiable, i.e., correct instance contains 196697 variables and 731851 clauses with an unwinding bound of 10 cycles. Chaff detects it to be unsatisfiable within 154 minutes. The instance consists of a total of 80 claims, which includes the automatically generated array bounds checks.

9 Conclusion and Future Work

We have described the translation of ANSI-C programs and Verilog designs into a SAT instance using Bounded Model Checking. We have performed multiple experiments, including a small processor given in Verilog and its ISA given in ANSI–C.

We are currently developing an extension of this technique to handle multiple clock domains. This continuation work will enable the specification of relationships between clock frequencies, and the verification of a multiple clock design under these assumptions.

We plan to add support for concurrent C programs, such as allowed by the SpecC language [13]. Furthermore, we plan to optimize the generation of the SAT instance using specialized bit vector decision procedures and abstraction techniques.

References

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

[2] 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.

[3] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a Power PC^{TM} microprocessor using symbolic model checking without BDDs. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV 99)*, Lecture Notes in Computer Science. Springer Verlag, 1999.

[4] 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.

[5] Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[6] 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.

[7] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation*. Springer, 2003. To appear.

[8] O. Shtrichman. Tuning SAT checkers for bounded model checking. In E.A. Emerson and A.P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, Lecture Notes in Computer Science. Springer Verlag, 2000.

[9] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, number 2102 in Lecture Notes in Computer Science, pages 436–453. Springer Verlag, 2001.

[10] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, number 2102 in Lecture Notes in Computer Science, pages 454–464. Springer Verlag, 2001.

[11] 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*. ACM Press, 2002.

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

[13] http://www.specc.org.

[14] I. Page. Constructing Hardware-Software Systems from a Single Description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[15] 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.

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

[17] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.

[18] Acellera, http://www.accellera.org.

[19] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series F*. Springer-Verlag, 1984.

[20] R. Cytron, J. Ferrante, B. K. Rosen, , M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM Press, 1989.

[21] Donald E. Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Kluwer, Boston;Dordrecht;London, 1991.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., San Mateo, CA, 1990.

[23] Philip M. Sailer and David R. Kaeli. *The DLX Instruction Set Architecture Handbook*. Morgan Kaufmann, San Francisco, 1996.