An Efficient Event Based Approach for Verification of UML Statechart Model for Reactive Systems

C.M. Prashanth [1], K.Chandrashekhar Shet [2], Janees Elamkulam [3]

[1,2] *Department of Computer Engineering, National Institute of Technology Karnataka*
Srinivasanagar Post, Surathkal, Karnataka, India - 575 025.
[1] prashanthcm@nitk.ac.in
[2] kcshet@nitk.ac.in

[3] *IBM India Pvt. Ltd,.*
Old airport road, Bangalore, Karnataka, India
janees.ek@in.ibm.com

***Abstract*—This paper describes an efficient method to detect safety specification violations in dynamic behavior model of concurrent/reactive systems. The dynamic behavior of each concurrent object in a reactive system is assumed to be represented using UML (Unified Modeling Language) statechart diagram. The verification process involves building a global state space graph from these independent statechart diagrams and traversal of large number of states in global state space graph for detecting a safety violation. In our approach, a safety property to be verified is read first and a set of events, which could violate this property, is computed from the model description. We call them as "relevant events". The global state space graph is constructed considering only state transitions caused by the occurrence of these relevant events. This method reduces the number of states to be traversed for finding a property violation. Hence, this technique scales well for complex reactive systems.**

As a case study, the proposed technique is applied to verification of Generalized Railroad Crossing (GRC) system and safety property "When train is at railroad crossing, the gate always remain closed" is checked. We could detect a flaw in the infant UML model and eventually, correct model is built with the help of counter example generated. The result of the study shows that, this technique reduces search space by 59% for the GRC example.

I. Introduction

The software systems are growing in size and complexity at a rapid rate. As a result, the task of timely delivery of quality software has become extremely difficult. Software verification techniques based on model checking have been an option for developers to build quality into software[1]. In the early 1990's, several model checking tools such as SPIN (Simple Promela INterpreter)[2], SMV (Symbolic Model Verifier)[3], SLAM[4], BLAST (Berkeley Lazy Abstraction Software verification Tool)[5]and RULE BASE[6] are developed. However, to use model checking tool, user must first develop a formal model of the application in native language of the tool. In larger applications, the effort to manually construct the formal model is considerable investment of time and expertise. Moreover, input language of most of the model checking tools is text based and lacks advantages of visual representation. In recent time, the UML (Unified Modeling Language) has become de facto standard for modeling the software systems. It has rich set of visual notations and diagrams for modeling. The literature suggest methods for verifying properties using off-the-shelf model checking tools, but issues like translation of model to model checker's input language and handling state explosion are very hard to address ([7],[8],[9],[10]). We have elaborated these issues in our previous papers [11],[12].

In this paper, we describe a method for verifying the reactive systems modeled using UML statechart diagrams without the aid of model checking tool. The statechart diagram for modeling complex systems is first proposed by David Harel in 1987[13]. Statecharts are extended state-transition diagrams with the notions of hierarchy, concurrency and communication. The reactive systems considered here are state oriented and respond to the occurrence of internal or external events. The response may result in change in state and also actions. For example, in a client-server system, client's request message (event) will change the server's sate from idle to busy and server respond with an acknowledgement message (action). Therefore, a reactive (event-driven) system's behavior is specified by set of states, events and actions.

The work described in this paper is an extension of the idea proposed in our earlier paper [14]. The technique presented in section 2, aims at reducing the number of states to be checked for detecting safety violation in the behavior of the reactive systems. We further state that, with this approach, systems with large state space can be verified and translation of UML statechart diagrams to input language of the model checker is avoided. To justify this, in section 3, we describe a case study of verifying UML statechart model of Generalized Railroad Crossing (GRC) system. The performance of the verification technique, applied to GRC is discussed in the section 4. The findings of this investigation are summarized in the section 5.

II. Proposed verification technique

A. Assumptions

It is assumed that, the system under consideration has multiple cooperative objects. These objects communicate via events. The dynamic behavior of the each object is modeled using UML statecharts. The objects change their state upon receiving an appropriate externally or internally generated event & the corresponding guard condition becoming true. The property to be verified is expressed in temporal logic and represented by the symbol ϕ. The verification process involves the translation of each UML statechart to the form of a tuple $\{S_i, E_i, T_i, I_i\}$,

Where

i represents an object.
S_i is a non empty finite set of states.
E_i represent set of events.
$T_i \subseteq S_i$ X S_i is a set of transitions.
$I_i \subseteq S_i$ is a set of initial states.
Let E_t be set of total events, i.e $E_t = \{E_1 \cup E_2 \mathinner{..} E_n\}$, Where n is number of objects in the system.

B. Event based verification approach

The state space of the system is built by combining (cartesian product) the state transitions of all objects upon occurrence of each event in E_t. Then the error state (negative behavior) represented as $\neg\ \phi$ is searched in the state space graph. The error state is checked during the construction of the state space (on-the-fly); if found further exploration of the state space is terminated and the error trace (counter example) is displayed.

The state transition of an object completely depends on externally or internally generated events and any technique which reduces the number of events to be considered for constructing state space graph will ultimately reduce the search space. The approach that is described in this section is based on this idea. This algorithm finds set of "relevant events" from the UML statechart of each object of the system. The union of all these set of relevant events constitutes the set Er_t. The relevant events are computed based on the undesired behavior ($\neg\ \phi$) looked for in the model and using the following rules:

R1: An event is relevant if
- R 1.1: there is a transition associated with this event and has current sate as part of error state ($\neg\ \phi$).
- R 1.2: there is a transition associated with this event and has next state as part of error state ($\neg\ \phi$).

R2: A set of events are relevant if
- R 2.1: there is a sequence of transitions associated with these events and takes the object from the initial state to a state, which is part of error state ($\neg\ \phi$). In other words, all events that participate in changing state of an object from its initial state, subsequently to a state which is part of the error state.

After set of relevant events is computed, UML statechart of each object is translated to a from of tuple $\{S_i, Er_i, T_i, I_i\}$, Where Er_i represent set of relevant events associated with an object O_i and Er_t set of total relevant events, i.e, $Er_t = \{Er_1 \cup Er_2 \mathinner{..} Er_n\}$. The state space is explored considering only the events in the set of total relevant events (Er_t). The moment error state is reached or all states are visited, further state space exploration is terminated. Thus, it saves the memory and handles systems with large state space (The exact amount of memory saved varies with the complexity of the system and property being verified). The flowchart and the algorithm of the above explained approach are shown in Fig.1 & Fig.2 respectively. This approach is very much suitable for verification of safety properties of a complex systems having considerable number of non-relevant events.

In the next section, we illustrate verification procedure by applying the above described algorithm to a benchmark case study, the "Generalized Railroad Crossing"(GRC) problem introduced by Heitmeyser et al[15].

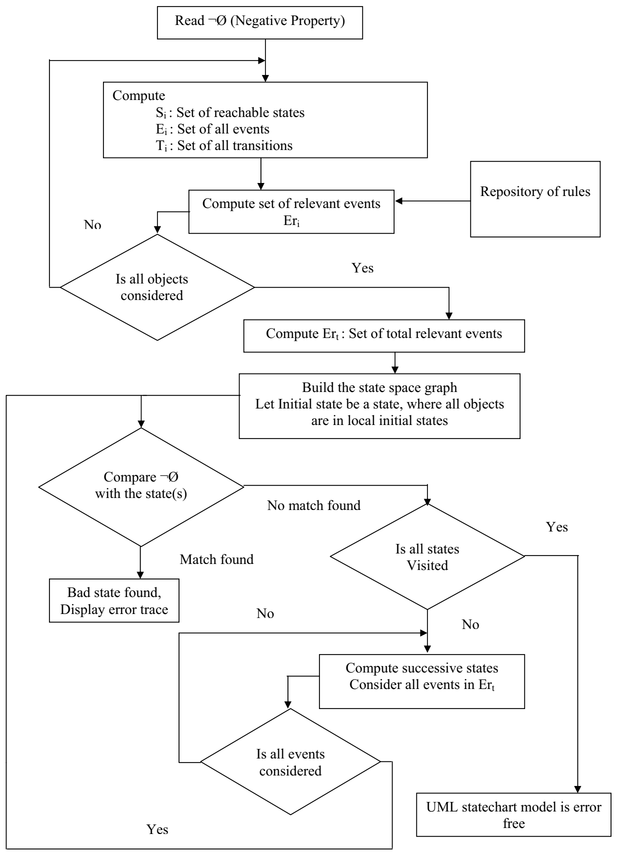

Read ¬Ø (Negative Property)
Compute
Sᵢ : Set of reachable states
Eᵢ : Set of all events
Tᵢ : Set of all transitions
Compute set of relevant events Erᵢ
Repository of rules
No
Is all objects considered
Yes
Compute Erₜ : Set of total relevant events
Build the state space graph Let Initial state be a state, where all objects are in local initial states
Compare ¬Ø with the state(s)
No match found
Match found
Is all states Visited
Yes
Bad state found, Display error trace
No
No
Compute successive states Consider all events in Erₜ
Is all events considered
Yes
UML statechart model is error free

Fig. 1: Flowchart

III. A CASE STUDY

A. Generalized Railroad Crossing(GRC) example

The Generalized Railroad Crossing system is expected to operate a gate at a railroad crossing (RC). The gate for two railroad tracks lies in an area of interest (A). The trains move in both the directions (left to right, right to left) on two tracks (T1, T2). The trains travel at different speeds and can pass each other. It is assumed that no two trains are allowed to move in opposite direction in A on same track at any point of time. There are sensors (S1, S2, S3, S4 & S5) positioned as shown in the Fig.3. The sensors indicate when the train

```
Read $\neg\emptyset$ (negative behavior or bad state) from the user;
 for each object i of the system (model)
 {
        Get $S_i$ set of reachable states;
        Get $Er_i$ set of all relevant events;
        Get $T_i$ set of all transitions;
        Get $I_i$ set of initial states;
 }
for (i=1 to No. of objects)
Compute $Er_t$= ($Er_t$ U Get_relevant_events ( $O_i$ ) );
// Build the state space (synchronous product of all objects)//
Start with state s (all objects are in their initial states);
 for (each relevant_event e∈ $Er_t$ enabled in s & s not empty)
 {
        $s^*$ = set of all successor states of s after $e_i$;
        While ($s^*$ not empty)
        {
               If (state $s_j \in s^*$, is not in state space)
               {
                      add $s_j$ to state space;
                      push $s_j$ on to stack;
                      If (state $s_j$ is same as $\neg\emptyset$)
                      {
                             Set found flag to true;
                             Break;
                      }
               Mark the state $s_j$ as visited;
               }
        $s_j$ = nextstate ($s_j$);
        }
If (found) Break;
s= pop ();
}
If (found)
        Display "No negative behavior seen in the model";
 Else
 {
        Display "Negative behavior found";
        Display Error Trace / Counterexample;
}
```

(a) Main routine

```
SET Get_relevant_events ( $O_i$ )
{
      Let $Er_i$ = EMPTY;
      while ( $T_i \neq$ EMPTY)
      {
            If $t_j$ ∈ $T_i$ has current state which is part of $\neg\emptyset$
                  add corresponding e to $Er_i$;
            If $t_j$ has next state which is part of $\neg\emptyset$
                  add corresponding e to $Er_i$;
            $t_j$ = next_transition in $T_i$;
      }
      while ($I_i \neq$ NULL)
      {
            s = one of the initial states $I_i$;
            while ($T_i \neq$ EMPTY)
            {
                  Find all $t_i$ ∈ $T_i$, taking initial
                  state s to state, which is part of $\neg\emptyset$;
                  add corresponding set of events E to $Er_i$;
            }
            s = next_element in $I_i$;
      }
      return ($Er_i$);
}
```

(b) Routine to compute relevant events

Fig. 2: Algorithm

arrives to region A, leaves the region A, enter RC & exit RC. The sensor S5 indicate whether gate is closed or open. The "occupancy interval" is defined as a time interval during which one or more trains in RC. The system is expected to satisfy the following properties

1. The gate is closed during all occupancy intervals (Safety)
2. The gate is open, if there is no train in the occupancy interval (Utility)
3. The gate is open for as much as time possible (Live ness)

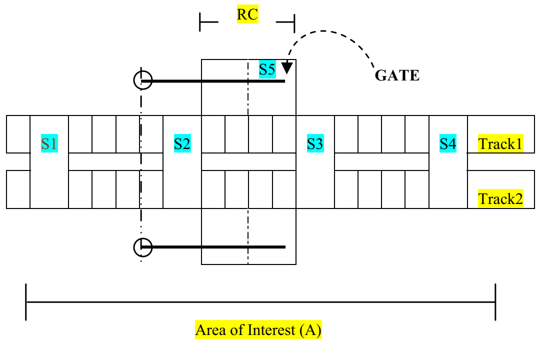


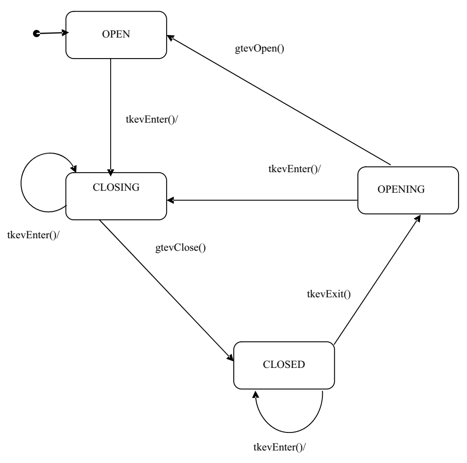
Fig. 3: Railroad crossing

B. *UML statechart model of GRC*

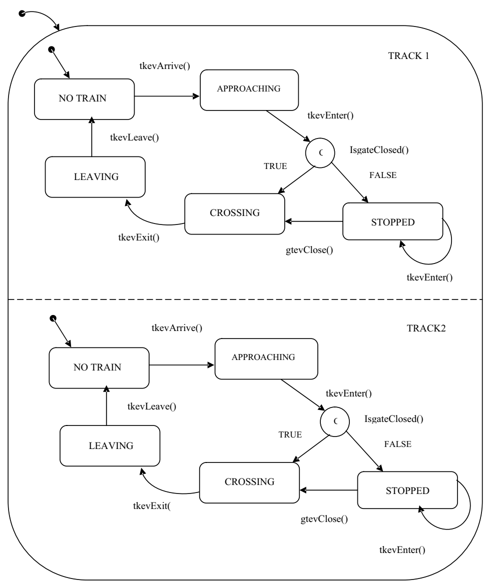
The dynamic behavior of objects Gate and Track of GRC system are specified using UML statechart diagrams as shown Fig.4. The UML statechart for Gate in Fig.4(a) shows an initial state and four simple states viz., Open, Closing, Closed and Opening. The gate reacts to external signals by opening & closing of gate. The UML statechart for Track in Fig.4(b) shows concurrent composite state consisting of two orthogonal regions for each track (Track1 & Track2), which are again having sequential states (OR states). Each orthogonal region has an initial sate and five simple states viz., No train, Approaching, Crossing, Stopped and Leaving. The transition from source states to target states can be possible, when an appropriate signal/event (given as label on the arrows in Fig.4) is triggered. All the events considered are listed in the table I.

TABLE I: Events associated with GRC model

Event	Code	Description
tkevarrive	1	Event by the track object, when train arrives at A.
tkeventer	2	Event by the track object, when train enters RC.
tkevexit	3	Event by the track object, when train exits RC.
tkevleave	4	Event by the track object, when train leaves A.
gtevclose	5	Event by the gate object, when gate is closed.
gtevopen	6	Event by the gate object, when gate is opened.



(a) Statechart for the object GATE



(b) Statechart for the object TRACK

Fig. 4: UML state chart model for GRC

C. State space construction

The state space is constructed from the description of the system in UML statechart diagrams. The dynamic behavior of all objects are combined to generate state space graph which represents the behavior of the entire system. The notion of ”Universe” (U) is useful in describing the construction of state space. It is the set of all possible combinations of local states of the objects of a system. The UML statechart model of the GRC system (see Fig. 4) has two objects Gate and Track, the Track object has two orthogonal states Track1 and Track2. The Gate object has 4 local states, Track1 has 5 local states and Track2 has 5 local states. The U for GRC system will contain (4 X 5 X 5) 100 states. It is common that the model restricts the number of reachable states. Thus set of possible states of state space is always a subset of U. As per UML model, the state space of the GRC system contains 46 states. The table II shows all possible states.

TABLE II: All possible states

Sl.No.	Gate status	Track1 status	Track2 status
S1.	Open	Notrain	Notrain
S2.	Open	Notrain	Approaching
S3.	Open	Notrain	Crossing
S4.	Open	Notrain	Leaving
S5.	Open	Approaching	Notrain
S6.	Open	Approaching	Approaching
S7.	Open	Approaching	Crossing
S8.	Open	Approaching	Leaving
S9.	Open	Crossing	Notrain
S10.	Open	Crossing	Approaching
S11.	Open	Crossing	Leaving
S12.	Open	Leaving	Notrain
S13.	Open	Leaving	Approaching
S14.	Open	Leaving	Crossing
S15.	Open	Leaving	Leaving
S16.	Closing	Notrain	Stopped
S17.	Closing	Stopped	Notrain
S18.	Closing	Stopped	Stopped
S19.	Closing	Stopped	Approaching
S20.	Closing	Stopped	Crossing
S21.	Closing	Stopped	Leaving
S22.	Closing	Approaching	Stopped
S23.	Closing	Crossing	Stopped
S24.	Closing	Leaving	Stopped
S25.	Closed	Notrain	Crossing
S26.	Closed	Approaching	Crossing
S27.	Closed	Crossing	Notrain
S28.	Closed	Crossing	Approaching
S29.	Closed	Crossing	Crossing
S30.	Closed	Crossing	Leaving
S31.	Closed	Leaving	Crossing
S32.	Opening	Notrain	Notrain
S33.	Opening	Notrain	Approaching
S34.	Opening	Notrain	Crossing
S35.	Opening	Notrain	Leaving
S36.	Opening	Approaching	Notrain
S37.	Opening	Approaching	Approaching
S38.	Opening	Approaching	Crossing
S39.	Opening	Approaching	Leaving
S40.	Opening	Crossing	Notrain
S41.	Opening	Crossing	Approaching
S42.	Opening	Crossing	Leaving
S43.	Opening	Leaving	Notrain
S44.	Opening	Leaving	Approaching
S45.	Opening	Leaving	Crossing
S46.	Opening	Leaving	Leaving

D. Event based algorithm applied to GRC

The safety property to be checked in the GRC model “When the train is at RC on Track1 or Track2, the Gate remain closed” is expressed in temporal logic as follows:

$$(\text{T1.Crossing} \vee \text{T2.Crossing}) \Longrightarrow \text{G.Closed}$$

In our approach, the above mentioned assertion is changed into negative and treated as an invalid behavior (safety violation). This invalid behavior is then proved wrong or correct by pruning the state space. If the claim is found correct then the model has a flaw and counter example/error trace is generated (path from the initial state to error state). The above stated assertion can be written in the negative form as follows:

$$(\text{T1.Crossing} \vee \text{T2.Crossing}) \implies \neg\ (\text{G.Closed})$$

This means that the train is crossing, when the gate is in open or opening or closing state.

Once the property to be verified is read, the set of relevant events from the UML statechart model by applying the rules stated in the section II-B is computed. The relevant event sets obtained for GRC case are Er_{Gate} = {gtevopen, gtevclose}, Er_{Track} = {tkevarrive, tkeventer, tkevexit} for objects Gate and Track respectively. The total set of relevant events is then computed by taking the union of Er_{Gate} and Er_{Track}, thus Er_t = {gtevopen, gtevclose, tkevarrive, tkeventer, tkevexit}. The "tkevleave"(see table I) is considered as the non relevant event and ignored during the construction of the state space. The Fig.5 shows the exploration of the state space by considering only the events in the set Er_t.

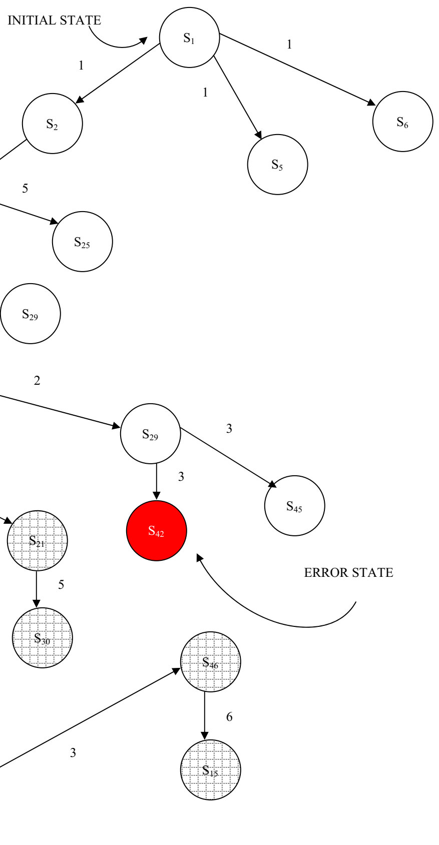


Fig. 5: State space exploration

The exploration starts from the initial state S_1 (see table II) and continued till a state is reached, which does not respond to any of the relevant events. Then we back track to a state which responds to one of the events in the set Er_t. The algorithm terminates when an error state is reached or no state is left for further exploration.

In the Fig.5, state exploration starts with initial state S_1. The set of successive states $\{S_2, S_5, S_6\}$(See table II) upon event "tkevarrive" are computed. The state S_2 is then picked randomly for further exploration, this is continued till the state S_{15} is reached, which does not respond to any of the relevant events. We then back track till the state S_{29} is reached, which leads to state S_{42} (darkened state in Fig.5) on event "tkevexit" (see table I). The state S_{42} is a bad state as it violates the safety property(i.e, when one of the train is at the crossing, the gate starts to open). Once the state exploration is terminated, the counter example or the error trace is generated as shown in Fig.6.

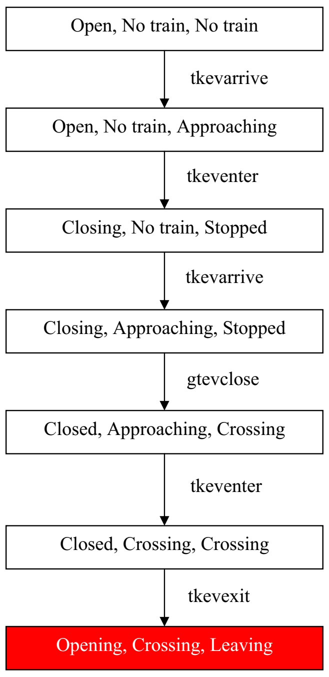

Open, No train, No train
tkevarrive
Open, No train, Approaching
tkeventer
Closing, No train, Stopped
tkevarrive
Closing, Approaching, Stopped
gtevclose
Closed, Approaching, Crossing
tkeventer
Closed, Crossing, Crossing
tkevexit
Opening, Crossing, Leaving

Fig. 6: Error trace or Counter example

IV. Results and Discussion

A. *Refinement of GRC model*

The error trace shown in Fig.6 depicts that, the Gate is allowed to open, as and when one of the trains crosses the RC and this leads to the bad state. This flaw in the model can be avoided by making sure that no train is in the occupancy interval, before allowing the Gate to open. The corrected UML statechart of the Gate object is shown in Fig.7. We have added a global variable "train Count" to the model, which is incremented every time a train enters the crossing and decremented every time a train leaves the crossing. There by we ensure that no train is at crossing, when the Gate begins to open. This results in a correct GRC model.

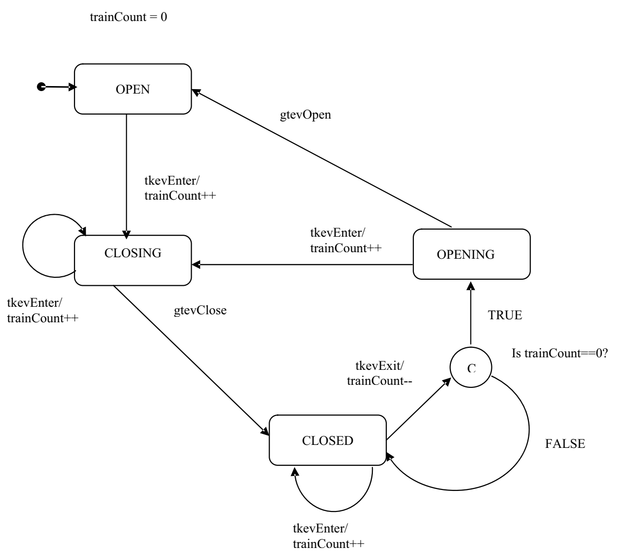


Fig. 7: Corrected UML statechart for the object GATE

B. Performance of the algorithm

The algorithm is evaluated based on the ability to reduce the state space during the state exploration. The result obtained by applying the algorithm to GRC system is shown in table III. It is found that, for detecting the safety property violation in the UML statechart model of the GRC system with state space of 46, the event based algorithm explores only 41% of the complete state space and generates counter example of length 6. Where 6 indicates number of hops to error state from the initial state.

TABLE III: Performance of the algorithm

State space	**States explored**	**Error path length**
46	19	06

V. Conclusions

A majority of the existing approaches translate UML statechart model into text based modeling language which is then verified using off-the-shelf model checker. The proposed verification technique does not translate UML statechart models to the text based language of the model checker, as it takes visual model as the input.

In this paper, we have described an event based algorithm for the verification of safety property violations in UML statechart model of reactive systems. The correctness of the verification technique has been illustrated taking "Generalized Railroad Crossing (GRC)" as a case study.

This approach considers only relevant events for the construction of the state space. This reduces state space significantly (59 % for GRC example) and produces error trace of shorter length (6 for GRC).

We have verified the UML statechart model of the GRC system for compliance of the safety property "The gate is closed during all occupancy intervals" and found a flaw in the initial model. We later corrected it by attaching a global variable "train count" to the model. The "train count" = 0 ensures no train is at crossing,when gate is open.

Acknowledgment

The authors wishes to acknowledge all those (at department of Computer Engineering, National Institute of Technology Karnataka, Surathkal and IBM India Private Limited, Bangalore, India), who have given valuabe suggestions to improve the quality of the work presented in this paper.

References

[1] E. M. Clarke, O. Grumberg, and D. A.Peled, *Model Checking*. MIT Press, 1999.

[2] G. J. Holzmann, "The model checker spin," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[3] K. L. M. Millan, "Symbolic model checking:an approach to the state explosion problem," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania, United States, May 1992. [Online]. Available: http://handle.dtic.mil/100.2/ADA250924

[4] (2007) The slam project of microsoft laboratory. [Online]. Available: http://research.microsoft.com/slam/

[5] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Proc. of 10th SPIN Workshop on Model Checking Software (SPIN), LNCS 2648*. Springer-Verlag, 2003, pp. 235–239.

[6] I. Beer, S. Ben-David, C. Eisner, and Landvar, "Rulebase-an industry-oriented formal verification tool," in *Proc. of 33rd Design Automation Conference(DAC)*. Asociation for Computing Machinery, 1996, pp. 655–660.

[7] E. Mikk, Y. Lakhnech, G. Holzmann, and others., "Implementing statecharts in promela/spin," in *Proc. of 2nd IEEE workshop on industrial strength formal specification techniques WIFT'98*, 1998, pp. 90–101.

[8] E. M. Clarke and W. Heinle, "Modular translation of statecharts to smv," Carnegie Mellon University, School of computer science,Pittsburgh, Technical report CMU-CS-00, 2000.

[9] P. R. Gluck and J. Holzmann, "Using spin model checking for flight software verification," in *Proc. of IEEE Aerospace Conference*, vol. 1, Big Sky MT USA, Mar. 2002, pp. 1–105–1–113.

[10] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models, LNCS 1491*. Springer-Verlag, 1998, pp. 429–452.

[11] C. M. Prashanth, K. C. Shet, and J. Elamkulam, "A survey of model checking the UML statechart model of embedded systems," in *Proc. of National Conference on Emerging Trends in Engineering & Technology, Frontier'07*, 2007, pp. 149–155.

[12] C. M. Prashanth, K. C. Shet, and J. Elamkulam, "A reality chek of model checking the UML statechart diagrams and research directions," in *Proc. of International conference on Computers,Communication,Control systems and Instrumentation 3CI-2007*, 2007, pp. 16–22.

[13] D. Harel, "Statecharts: A visual formalism for complex systems," *Science Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[14] C. M. Prashanth, K. C. Shet, and J. Elamkulam, "Verification framework for detecting safety violations in UML statecharts," in *Proc. of Second Asia International conference on Modeling and Simulation (AMS 2008)*. IEEE computer society, May 2008, pp. 849–854.

[15] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Comparing different approaches for specifying and verifying real-time systems," in *Proc. of 10th IEEE workshop on Real-Time Operating Systems and Software*, 1993, pp. 122–129.