International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 04 Issue: 07 | July -2017 www.irjet.net p-ISSN: 2395-0072

A General Framework for Electronic Circuit Verification

Afsal K

Assistant Professor, Dept. of Computer Science,
Malabar College of Advanced Studies, Kerala, India

---***---

Abstract - *The design and implementation of digital electronic circuits is to undergo proper testing so that bugs can be eliminated from the development environment itself. Since digital electronic circuits and computer programs are similar in nature, methods adopted in program verification can be effectively used for the verification of digital systems also. Formal modelling is an important method in any kind of verification and in this paper, we present a novel approach to formally modeling a digital system. We prove that our approach can be used for modeling digital systems with any number of gates as the resulting model will have a finite number of states only. We use LTL to specify properties of digital systems and we use a symbolic model checker to verify those properties.*

***Key Words*: Digital electronic systems, program verification for digital systems, LTL for specifying properties of digital systems, formal modelling, using SAL for verification**

1. INTRODUCTION

Proving correctness of programs have always been a hot topic in the realm of computer science. The software systems always stressed the importance of testing from the very beginning and as a result, software development life cycle (SDLC) always had an important component in testing [1]. A software does not turn out to be usable unless it passes the test cases prepared by the testing team. Modern day SDLC systems present large number of testing methods to avoid any minor defect in the software.

Although the present black box testing is sufficient for a large number of software applications as a passing criterion, there are certain mission critical applications where the present black box testing is not enough. On June 4, 1996, the unmanned rocket Ariane 5 (from European Space Agency) exploded just after 40 seconds from its launch. On detailed evaluation later, it was found out that a 64-bit integer relating to its velocity component was converted to a 16-bit integer [2]. The system had passed all the black box test cases, though it is not easy for that kind of testing to reveal the bug.

We need a better system which can figure out the bugs and thereby verifying the correctness of the program. Since it is not even possible to run and verify certain systems, it would be really beneficial if there is a testing / verification. system which predicts errors by static analysis of the code / algorithm. Moreover, if the entire software system can be considered as a mathematical model, verification of certain properties would be really easy. That is the whole idea behind program verification.

A program may have a large number of components like variables, semaphores, locking mechanisms etc. and it is really hard to determine and verify all the features of all the components. Hence we limit the components and features. As an example, consider the following program segment.

```
for (;;) {

    x = 0;
    .....
    x = 1;
}
```

Here we would like to verify the properties of x only. Hence we convert the program as a finite state machine with two states, i.e., x=0 and x=1, as shown in figure1.

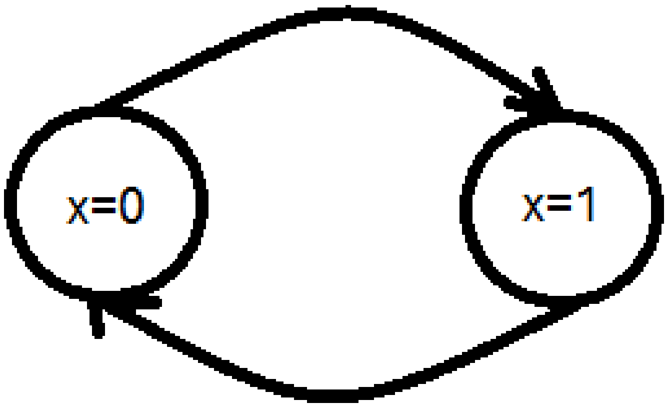

Figure 1.

Transition system for the states of the variable x.

The start and final states of the automaton are not shown in figure 1 as the variable x does not feature them. Hence it is rather a transition system than a deterministic finite automaton. However, the transition system itself is powerful enough to prove certain properties of the program. For example, suppose we want to verify that the value of x is always either 0 or 1. It is very easy for a model checker to do the above, given the above verification condition is

The example given above may seem slightly naive as there is no real bug in the program. However, in real world complex programs, even after multiple peer reviews, it is common that there are semantic errors which go undetected. The development and production environments can be really different and it is very hard to reproduce bugs in development environments which appear in production arena. A black box tester who does not know much about the program can do little in above kind of bugs. Also, it is not easy to reproduce the entire production settings in the development site. That is why formalization comes as a handy tool to find out potential bugs at the development site itself, which otherwise would have been hazardous to the system. As an example, consider the following program segment.

```
   byte temperature = 0;
   bool fan_on = false;
..........

   if (temperature > 500)
      fan_on = true;
..........
```

The fan will never be on as the maximum value temperature can hold is 255. This is a potentially undetected bug if the development environment does not have provision to check temperature beyond 255.

Just like programs, design of every digital electronic system goes through rigorous phases of testing. However, the basic black box testing is not enough in this case also. In this work, we present a few ideas on formalizing digital systems. We then present a novel approach to verifying certain properties of digital electronic systems, using linear temporal logic (LTL). This paper is divided into following sections. Section II gives a brief idea about the basics of LTL. Section III puts forward how digital systems can be truthfully modelled and how LTL can be used for proving certain properties of those systems. Section IV shows the experimental work we have done and section V proposes the conclusions and future work.

2. THE LINEAR TEMPORAL LOGIC (LTL)

Logic is an integral part of formal verification. Temporal logic extends propositional or predicate logic by temporal modalities [3]. LTL implements a transition system where each state holds a proposition. Figure 2 depicts the transition system where a1 is the proposition that holds in the first state. A LTL formula φ can be expressed to be

φ = true | a | $\varphi_1 \wedge \varphi_2$ | $\varphi_1 \vee \varphi_2$ | $\neg\varphi$ | Oφ | $\varphi_1 U \varphi_2$ | ▯φ | ♦φ

where a ∈ AP, the atomic proposition.

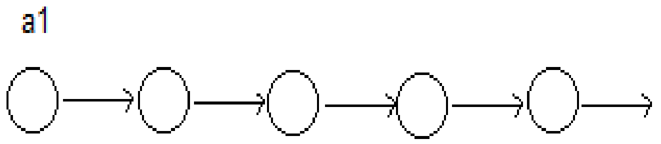

Figure 2.

The various operators and symbols in LTL are

- *Next* operator, O - shows a holds in the second state from beginning. Figure 3 shows Oa_2.
- *Until* operator, U - shows the first proposition continues to hold in all states until the second proposition is met. Figure 4 shows a U b.
- *Eventually* operator, ♦ - shows that the operand is true eventually in the transition system. ♦φ = true U φ Figure 4 shows ♦b.
- *Always* operator, ▯ - as the name indicates, the operand holds in all states of the transition system. Figure 5 shows ▯ a.
- *And* operator, ∧ - used to connect two propositions using logical AND.
- *Or* operator, ∨ - used to connect two propositions using logical OR.
- *Not* operator, ¬ - used to assert negation of the proposition holds.

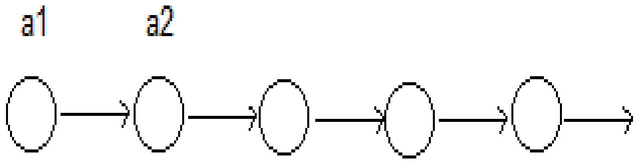

Figure 3. A transition system for the O operator.

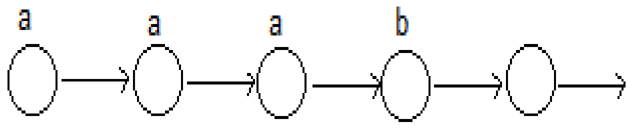

Figure 4. A transition system for the U operator.

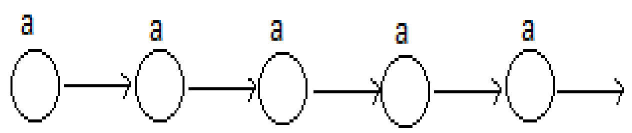

Figure 5. A transition system for the □ operator.

These formulas are very powerful and almost every logical feature of the model can be expressed using LTL. As an example, consider an emailing system where the property is that whenever we try to send an email, the next state should be that it is delivered. It is expressed as a simple LTL formula

(try → O delivered)

Later, if we want to change the property that the emailing system should continue to try sending the email until it is delivered, we change the LTL formula as

(try → try U delivered)

Finally, if the desired property of the system is that whenever we try to send an email, it should be delivered some time in future, the corresponding LTL formula can be written as

(try → ♦ delivered)

We have two more derived semantics for LTL. They are

- infinitely often, □ ♦φ and
- eventually forever, ♦ □ φ

3. LTL FOR DIGITAL ELECTRONIC CIRCUIT VERIFICATION

Digital electronic circuits are mostly made of logic gates and the data they deal with are normally logical zeros and ones. This makes them an ideal candidate for logic verification. Digital electronic circuits can be quite complex and debugging and verifying the circuit is not an easy task. It would be really helpful if they are modelled in a suitable format so that automated verification of certain properties can be done.

LTL is an ideal tool for verification of properties of a digital electronic circuit. An electronic circuit, combinational or sequential, can be considered to produce a finite set of states where each state is an output of a combination of logical operations on the inputs. We can create an exploded model of the circuit by taking each logic gate in the circuit and adding a state corresponding to every combination of the possible inputs to that gate. (For a digital circuit, the possible values of an input are 0 and 1only).

As an example, consider a simple circuit given in figure 6. The possible combinations of inputs for the pair (x1, x2) are (0, 0), (0, 1), (1, 0) and (1, 1). Hence the corresponding transition system model will have five states, as in figure 7.

Lemma 1. The number of states in a transition system model of a digital electronic circuit remains finite.

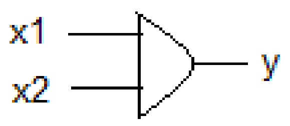

Figure 6. A simple circuit having only one logic gate.

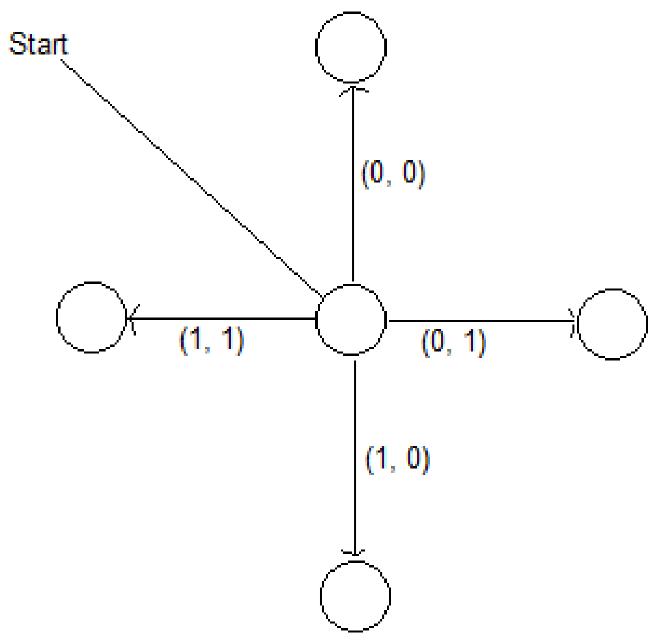

Figure 7. Transition system for the circuit in figure 6.

Proof: We prove the claim using the method of mathematical induction on the number of logic gates in the system, by adding gates progressively one by one. If there is only one gate with k inputs, the maximum number of states in the transition system will be 2^k . For a circuit with m gates, let the no. of states in the transition be n. The maximum number of outputs possible in the above circuit is m. If we add an extra gate to the above circuit (Total number of gates = m + 1) with k external inputs to the new gate, the maximum number of extra states needed in the transition system to accommodate the new gate will be 2^{m+k} Since the total number of gates in any electronic circuit is finite, the total number of states needed in the transition system also remains finite.

3.1 Using LTL for verification

Debugging a complex digital circuit is never easy. The debugger has to go through output of each state correspond- ing to each combination of inputs and obviously this is a tedious and error prone process. However, once we have a model of the circuit, each state corresponds to an output for an input combination. Verifying whether a particular output is obtained is simply a graph reachability [4] problem through a specific path. There may be other aspects of verification also. For example, under certain conditions, the output should never be a particular value. These properties can be easily expressed in LTL and then can be given as an input to a model checker. The model checker can easily verify whether the LTL property is satisfied by the system.

3.2 SAL - A symbolic model checker

SAL (Symbolic Analysis Laboratory) is a framework used for model checking, program analysis and theorem proving of transition systems [5]. The language of SAL allows the users to specify the transition systems. Just like many other model checking frameworks, SAL input language allows the transition system to be specified in terms of initialization and transition commands. It has a binary decision tree (BDD) based symbolic model checker for finite state systems. Most importantly, LTL properties can be specified using the language of SAL which the symbolic model checker can easily verify. SAL generates a counter example sequence of steps if it is not able to verify a property.

We use SAL for model checking of digital electronic circuits.

4. EXPERIMENTS AND RESULTS

The modeling method suggested in section III has an exponential complexity. The state space grows really big as the number of gates and inputs increase. As a result, the number of lines of code for specifying the transition system in SAL is considerably high for an average electronic circuit that uses tens of gates. This results in prolonged execution time for the model checker and a considerable use of system resources.

However, our first aim is to verify the result for all possible combinations of inputs and the result is obtained by performing a series of logical operations on the inputs. SAL allows to specify these operations directly in the transitions. For example, consider the circuit shown in the figure 8. The circuit has three gates and one input, viz. x. The final output is y_2 and the intermediate outputs are y_0 and y_1. They can be represented as

y_0 =xANDy$_2$,y_1 =NOT(xANDy$_1$), y_2 =y_0 ANDy$_1$

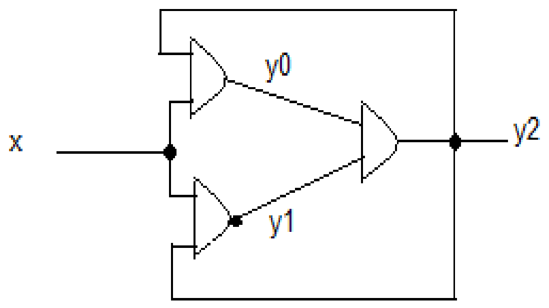

Figure 8. A circuit using three gates and one input.

4.1 Modeling using SAL

As we are interested in the output only, we need only two variables. The type of the variable is boolean.

```
  input x: boolean
  output y2: boolean
```

The initial states of the above variables are indeterminate. Hence there is no initialisation section for our model.

Transition is the most important stage that helps us to truthfully model the circuit. Since SAL allows the logic operations that are performed by gates, we can directly model them as follows.

```
transition

[

     true --> y2’ = (x AND y2) AND
                    (NOT(x AND y2));
[]  else -->   % Do nothing

]
```

4.2 Specifying properties using LTL

Once we model the entire circuit, we can specify the properties as LTL. Here we verify the basic property that once the input is 0, the output continues to be 0 forever. This can be represented in LTL as

$\neg x \rightarrow \blacklozenge \neg y_2$ The corresponding SAL statement is

```
low_output: THEOREM circuit |-
        ((x = false) => (F(G(y2 = false))));
```

It specifies that in the module circuit, the LTL property low_output holds. *C. Results*

We use the symbolic model checker of SAL to verify the LTL assertions. For the above sample, it verified the assertion quickly.

We validated the above method of verification using sample circuits like flip-flops and counters and all the time, the results were satisfactory.

5. CONCLUSIONS AND FUTURE WORK

We have presented an easy way to convert a digital electronic circuit to a transition system and then to verify their properties. The method uses the BDD based symbolic model checker of SAL. The method can verify almost any feature expressable in LTL.

As the future work, we would like to create a system that can automatically convert digital systems to the transition system based model. We need to identify methods to reduce

the number of states in the system. We would also like to extend the work to some complex electronic circuits and see if there is any upper bound on the number of gates and inputs.

REFERENCES

[1] C. Bayrak, M. Sahinoglu, T. Cummings, High assurance software testing in business and DoD, in: Proceedings of the Fifth IEEE International Symposium on High Assurance Systems Engineering (HASE 2000), 2000, pp. 207 - 211.

[2] G. Le Lann, An analysis of the Ariane 5 flight 501 failure-a system engineering perspective in: Proceedings of International Conference and Workshop on Engineering of Computer-Based Systems, 1997, pp. 339 - 346.

[3] Dov M. Gabbay, A. Kurucz, F. Wolter, M. Zakharyaschev, Many-dimensional modal logics: theory and applications, El- sevier, p. 46.

[4] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, Adding regular expressions to graph reachability and pattern queries, in: Proceedings of the Twenty Seventh International Conference on data Engineering (ICDE), 2011, pp. 39 - 50.

[5] SAL: Tutorial, http://sal.csl.sri.com/doc/salenv tutorial.pdf