CTL Property Language in Formal Verification of Systems A System Approach

Hamid Shojaei, *Mojtaba Shahidi
Electrical and Computer Engineering Department, Faculty of Engineering
University of Tehran, Iran
*Electrical and Computer Engineering Department,Faculty of Engineering
Ferdowsi University of Mashhad
Address: Electrical Engineering Department,Faculty of Engineering,
Ferdowsi University of Mashhad, Mashhad, Iran

Abstract: *We use symbolic model checking to verify a VHDL design. This paper mainly focuses on Computational Tree Logic (CTL) for model checking problem. We have explained these two terms "CTL" and "model checking" for providing a clear idea about these two. Most importantly we have explored the ways of uses of CTL formulae in the case of model checking. The importance of the model checking, the ways of specifying properties in CTL and some most commonly used CTL formulae in checking are also stated. Also the uses and importance of fairness constraints in CTL formula and the conversion of CTL operators have also been included in this paper. Lastly, we have given an example of the processes of model checking.*

Keywords: VHDL, Formal Verification, Symbolic Model Checking, CTL

1 Introduction

Hardware systems are generally specified as a set of interacting finite state machines (FSMs). Network protocols and CPU controllers are such systems that their base structures consist of finite state machines. Therefore, checking the correctness of these systems is a major issuie in digital system design and verification.

Current techniques for testing FSMs are simulations and testing. But the problems with the simulations are: it can not run for ever; it will be many times slower than the simulated system; can be very expensive; and there is no guarantee all the possible runs will be simulated. Testing also suffers from similar disadvantages. There are some additional disadvantages such as, not all (infinite) inputs can be presented; input patterns needs to be automatically-generated; no guarantee that bad inputs will be presented; and especially messy for concurrent systems like multi-component systems. These two can reveal the presence of bugs but can never establish the absence of bugs.This is a fundamental limitation for safety-critical systems. For these reasons, the application of model checking is increasing day by day. Model checking uses transition systems (*Kripke Structure*) to model systems and *temporal logics* to specify properties. It makes the verification problem reduced to graph algorithmic problems that can be fully automated [1, 2, 3, 6] and relatively easy to use. It is very successful in verifying hardware, communication protocols and other many *embedded systems.* It is increasingly popular in industry. Besides, there are some robust tools such as SPIN, SMV, COSPAN, VIS, SMART etc. that can be easily and effectively used as verification tools. But the problem is as the number of interacting components increase the size of the transition system increases exponentially that creates a very serious problem namely the state explosion problem.

The use of modeling checking in place of simulation or testing is now common in most of the cases of hardware, software, concurrent systems, reactive systems design verification. This checking uses *temporal logic* (mainly CTL) to specify the properties to be verified. In our paper, we have explored a brief idea on how we can use Computational Tree Logic (CTL) for model verification purpose. We have explained about the model, the CTL formula, the necessity of *fairness*

in CTL formula and lastly a full elaborative example of model checking

2 Symbolic Model Checking

Symbolic CTL model checking is a formal verification technique which has proven itself practical in the verification of hardware specifications and implementations. A design of N latches is viewed as a state machine containing 2**N states, and the temporal logic CTL is used to reason about the design. Symbolic representation and manipulation of the design allows the state machine to be traversed without explicitly building it, thus making the technique feasible for N=100 andmore.

To understand the term "*model"*, we need to be familiar with transition system and *Kripke Structure*. A transition system is a structure *TS = (S, S0, R)* where, *S* is a finite set of states; $S0 \subseteq S$ is the set of initial states and $R \subseteq S \times S$ is a transition relation which must be total i.e. for every *s* in *S* there exists *s'* in *S* such that *(s, s')* is in $R\ (\forall s \in S\ \exists s' \in S\ .\ (s, s') \in R)$. On the other hand, *M = (S, S0, R, AP, L)* is a *Kripke Structure*; where *(S, S0, R)* is a transition system. *AP* is a finite set of *atomic propositions* (each proposition corresponds to a variable in the model) and *L* is a labeling function. It labels each state with a set of atomic propositions that are true in that state. The atomic propositions and *L* together convert a transitions system into a model.

The foremost step to verify a system is to specify the properties that the system should have. For example, we may want to show that some concurrent program never deadlocks. These properties are represented by *temporal logic*. Computational Tree Logic (CTL) is one of the versions of *temporal logic*. It is currently one of the popular frameworks used in verifying properties of concurrent systems [4].In this paper; we take consideration of only this type of logic for model checking. Once we know which properties are important, the second step is to construct a *formal model* for that system. The model should capture those properties that must be considered for the establishment of correctness. Model checking includes the traversing the state transition graph (*Kripke Structure*) and of verifying that if it satisfies the formula representing the property or not, more concisely, the system is a model of the property or not.

Each CTL formula is either true or false in a given state of the *Kripke Structure*. Its truth is evaluated from the truth of its sub-formulae in a recursive fashion, until one reaches atomic propositions that are either true or false in a given state. A formula is satisfied by a system if it is true for all the initial states of the system. Mathematically, say, a *Kripke Structure K = (S, S0, R, AP, L)* (system model) and a CTL formula Ψ (specification of the property) are given. We have to determine if $K \models \Psi$ holds (*K* is a model of Ψ) or not. $K \models \Psi$ holds iff *K, s0* $\models \Psi$ for every $s0 \in S0$. If the property does not hold, the model checker will produce a counter example that is an execution path that can not satisfy that formula.

3 Standard CTL Formulae

Formulas in standard CTL are built from atomic propositions, which correspond to variables in the model being verified, standard Boolean operators (e.g., AND, OR, XOR, NOT), and temporal operators. Each temporal operator consists of two parts: a path quantifier (A or E) and a temporal modality (F , G , X , U). There are two different path quantifiers: A indicates that the modality defines a property that should be true on all possible paths and E indicates that the property needs only hold on some path. The temporal modalities describe the ordering of events in time along a path and have the following meaning:

$F\phi$: (reads ϕ holds sometime in the future) is true of a path if there exists a state in the path where formula ϕ is true.

$G\phi$: (reads ϕ holds globally) is true of a path if ϕ is true at every state in the path.

$X\phi$: (reads ϕ holds in the next state) is true of a path if ϕ is true in the state reached immediately after the current state in the path.

$\phi \cup \psi$: (reads ϕ holds until ψ holds, called strong until) is true of a path if ψ is true in some state in the path, and ϕ holds in all preceding states.

The semantics of the CTL operators are stated below:

K, s |= EX (Ψ) there exists s' such that $s \rightarrow s'$ (R(s, s')) and K, s' |=Ψ. It means that s has a successor state s' at which Ψ holds.

K, s |= EU (Ψ1, Ψ2) iff there exists a path L = s0, s1, … from s and k >= 0 such that: K, L(k) |= Ψ2 and if 0 ≤ j < k, then K, L(j) |= Ψ1.

K, s |= AU(Ψ1, Ψ2) iff for every path L = s0, s1, … from s there exists k >= 0 such that: K, L(k) |= Ψ2 and if 0 ≤ j < k, then K, L(j) |= Ψ1.

AX (Ψ): It is not the case there exists a next state at which Ψ does not hold i.e. for every next state Ψ holds.

EF (Ψ): There exists a path L from s and k >= 0 such that: K, L(k)|=Ψ.

AG (Ψ): It is not the case there exists a path L from s and k>= 0 such that: K, L(k)|= Ψ i.e. for every path L from s and every k >= 0;K, L(k)|=Ψ

AF(Ψ) : For every path L from s, there exists k>= 0 such that: K, L(k)|= Ψ.

EG(Ψ): It is not the case that for every path L from s there is a k >= 0 such that K,L(k)|=Ψ. It means that there exists a path L from s such that, for every k>= 0: K, L(k) |= Ψ.

Some basic CTL operators among those stated above are shown graphically for easy understanding in Figure 1. In this figure, if it is assumed that in the filled states, the formula f holds, then we can say that the formula EF f, AF f, EG f and AG f are satisfied in the initial states (a) in the figures 1.a, 1.b, 1.c and 1.d respectively.

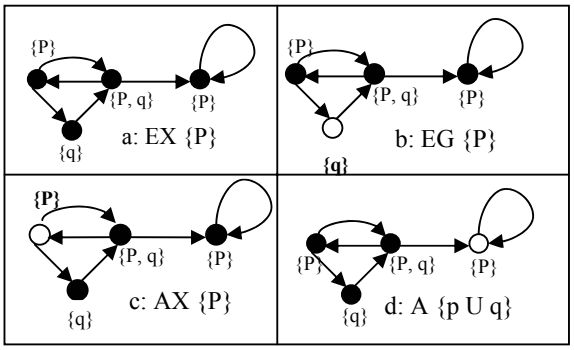

Figure 1: Basic CTL Operators

4 CTL Formula Conversion (Universal formula to Existential formula)

For a universal *CTL* formula all states in a design that are reachable from the initial states should be checked. However for an existential *CTL* formula only one case from the initial states should be found that satisfies the formula. It is clear that algorithms of existential *CTL* formula can be implemented easier than universal *CTL* formula, so universal formulas are converted to existential formulas. That is, all universal path quantifiers are replaced with the appropriate combination of existential quantifiers and *Boolean* negations. Also "*finally*" operators are converted to "*until*" operators. This returns a new formula that shares absolutely nothing with the original formula (not even the strings). The "*original Formula*" field of each new sub formula is set to point to the formula passed as an argument. In addition, if and only if the original formula is of type *AG, AX, AU, AF, or EF*, the "*converted flag*" is set.

These conversions are as below:

Formulae	Converted Formulae
AX f	*~ EX (~f)*
EF f	*E (True U f)*
AG f	*~ EF(~f)*
AF f	*~ EG (~f)*
A (f U g)	*~E[~g U (~f ∧ ~g)] ∧ ~EG ~g*

Table 1: Conversion of CTL formulae

5 Unwinding

We use usually the Computation Tree Logic (CTL) for specifying these kinds of formula for model checking. It is one of the versions of *temporal logic*. State Transition Graph (STG) is used to derive the computation trees. Now the question is how we can build a tree from the STG. The graph structure is unwound into an infinite tree rooted at the initial state. Figure 2 shows an example of unwinding a graph (traffic-light controller; R, G and Y indicate RED, GREEN and YELLOW respectively) into a tree. All possible computations of the system being modeled are represented by the paths in the tree. Formulae in CTL refer to the computation tree derived from the model. CTL is classified as branching time logic because it has operators that describe the branching structure of this tree.

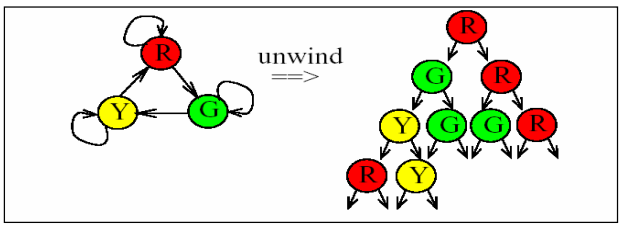

Figure 2: Unwinding of state transition graph.

Now, we may describe some formulae of it. The formula *EG (RED)* is true as there exists at least one path where in all its states; there is RED (the path *R, R, R …….*). The formula E *(RED U GREEN)* is also true as there is at least a path (*R, R, G……*) where all the states hold *R* until we reach a state with *G*. But the formula *AF (GREEN)* is false as there is at least one path in which no state is with the atomic proposition *G*.

6 Expressing properties in CTL

CTL formulas are sometime problematical to interpret. For this, a designer may fail to understand what property has been actually verified. Here we want to add some common constructs of CTL formula used in hardware verification.

1. *AG (Request → AF Acknowledgement)*: For all reachable states (*AG*), if *Request* is asserted in the state, then always at some later point (*AF*), we must reach a state where *Acknowledgement* is asserted. *AG* is interpreted relative to the initial states of the system whereas *AF* is interpreted relative to the state where *Request* is asserted. A common mistake would be to write *Request → AF Acknowledgement* in place of *AG (Request → AF Acknowledgement)*. The meaning of the former is that if *Request* is asserted in the initial state, then it is always the case that eventually we reach a state where *Acknowledgement* is asserted, while the latter requires that the condition is true for any reachable state where *Request* holds. If *Request* is identically true, *AG (Request → AF Acknowledgement)* reduces to *AG AF Acknowledgement*.
2. *AG (AF DeviceEnabled)*: The proposition *DeviceEnabled* holds infinitely often on every computational path.
3. *AG (EF start)*: From any reachable state, there must exist a path starting at that state that reaches a state where *start* is asserted. In other words, it must always be possible to reach the restart state.
4. *EF (x ∧ EX (x∧ EX x)) → EF (y ∧ EX EX z)*: If it is possible for *x* to be asserted in three consecutive states, then it is also possible to reach a state where *y* is asserted and from there to reach in two more steps a state where *z* is asserted.
5. *EF (~Ready ∧ Started)*: It is possible to get to a state where *holds started*, but *ready* does not hold.
6. *AG (Send → A (Send U Receive))*: It is always the case that if *Send* occurs, then eventually Receive is true, and until that time, *Send* must continue to be true.
7. *AG (in → AX AX AX out)*: Whenever *in* goes high, *out* will go high within three clock cycles.
8. *AG(~storage_coke→ AX storage_coke)*: if the coke storage of a vending machine becomes empty, it gets recharged immediately.
9. *AG AF ((~storage_coke ∨ ~storage_coffee) ∧ (storage_coke ∧ storage_coffee)):* the recharge transaction of a vending machine (of coke and coffee) takes place infinitely often.

7 Fairness properties in CTL

In verifying concurrent systems, we are occasionally interested only in correctness along *fair* execution [5]. It is often necessary to introduce some notion of *fairness*. For example, if there are two processes trying to use a shared resource using an arbiter, we may wish to consider only those executions in which the arbiter does not ignore one of its request inputs from either of the processors forever. Alternatively, we may want to consider communication protocols that operate over reliable channels which have the property that no message is ever continuously transmitted but never received. A *fairness constraint* can be an arbitrary set of states, usually described by the formula of the logic. If *fairness constraints* are interpreted as a set of states, then a fair path must contain an element of each constraint infinitely often. If *fairness constraints* are interpreted as CTL formulas, then a path is *fair* if each constraint is true *infinitely often* along the path. The path quantifiers in the logic are then restricted to fair path [6]. An example of a *fairness* condition is *P* that restricts the system to only those paths where *P* is asserted infinitely often. Basically we use *fairness constraints* to rule out undesired executions. Let us discuss this with an example.

In Figure 3 (a), the two processes (*PR1* and *PR2*) want to use the shared resource using the arbiter. Figure 3 (b) shows the corresponding STG. For this example, the atomic propositions are, $AP = \{idle_1, waiting_1, using_1, idle_2, waiting_2, using_2\}$ where, $idle_i$, $waiting_i$ and $using_i$ mean that process *i* is idle, waiting for and using the resource respectively. Here, the state *0* is the initial state. The *AP*s that are "True" in a specific state are expressed by a labeling function *L*. So, from the

Figure 3 (b), we can find that $L(0) = \{ idle_1, idle_2 \}$; $L(1) = \{ waiting_1, idle_2, \}$; $L(2) = \{ using_1, idle_2 \}$; $L(3) = \{ idle_1, waiting_2 \}$; $L(4) = \{ idle_1, using_2 \}$; $L(5) = \{ waiting_1, waiting_2 \}$; $L(6) = \{ using_2, waiting_1 \}$; and $L(7) = \{ using_1, waiting_2 \}$. From this, we can make some assertions about a computation of the above graph. These are a) If at some stage process 1 is waiting then at some later stage it is using the resource (the path (1, 2) or (3, 4) or (5, 6, 7)); b) at no stage both processes are using the resource (all the states); c) If a process is waiting then it does so until it starts to use the resource (as in a)) and d) There is a stage at which both processes are waiting (5).

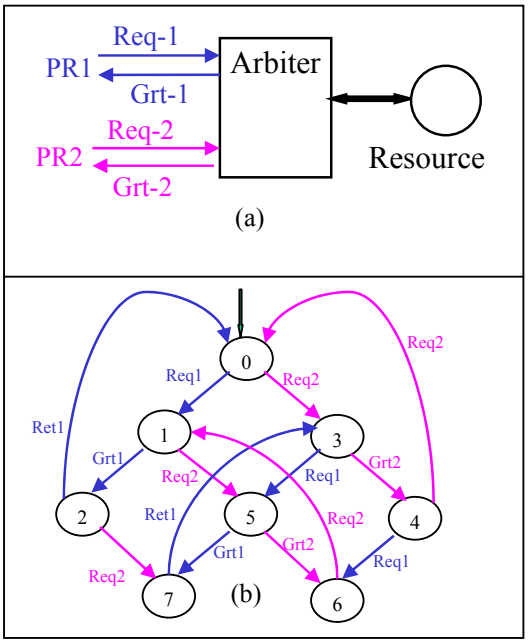

Figure 3 (a): Two processes use the shared resource using the arbiter; (b) the state transition graph

A computation in which 3, 5 and 7 are visited infinitely often but 4 and 6 are visited only finitely often is really unfair. So, it may be the case that the computation in the sequence 3, 5, 7, 3, 5, 7 … goes on infinitely. In this case, process2 never gets the access to the shared resource and as a result the formula $AG\ (waiting_2 \rightarrow AF\ (using_2))$ is "False". But we, of course want to see this formula "True" for the correctness of the design. To avoid such unwilling thing, we can put the constraints that any fair path must visit the any of the states got applying by the rule (~waiting$_2$ V using$_2$). This rule gives us all the states except (3, 5 and 7). Note that, it is clear from the graph that from the states 3, 5 and 7, if we want to hit any others states that must be visited; we have only to options either 4 or 6. In these two states, the process 2 gets the resource to use. So, it satisfies $AG\ (waiting_2 \rightarrow AF\ (using_2))$. For any design, we may need to add more than one *fairness constraints*. Here, we see the same problem if a computation goes infinitely often over the path 0, 3, 4, 0, 3, 4 …or 0, 1, 2, 0, 1, 2…or 1, 5, 6, 1, 5, 6… . Here, we have just tried to narrate the necessity of *fairness* with example.

8 Case Study

In this section we use the controller part of a simple processor, *SAYEH* and we verify this controller by standard *CTL*. The architecture of this processor is simple, but it has enough hardware for our work in formal verification and test and testability research. The processor has a 16-bit data bus and a 16-bit address bus. The processor has 8 and 16-bit instructions. Short instructions may contain shadow instructions, which effectively pack two such instructions into a 16-bit word.

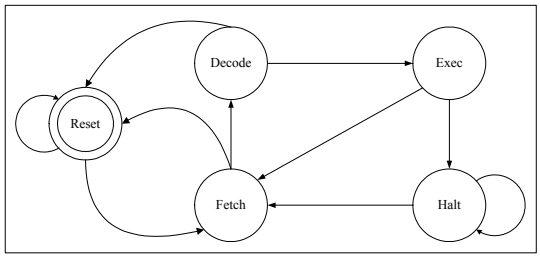

Figure 5: State machine of *SAYEH* Controller

The controller of *SAYEH* has five states*: reset, halt, fetch, decode,* and *exec*. External signals *ExternalReset* and *instruction* control transitions between states of this state machine. The state machine of *SAYEH* controller is shown in Fig. 5.

With *CTL*, all properties of this state machine are written. These properties are in three classes. With these three classes that are explained below, each state machine will be completely verified. The three classes are as follows:

The first class of properties should be checked for all states. This class is divided into three sets of properties:

"There is no deadlock in any state". This property is expressed in ECTL as Equation (1).

$$AG((Pstate = S) \rightarrow EX(Pstate != S)) \quad \forall S \in \{reset, fetch, decode, exec\} \qquad (1)$$

"States are reachable from the initial state (reset)". This property is presented in ECTL as Equation (2).

$$AF(Pstate = S) \quad \forall S \in \{reset, fetch, decode, exec\} \tag{2}$$

"Each state is reachable from any state". This property is shown in ECTL as Equation (3).

$$AG((Pstate = exec) \rightarrow EX(Pstate != reset)) \tag{3}$$

The second class of properties is different from one state to another. In this class of properties *"immediate states after each states"* are checked. For example:

$$AG((Pstate = exec) \rightarrow EX(Pstate != reset)) \tag{4}$$

The third class of properties is to check transitions between states with respect to the input signals and instructions. For example:

$$AG((Pstate = exec \,\&\, External\,\mathrm{Re}set = 1) \rightarrow AX(Pstate != reset)) \tag{5}$$

9 Conclusions

Formal verification replaces simulation in certain applications. For testing the correctness of a digital system that consists of FSMs verification is efficient and easy to use. This is an exact method and does not require test data.

Model checking has many important advantages over the mechanical theorem provers or proof checkers for verification of the different hardware and protocols. In the cases like protocol design, network design and many others where it is more complex and expensive to check/test after the implementation, model checking can be brilliantly used in those cases to find the bug before the implementation. In model checking, CTL, an important family of *temporal logic* is effectively and increasing used to specify the properties of the model to be verified.

10 References

1. E. M. Clarke, J. R. Burch, O. Grumberg, D. E. Long and K. L. McMillan, "Automatic verification of sequential circuit designs", Phil. Trans. R. Soc. Lond. A; pp. 105- 120. 1992.
2. E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," in ACM Transitions on Programming Languages and Systems, 8(2), pp. 244–263, 1986.
3. Jerry R. Burch, M. Clarke, David E. Long, L. McMillan, David L. Dill, " Symbolic Model Checking for Sequential Circuit Verification", IEEE Transactions on Computer-Aided Design of Integrated Circuit and Systems; Vol. 13, No. 4, 1994.
4. Steve Easterbrook and Victor Petrovykh, "*Model-Checking over Multi-Valued Logics*", in *Proceedings of Formal Methods Europe (FME'01)*, March 2001
5. Michael Huth, "Logic in Computer Science: tool-based modeling and reasoning about systems", IEEE Session T1C; 2000.
6. E. M. Clarke, E. A. Emerson and A. P. Sistla, Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic", ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, pp. 244-263, April 1986.
7. Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled, "Book: "Model Checking", The MIT Press; Second printing 2000.