Specifying and Verifying Systems with Multiple Clocks *

Edmund M. Clarke, Daniel Kroening, and Karen Yorav
Computer Science Department
Carnegie Mellon University
e-mail: emc,kroening,kareny@cs.cmu.edu

Abstract

Multiple clock domains are a challenge for hardware specification and verification. We present a method for specifying the relations between multiple clocks, and for modeling the possible behaviors. We can then verify a hardware design assuming that the clocks meet these constraints. We implement our ideas in the context of SAT based Bounded Model Checking (BMC), using ANSI–C programs to specify the functional behavior of the design.

1 Introduction

Formal methods have become indispensable in designing hardware systems. The ability of formal methods, and in particular model checking, to check large systems has increased dramatically in the past few years. The presence of multiple clock domains in a hardware design adds additional complexity that makes formal methods even more desirable. At the same time, the specification of multiple clock domains is a challenge. Typically, high level specification languages such as SystemC [10] require the users to generate the clocks themselves. Commercial model checkers available today will either require the user to generate the clocks, or not support multiple clock domains at all.

The behavior of designs with multiple clocks often depends on specific properties of these clock signals. As a trivial example, consider a simple parallelizer, i.e., a circuit that takes as input a serial signal from one clock domain, *A*, and outputs it in eight bit packets in the other clock domain, *B*. This design will fail if the clock of domain *A* is faster than eight times the clock of domain *B*. Thus, without constraints that represent relationships between clocks, any verification effort is bound to fail.

We present a way to specify complex relationships between two or more clocks, including constraints on the frequency of the clocks. The clock specifications can be loose, allowing many different clocking schemes rather than a specific relationship between clocks. We then show how to model these specifications, so that the system can be verified under the specified constraints, using any type of formal verification technique. We do this by adding a special purpose state machine that determines the behaviors of the clocks. This machine is non-deterministic, which allows us to model different clocking schemes at the same time. A counterexample, if found, will contain the full timing information for a particular clocking scheme causing this error.

We have implemented our ideas in the context of *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 using a SAT procedure such as Chaff [8]. If the formula is satisfiable, a counterexample is extracted from the output of the SAT procedure. If the formula is not satisfiable, the state machine and its specification are 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 [12, 6, 3].

Our tool uses BMC to verify Verilog designs against specifications written in ANSI-C [5]. When a new device is designed, a ”golden model” is often written in a programming language such as ANSI–C. This model is extensively simulated to insure both correct functionality and performance and later on implemented in a hardware description language such as Verilog. It is essential to determine that the C and Verilog programs are consistent. We extend the tool of [5] with the results presented here. We translate clock

*This research was supported by the National Science Foundation (NSF) under grants no. CCR-0121547 and CCR-0098072, by the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, by the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARO, ONR, NRL, NSF, SRC, the U.S. Government or any other entity.

constraints into Boolean constraints that are added to the Boolean formula representing the (bounded) computation of the design. Thus, when a satisfying instance is found (a counterexample), it contains a computation of the design in which the clocks behave according to the specification.

The results we present here can be integrated with many verification paradigms, in addition to Bounded Model Checking. The same method that we use here in order to analyze and translate clock constraints can be used in other formal verification methods such as Explicit State and Symbolic Model Checking.

Related Work

In [11], a tool for verifying the combinational equivalence of RTL-C and an HDL is described. The RTL-C code is translated into HDL, and then standard equivalence checkers are used to establish equivalency. However, the whole design is assumed to have a single clock.

There are variants of C designed specifically for the purpose of hardware specification. The System C standard defines a subset of C++ that can be used for synthesis [10]. Other variants of ANSI–C are Spec C and Handel C. We chose to work with the the integer subset of the ANSI–C language, which is more powerful than the variants meant for synthesis. This enables us to handle programs that were written as high-level system specifications, rather than programs that were targeted at hardware specifically. Although we do not yet support the use floating point operations, our tool supports most other ANSI-C operations, including pointer arithmetics, bit-wise operators, and type conversion.

We are not aware of any specification language used for formal verification that enables specifying relationships between clock frequencies. For example, the new property specification language developed by Accellera [1] allows specifying a *Clock Expression*. This means that the clock expression determines what the formula will consider as a basic step. However, the language does not supply any mechanism for specifying the clocks themselves, for example specifying relationships between clock frequencies.

The work of [14] is related in that it shows how to model phenomena that occur in multiple clock designs. In that work the effect of a synchronizer flip-flop going into a meta-stable state is modeled using non-determinism. This enables the detection of a certain type of design errors using model checking. In this work we are not concerned with such low level phenomena, we view the design at a higher level of abstraction and model its behavior in a world where setup and hold times are zero, and flip-flops can never get into meta-stable states. This is the level in which model-checking is usually used, because at this level everything can be described in a discrete manner.

In order to implement a clocking scheme we assign a variable for each clock. We imagine a global *verification tick*, which is a discretization of time. At every verification tick each clock in the system may either tick, or not. This is marked by the variable associated with each clock having the value 1 or 0 respectively. The RuleBase model checker [9] uses a similar mechanism for implementing clocking schemes. However, the clocking scheme needs to be explicitly implemented by the user. The VIS [13] model checker also supports multiple clock domains and requires manual creation of clocking schemes. In our case the clocking schemes are generated automatically, according to the user specification.

Outline Section 2 explains what kind of constraints we allow for multiple clock signals. Section 3 describes how we translate linear equality constraints into finite state machines, while Section 4 does the same for inequalities. Section 5 shows how to use these finite state machines for Bounded Model Checking, and Section 6 gives details on the examples we have been able to verify. Section 7 gives conclusions and plans for future research.

2 Multiple Clock Domains

2.1 Clock Constraints

The main challenge when specifying designs with multiple clocks is that the desired functional behavior in the general case requires assumptions on the clock signals. If these assumptions are not met by the clocks the design may not fulfill its functional specification. In order to solve this problem we allow the user to pose constraints specifically on the clock signals. We distinguish between *frequency constraints* and *source constraints*.

Frequency Constraints These constraints specify or restrict the frequency of clocks. We allow any number of linear equalities and inequalities, connected with Boolean conjunction or disjunction. The syntax is formally defined as follows. A *frequency expression* (*freq_exp*) consists of the following:

$$\begin{aligned}
&\mathit{freq_exp} := \\
&\quad x\ \texttt{Hz} \mid x\ \texttt{KHz} \mid x\ \texttt{MHz} \mid x\ \texttt{GHz} \mid \texttt{freq}(\mathit{clk}) \mid \\
&\quad x * \mathit{freq_exp} \mid \\
&\quad \mathit{freq_exp} + \mathit{freq_exp} \mid \mathit{freq_exp}\ \text{-}\ \mathit{freq_exp}
\end{aligned}$$

where x is a rational number and *clk* is the name of a clock signal. The syntax for *equality constraints* (*eq_const*) is:

$$\mathit{eq\_const} := \mathit{freq\_exp1} = \mathit{freq\_exp2}$$

The syntax for an *inequality constraint* (*ineq_const*) is:

$$\mathit{ineq\_const} := x * \texttt{freq}(\mathit{clk1}) \bigtriangleup y * \texttt{freq}(\mathit{clk2})$$

where x, y are rational numbers, *clk1, clk2* are two (different) clock signal names, and $\triangle \in \{>=, <=\}$. Finally, the syntax for a *frequency constraint* (*freq_const*) is a combination of the above:

freq_const :=
  eq_const |
  ineq_const |
  freq_const1 && *freq_const2* |
  freq_const1 || *freq_const2*

For example, the frequency equality constraint

```
freq(clk1)=2*freq(clk2)
```

specifies that the frequency of `clk1` is exactly twice the frequency of `clk2`, while the frequency constraint

```
freq(clk)=100 MHz ||
freq(clk)= freq(clk1) + 40 MHz
```

specifies that the frequency of `clk` is either 100MHz or 40MHz more than the frequency of `clk1`. Equality constraints are allowed to contain any number of clock frequencies, while inequalities may only involve the frequency of two clocks. For example

```
freq(clk1)>=2*freq(clk2)
```

specifies that the frequency of `clk1` is at least twice as high as the frequency of `clk2`.

We also allow the user to specify the *offset* for a given clock. This number is the first time the clock ticks. For a clock with frequency `freq(clk)`, the offset is a number in the range $[0, 1/\texttt{freq(clk)})$, i.e., it is greater or equal to 0 and less than the period of the clock. It is useful to specify offsets in order to determine a specific delay between clocks when this is known. The syntax for offset constraints is the same as for frequency constraints. For example, to specify that `clk1` and `clk2` have the same frequency, and an offset of 2ns between them, we can write:

```
(freq(clk1) = freq(clk2)) &&
(offset(clk1) = 0ns) &&
(offset(clk2) = 2ns)
```

As is the case with frequency specifications, we allow the user to specify relationships between offsets, such as:

```
offset(clk1) >= offset(clk2)
```

We automatically verify that the constraints specified by the user do not contradict each other, or produce values that are illegal (such as negative frequencies or offsets that are larger than the period).

Source Constraints These constraints specify a partitioning of the clock domains of the system into disjoint sets of clocks. All clocks within one of these sets are assumed to be synchronized, i.e., distributed without any drift. This is used for clocks that are generated from a single source. Formally, any events triggered at the same time by synchronized clocks are assumed to be simultaneous. In contrast, if two clocks are un-synchronized then even when they are supposed to trigger events at the same time there could be a slight difference between them. The syntax for specifying that the two clocks `clk1` and `clk2` are synchronized is:

```
SYNC clk1, clk2
```

We only allow conjunctions of clock source constraints, no other Boolean connectives. This is meant to make sure that we get a strict partitioning of the clocks. Unless specified otherwise, all clocks are assumed to be un-synchronized. We use $c_1 \overset{S}{=} c_2$ to denote that c_1 and c_2 are synchronized, and $c_1 \overset{S}{\neq} c_2$ to denote that they are un-synchronized.

Note that two clocks c_1 and c_2 can be specified as synchronized even when they have different frequencies. If, for example, the frequency of c_1 is twice as high as the frequency of c_2, and if there is no offset between the clocks, every second tick of c_1 happens simultaneously with a tick of c_2. Because they are specified to be synchronized, we model the circuit so that all flip-flops in both domains change value at the same instant. If the clocks were un-synchronized, we would need to consider the possibility of a slight difference between the clock timings.

2.2 Modeling Clock Constraints

We use a discretization of time called a *verification tick*. This concept is similar to a simulation tick in an event-based simulation of a design. At every verification tick each of the clocks in the system may either tick, or not. For each clock domain c_i, this is indicated by a Boolean *Clock Enable* variable ce_i. In the special case of a design with a single clock the verification tick and the clock tick are one and the same.

We translate the clock constraints described above into a Boolean formula together with the Verilog design. This translation process has two steps:

Step 1 We translate the various clock constraints from the specification into a state machine, where a single step of the state machine represents exactly one verification tick. The transitions (edges) of the state machine are labeled with sets of clocks, each describing a valid assignment to the variables ce_i. A clock is enabled if it is in the set, and disabled otherwise. This is described in Sections 3 and 4.

Step 2 We unwind the state machine generated for the clock constraints together with the Verilog design up to the given bound, and create a Boolean formula that represents this computation of the state machine. The details of this process are described in Section 5.

We start with a preparation phase, in which we convert the expression containing the clock constraints into a sum of products (DNF). Since we expect the number of constraints to be small, we do not anticipate a problem with the potential exponential blowup.

Let $D_1,\dots,D_d$ denote the d terms of the DNF. These are translated separately into finite state machines $A_1,\dots,A_d$. The final state machine A then non-deterministically picks one of the state machines A_i, by having multiple initial states. In the following we assume a single term D that is to be translated into a finite state machine A.

Next, we sort the constraints in D into equality and inequality constraints. The set of equality constraints will result in two state machines, A_{fixed} and $A_{variable}$, describing clocks with fixed and variable frequencies respectively. The set of inequality constraints results in a third machine, A_{ineq}. We then create the parallel composition of the three machines, to get the desired machine A.

3 Scalar Equality Constraints

Assume a given term of equality constraints containing m constraints on n different clocks. The first step is to optimize by solving the linear equation system given by the collection of frequency constraints. If this equation system is unsatisfiable we inform the user of this fact. If there is a solution to the equation system, there may be clocks that have a unique, fixed frequency, i.e., an exact number, and also clocks that have a variable frequency, i.e., are specified only in relation to other clocks. Similarly, the solution to the equation system may result in either exact or relative values for clock offsets, if it constrains the offsets at all. For example, in the following specification `clk1` and `clk2` have variable frequencies, while the frequency of `clk3` is fixed:

```
5*freq(clk1) = 3*freq(clk2) &
freq(clk3) = 150MHz
```

We deal with these two types of clocks separately.

3.1 Clocks with Fixed Frequency

Assume we have n clocks $c_1,\dots,c_n$ with fixed frequencies $freq(c_i)$. For the time being we also assume that we have fixed offsets $offset(c_i)$; we later explain how to handle the general case. The state machine generated for these clock signals follows event queue semantics. In our case, an event is a clock tick of one of the clocks. The set of events is $E=\{e_1,\dots,e_n\}$, where the event e_i represents the clock c_i ticking. A configuration Q of an event queue is a mapping from the set of events E into the amount of time until they occur next:

$$Q : E \longrightarrow \mathbb{Q}$$

(where $\mathbb{Q}$ is the set of rational numbers).

For each event e_i, $Q(e_i)$ is the time that will pass until this event happens next. For a given configuration Q the time of the next (closest) event is denoted by $\nu(Q)$ and is the minimum time of all events:

$$\nu(Q) := \min\{Q(e_i)\,|\,i=1,\dots,n\}$$

Since several events can happen synchronously, there may be more than one event with $Q(e_i)=\nu(Q)$. The set of *active* events E^a is the collection of events that are going to happen next and is given by:

$$E^a(Q) := \{e_i\,|\,Q(e_i)=\nu(Q)\}$$

For each possible configuration of the queue we generate a state in the state machine A_{fixed}. The initial state Q_0 is the initial configuration of the queue, which is determined by the first occurrences of all events:

$$\forall e_i \in E : Q_0(e_i) := \mathit{offset}(c_i)$$

Since each clock has a specified frequency $freq(c_i)$, its clock period is given by $p_i = 1/freq(c_i)$. Each configuration Q_j, for $j \geq 0$, has a single successor configuration Q_{j+1} that is computed by:

$$Q_{j+1}(e_i) := \begin{cases} Q_j(e_i)+p_i & : \; e_i \in E^a(Q_j) \\ Q_j(e_i) & : \; \text{otherwise} \end{cases}$$

Two configurations Q_j, Q_k are said to be equivalent if they specify the same time differences between clocks:

$$\begin{aligned} Q_j \equiv Q_k \quad :\Longleftrightarrow \quad & \forall e_i \in E : \\ & Q_j(e_i)-\nu(Q_j) = Q_k(e_i)-\nu(Q_k) \end{aligned}$$

Since we have a finite number of clocks, each with a fixed period, the sequence $Q_0, Q_1, \dots$ is periodic, i.e., there exists a positive number α such that $Q_\alpha \equiv Q_0$. Let α be the smallest such number.

The finite state machine A_{fixed} is computed as follows: We compute the sequence Q_j up to $j=\alpha-1$ or until the BMC bound is reached. For each unique configuration Q_j a state is generated. The initial state of the machine is the state labeled with Q_0. We add transitions from each state Q_j to state Q_{j+1}. In case the period of the sequence was found before the bound was reached, a back loop from state $Q_{\alpha-1}$ to state Q_0 is added. Furthermore, we add an idle transition (self-loop) to each state.

We add the following constraints to the transitions: for each state Q_j, the transition from this state into Q_{j+1} is labeled by the constraint $E^a(Q_j)$, i.e., all the clocks given in the set $E^a(Q_j)$ must be active during the step from Q_j to its successor, and all the clocks not in the set must be inactive. Self-loops are constrained using the empty set, i.e., all clocks in E must be inactive. These self loops are required only when there are several machines running in parallel, for example, when we have both fixed frequency and variable frequency clocks. Each group generates a machine so that when they are put together they enable any interleaving of the two sets of clocks that they describe. In the final state machine we add a constraint that at each verification tick at least one clock is active, to eliminate computations in which nothing happens.

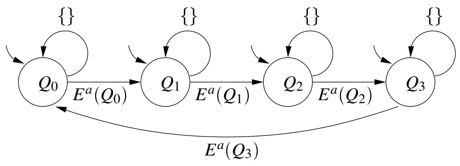
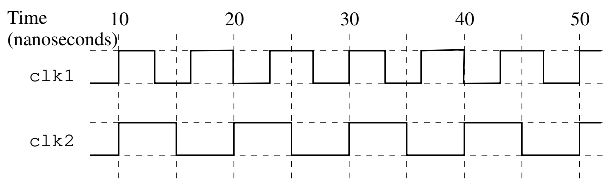


Figure 1. Translation for synchronized clocks with fixed frequencies

As an example, assume we have a design with two clocks and the specification:

```
clk1 = 150 MHz & clk2 = 100 MHz &
SYNC clk1, clk2
```

There are two events in the system: e_1 (for `clk1` with a period of $p_1 = 6.6ns$ and e_2 (for `clk2`) with a period of $p_2 = 10ns$. Notice that the period of e_1 is actually $6\frac{2}{3}$. We use accurate arithmetic computations to make sure that we recognize equivalent configurations. For the purpose of this example, we set the first occurrence (the offset) for both of them to $10ns$. The timing diagram for these clocks looks like this:



The configurations Q_j according to the above definition, along with the events that occur in each configuration are:

$$
\begin{array}{ll}
Q_0 = (10, 10) & E^a(Q_0) = \{e_1, e_2\} \\
Q_1 = (6.6, 10) & E^a(Q_1) = \{e_1\} \\
Q_2 = (6.6, 3.3) & E^a(Q_2) = \{e_2\} \\
Q_3 = (3.3, 10) & E^a(Q_3) = \{e_1\} \\
Q_4 = (6.6, 6.6) \equiv Q_0 &
\end{array}
$$

The first configuration, Q_0, is created by the two offsets. The next configuration, Q_1, is created from Q_0 by taking both periods, since in Q_0 both clocks were active. In Q_1 only clk_1 is active, so we have $Q_2(e_1) = p_1$, and $Q_2(e_2) = Q_1(e_2) - \nu(Q_1) = 10 - 6.6 = 3.3$. This continues until we have $Q_4 = (6.6, 6.6)$, which is equivalent to Q_0. Each configuration generates a state in the state machine created for this specification (Figure 1). When translated into verification ticks we get the following clocking scheme:

Verification Tick	1	2	3	4	5	6	7	...
ce_1	1	1	0	1	1	1	0	
ce_2	1	0	1	0	1	0	1	

Up until now we assumed that all offsets were given specifically, which gives us a specific initial configuration Q_0. In practice, the offsets may be totally or partially specified, using equalities and inequalities. To make sure that we take into account all possible clock schemes we enumerate all possible permutations between the clocks. For two clocks, c_1 and c_2, for which there are no constraints on the offsets, we need to generate three different clock scenarios: one in which c_1 ticks first, one in which c_2 ticks first, and one in which they tick together. Furthermore, if $p_1 < p_2$ we need to consider a situation in which c_1 ticks twice (or more) before the first time that c_2 ticks. We systematically generate all possible scenarios, and choose representative offset values for each scenario. Each of these generates a different state machine (starting from a different initial configuration). We put all of these together to get a machine A_{fixed} that has several initial states.

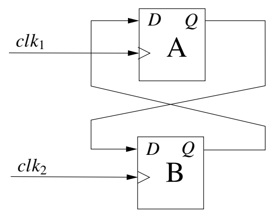
When we have constraints on the clock offsets, we again solve the linear equation system that they create. The solution to this system can be translated into constraints on the possible permutations between the clock offsets. This enables us to generate a smaller number of scenarios and results in a smaller machine A.

3.2 Un-Synchronized Clocks

The state machine generated above represents accurate semantics for clocks assuming they are all synchronized, i.e., that all events occurring at the same time are actually processed simultaneously. However, this is not a good model of the physical world, in which events triggered by non-synchronized clocks are rarely really simultaneous.

We model these effects by non-deterministically picking between a synchronous transition and the possible asynchronous transitions. If there are two events e_1 and e_2 that are not synchronized but occur at the same time according to the event queue, the following event orderings are allowed: (1) The event e_1 first, then the event e_2, (2) the event e_2 first, then the event e_1, (3) the events e_1 and the event e_2 simultaneously.

To illustrate why this is necessary, consider that these two events correspond to two clock signals that clock two latches `A` and `B`, as shown below. If e_1 occurs first and e_2 second, the original value of B will be present in both latches. If both events occur simultaneously, the values of A and B are swapped.



To model this phenomenon, we split states into sub-states. Every state Q_j that has two or more un-synchronized

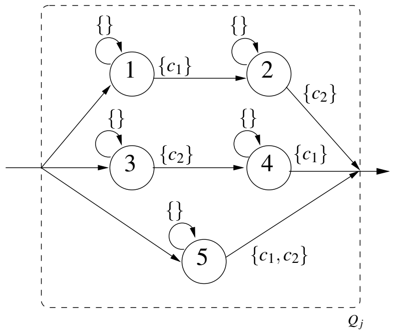


Figure 2. State machine for two un-synchronized events e_1 and e_2 that occur at the same time.

events in $E^a(Q_j)$ is split into sub-states. Each sequence of sub-states represents one possible ordering of the synchronized events, allowing multiple events to occur simultaneously. In the case of two un-synchronized events e_1 and e_2, the state Q_j is replaced by five sub-states as depicted in Figure 2. Each path represents one of the three orderings of the events as described above.

This construction is easily generalized to any number of un-synchronized events. Again we are not concerned with the blow-up in the number of states of the state machine since we do not anticipate more than a few clock domains in a single design. For a small number of clocks the situation where two events happen at the same time will be relatively rare. Each such instance adds 5 states. A situation where more than two events happen at the same time will produce more states, but will be too rare to effect the efficiency of the whole run. Note also that this "blow up" effects only the machine that describes the clocks, and is independent of the size of the design.

3.3 Clocks with Variable Frequency

This subsection describes how constraints of the form

```
a * freq(clk1) = b * freq(clk2)
```

are converted into a finite state machine. We separate the set of clocks involved in these constraints into disjoint sets so that the clocks in each set depend on each other, directly or indirectly, and do not depend on clocks outside the set. We generate a separate state machine for each of these sets. The overall state machine $A_{variable}$ then is the parallel composition of the state machines. Since the sets of clocks are all disjoint, the constraints of the parallel state machines do not affect each other.

Let $c_1, \ldots, c_n$ be the clocks in one such set. We solve the linear equation defined by the set of inequalities involving $c_1, \ldots, c_n$. We require that the dependency between clocks is linear, and that in the solution space of the equation system all the axes are orthogonal. This is a restriction, but we believe that constraints violating it are of low practical value. It is then possible to choose a single clock c, which we call the *reference clock*, and define a linear factor $\theta(c')$ for every other clock c' in the set such that:

$$freq(c') = \theta(c') * freq(c)$$

Obviously, the actual frequencies of the clocks are not relevant, only the relationships between them. We therefore fix the period of the reference clock to 1, and the period of any other clock c' to $1/\theta(c')$. The state machine for this set of clocks is generated as described in subsection 3.1.

It should be noted that whenever we create a parallel composition of more than one state machine we generate a slight over-abstraction of the behavior of the design. This is because there is no bound on the number of times each machine can take the idle transition. In theory, we may have a counter example in which one machine takes many steps, while the other is idle, and then the second machine takes several steps while the first machine is idle. This computation cannot happen in reality, since it corresponds to unstable clocks that change frequencies during the computation. However, since we are using bounded model checking, the counter example is the shortest possible, so it will include the minimum number of idle transitions. It is our experience that in such examples it is usually (though not always) the case that the problem is real, even if the behavior of the clocks is not exactly realistic. One of our future research goals is to create an algorithm that will automatically detect this situation, and check whether there is a similar counter example in which the clock frequencies behave realistically.

4 Inequality

We next describe how constraints of the form

$$a \cdot freq(c_1) \quad \leq \quad b \cdot freq(c_2) \tag{1}$$

are converted into a finite state machine A_{ineq}. For efficiency reasons, this conversion is not exact but an over-approximation, i.e., all behaviors allowed by the exact constraints are also allowed by our approximation, but our approximation allows additional behaviors.

As a first step, the constraint is transformed using an auxiliary clock variable, here denoted by c'_1, with a frequency that is a/b times the frequency of c_1. This clock is also synchronized to the clock c_1.

$$freq(c'_1) = \frac{a}{b} freq(c_1) \quad \wedge \quad c'_1 =_s c_1 \tag{2}$$

The offset of c'_1 is the offset of c_1 adjusted accordingly to the factors a and b:

$$offset(c'_1) \quad = \quad \frac{a}{b} offset(c_1) \tag{3}$$

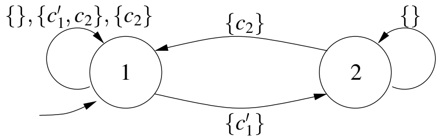


Figure 3. State machine for two clocks c'_1 and c_2: c'_1 is enforced to be slower than c_2.

The frequency equality constraint (2) is converted to a state machine as described in subsection 3.3. We can now rewrite (1) as follows:

$$freq(c'_1) \quad \leq \quad freq(c_2) \qquad (4)$$

We over-approximate this constraint using a state machine with two states, s_1 and s_2, as shown in Figure 3. This machine is intended to make sure that the faster clock ticks at least as many times as the slower one. The first state, s_1, is the initial state, and is used to indicate that the faster clock c_2 has made a tick after or at the same time as the slower clock c'_1. The second state, s_2, indicates that the slower clock has made a tick after the last tick of the faster clock. This is realized by adding three transitions: (1) A self loop on s_1 requires the faster clock c_2 to tick. It does not constrain the slower clock c'_1, it may or may not tick. (2) A transition from s_1 to s_2 requires the faster c_2 clock not to tick, and the slower clock c'_1 to tick. (3) A transition from s_2 back to s_1 requires that the faster clock c_2 ticks, while the slower clock c'_1 is not allowed to tick. In addition to the transitions above, we add an "idle" transition to both states, which allows none of the clocks to tick.

5 Implementation in the context of BMC

The input to the tool is an ANSI–C program with embedded assertions, and a Verilog design. The C program can refer to the values of signals in the Verilog design at any clock cycle. Verification is performed by transforming the C program, the Verilog design, and the assertions into a bit-vector equation such that a solution to the equation represents a counterexample. This equation is translated into a CNF formula and handed to an efficient SAT solver. If some assertion fails, the satisfying instance given by the SAT solver is presented to the user in the form of a counterexample execution, which includes both the C program execution and the Verilog execution sequence.

To use our method in the context of SAT based Bounded Model Checking, we need to create a Boolean formula that represents a bounded number of steps of the design, following the clocking scheme given by the state machine A. Let k be the model checking bound. The state machine A is unwound in the classical way for k steps. We assign a variable V_A^l for the state of A at verification tick l. We create clauses that represent the (possibly non-deterministic) transition relation of A so that any satisfying instance to the resulting Boolean formula assigns the variables $V_A^1, \ldots, V_A^k$ with a valid path in A.

Next, we need to make sure that each clock ticks when it should. For this we use the precomputed sets E^a for each state of A. We generate clauses that specify that

$$V_{clk_i}^j = 1 \Leftrightarrow e_i \in E^a(V_A^j)$$

We continue with the same example of Figure 1. Let the model checking bound be $k = 3$. The formula we create will use the variables $V_A^0, \ldots, V_A^3$ to keep the current state in A. Recall that different values for the first occurrence of each clock result in different state machines. Let us assume that the state machine depicted in Figure 1 is A_1, and the global state machine A chooses non-deterministically between A_1 and another machine, called A_2. Let R_0 be the initial state of A_2. The formula we generate will be the conjunction of several sub-formulas. The first defines the non-deterministic choice between A_1 and A_2:

$$f_1 := (V_A^0 = Q_0) \lor (V_A^0 = R_0)$$

Next, we generate a formula that represents the transition relation of A_1:

$$f_2 := \bigwedge_{j=0,\ldots,2} (V_A^j = Q_0 \rightarrow V_A^{j+1} = Q_1) \land$$
$$(V_A^j = Q_1 \rightarrow V_A^{j+1} = Q_2) \ldots$$

We do the same for the transition relation of A_2, resulting in a formula f_3. Next, we create a formula that specifies when each clock should tick:

$$f_4 := \bigwedge_{j=0,\ldots,3} (V_A^j = Q_0 \rightarrow V_{clk_1}^j \land V_{clk_2}^j) \land$$
$$(V_A^j = Q_1 \rightarrow V_{clk_1}^j \land \overline{V_{clk_2}^j}) \ldots$$

Again we have a similar formula f_5 for A_2.

Finally, we conjoin the formulas $f_1, \ldots, f_4$ and transform them into CNF. The resulting formula is passed to the SAT checker, along with the CNF formulas of the Verilog design, the C specification, and the assertions that are to be proven.

6 Experiments

A Serializer The serializer takes two clock signals `cp` and `cs` as input. In addition to that, an eight bit data word is read from the environment when `cp` ticks. The word is sent out bit by bit on every tick of `cs`. The C program waits for `cp` to tick, stores the data word in a local variable and then compares it to the output on the serial side. The frequency of the clock `cs` needs to be eight times the frequency of `cp`. This is realized by the following constraint:

```
f(cs)=8*f(cp) & SYNC(cs,cp)
```

This constraint generates about 210 clauses and 63 literals per transition.

A FIFO This design was given to us by Marvell [2]. It is a FIFO that is used to transfer data packets between two clock domains in a communications switch. The design is implemented using two internal FIFOs, and it switches between them on every packet. The unique feature of this design is that it uses the same FIFO to transfer both data and control. This is an obstacle for symbolic model checking algorithms, and in fact this design could not be verified using SMV based model checkers.

The fifo is meant to operate under specific conditions. The two clock domains are not synchronized, and need not have exactly the same frequency, but the frequencies must be very close. Because the fifo has no overrun/underrun detection, it will fail to operate properly if the difference between the clocks is too large. The environment that drives packets into the FIFO will always insert a certain delay between two consecutive packets, so that the FIFO has a chance to "catch up" even if output clock is slower than the input clock. The question is – what is the minimum number of clock cycles between packets that guarantees no overflow? Without the ability to specify the relationship between the clocks, the design can only be verified using a specific ratio that would be coded by hand. However, the FIFO needs to be verified under several different possibilities, which could prove difficult to check manually. Our method allows us to specify a more general environment. The clock specification we used is as follows:

```
freq(clk_in) >= 100 MHz
freq(clk_out) >= 0.9*freq(clk_in)
freq(clk_out) <= 1.1*freq(clk_in)
```

These constraints result in the clocks either alternating or ticking together, with the possibility of one clock ticking twice every now and then, but not more than once every 10 clock cycles. We verified the design with a bound of 50. With this bound, the clock constraints generated about 33000 clauses, out of the overall 2 million clauses that were given to Chaff. This shows that the overhead for using clock constraints is not significant.

Another interesting feature of this design is that it expects packets to be of at least length 5. Our tool took seconds to show a scenario where a packet of length less than 5 is dropped.

7 Conclusion and Future Work

We have shown how complex relationships between clocks in a design can be specified for use in formal verification. We translate clock constraints into state machines that represent all possible clock schemes. These state machines can be used in a variety of ways.

While we have presented an implementation for bounded model checking, our results are equally applicable to "unbounded" classical model checking. Instead of unwinding the machine that represents the clock constraints, we create a model of the machine and add it to the transition relation of the design. This can be used to add support for multiple clock systems to *Symbolic Model Checking*, as well as *Automata Based Model Checking* [4].

This work has opened to us several directions for continued research. We believe that these ideas could lead to an innovative way of verifying asynchronous circuits, which are circuits that do not use clocks. These systems are usually simulated using an event-driven semantics, which fits with our event-driven queue semantics using state machines.

References

[1] Accellera website. `http://www.accellera.org`.

[2] http://www.marvell.com.

[3] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Proceedings of CAV 2001*, volume LNCS 2102, pages 454–464. Springer Verlag, 2001.

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

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

[6] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of CAV 2001*, volume LNCS 2102, pages 436–453. Springer Verlag, 2001.

[7] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume LNCS 2575, pages 298–309. Springer Verlag, 2003.

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

[9] http://www.haifa.il.ibm.com/formal_m.html.

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

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

[12] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *Proceedings of CAV 2000*, volume LNCS 1855. Springer Verlag, 2000.

[13] http://www-cad.eecs.berkeley.edu/respep/research/vis/.

[14] K. Yorav, S. Katz, and R. Kiper. Reproducing synchronization bugs with model checking. In *Proceedings of CHARME 2001*, volume LNCS 2144. Springer, 2001.