Efficient Algorithms for Verification of UML Statechart Models

C.M. Prashanth[1], K.C. Shet[2]
Dept. of Computer Engineering
National Institute of Technology Karnataka, Surathkal, INDIA
Email: {prashanthcm[1], kchshet[2] }@nitk.ac.in

Abstract— **In this article, we present algorithms devised for the automatic verification of UML(Unified Modeling Language) statechart models. The basic algorithm checks the safety property violation during the construction (on-the-fly) of the state space graph and if any property violation is found, it generates a counter example. The second algorithm builds the state space considering only those events, which could lead to the negative behavior of the system. In other words, a set of relevant events is generated first and state space is constructed considering only the state transitions of the objects caused by these relevant events. Thus search space is reduced in both the methods. As a case study, we have verified UML statechart model of the Generalized Railroad Crossing (GRC) system using the proposed algorithms. The safety property "When the train is at rail road crossing, the gate always remain closed" is verified. We could detect property violation in the initial UML statechart model of GRC and eventually it is corrected with the help of the counter example generated by the algorithms. The case study results show that event based verification algorithm yields 59% reduction in the state space for the GRC example.**

Index Terms— **Software verification, Model checking, Statechart, Unified Modeling Language, Reactive systems**

I. Introduction

Model driven software development has been a prominent means to enhance the understandability of the system's structure and behavior. It has prompted industries to develop tools which can generate the code from the model in high level languages like C, C++ or JAVA (IBM's Rational Rose RT [1] is one such tool used for the development of embedded real time systems). Therefore ensuring model's correctness becomes highly essential. The traditional way of verifying software systems is through human inspection, simulation, and testing. Though these methods are cost effective, unfortunately these approaches provide no guarantee about the quality of the software. The human inspection or code review is limited by the abilities of the reviewers. Simulation and testing can only explore a minuscule fraction of the state space of any software system.

The commonly used formal model verification technique is model checking. Model checking is a pragmatic technique that, given a finite-state model of a system and a logical property, systematically checks whether model holds the property or not. If the model does not hold the expected property, an error trace (counter example) is generated [2]. The original model can be refined by leveraging information given by the counter example, this approach is known as counter example guided model refinement [3]. Several model checking tools like SPIN (Simple Promela INterpreter) [4], SMV (Symbolic Model Verifier) [5], SLAM [6] and RuleBase [7] are in existence.

The major drawback of using model checking tools for verification is that, they expect system to be modeled using their proprietary input language. The input language of most of these tools are text based and lacks advantages of visual representation. Therefore, UML satechart diagrams (provides visual representation) are used for designing the reactive systems. In this article, we describe our approach for verifying the reactive systems modeled using UML statechart diagrams and we do not take the aid of model checking tool. The statechart diagrams for describing dynamic behavior of systems is first proposed by David Harel in 1987 [8]. The statechart diagrams 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 of state and also actions. For example, in a client-server system, client's request message (event) will change server's state from idle to busy and the server responds with an acknowledgement message (action). Therefore, a reactive (event-dreiven) system's behavior is specified by set of states, events and actions.

The work described in this article is an extension of the idea proposed in our earlier paper [9]. We have made a critical inspection of related work in the section II. In the section III, a brief introduction to the methodology followed for verification is given. In section IV, algorithms devised to verify safety properties of reactive systems are presented. In section V, we have presented a case study of verifying Generalized Rail road Crossing (GRC) system. The results and performance of the verification techniques are discussed in the section VI. The findings of this investigation are summarized in section VII.

This research work is partially supported by Center for Advance Studies (CAS), IBM India Pvt. Ltd., Bangalore. Manuscript received on 1^{st} July 2008, revised on 21^{st} Nov 2008 and accepted on 10^{th} Feb 2009

II. Related work

In this section, we describe the existing works on verification of statechart models, which uses model checking techniques. We begin with a discussion on earlier works related to verification of variants of statechart models like RSML (Requirements State Machine Language) [10] and STATEMATE. Later, we discuss prior works related to verification of UML statecharts and list out the research challenges need to be addressed.

In 1998 William Chan et.al. demonstrated the application of software model checking technique to verify properties of an aircraft Traffic alert and Collision Avoidance System (TCAS-II) [10]. The requirements specification written using RSML, is translated to input language of the model checker SMV [5]. The SMV is used to successfully verify the robustness properties and safety critical properties of the system. In the same year G.J.Holzman et.al. described an approach for the verification of safety properties of a hypothetical production cell system in [11]. The STATEMATE statechart representation of the production cell system is verified using SPIN model checker. The statechart representation is translated into EHA (Extended Hierarchical Automata) and then to PROMELA (PROcess MEta LAnguage), the input language of the SPIN. In recent times, UML has become defacto standard for model driven development. The dynamic behavior of the realtime systems is specified using UML statechart diagrams. There are efforts to demonstrate application of software model checking techniques to verify these diagrams ([12], [13] and [14]). In these articles, translation of UML statechart models to input language of the model checker is not emphasized. Subsequently, many tools such as vUML [15], vPROMELA [16], HUGO [17] and TABU [18] are developed. These tools gave importance to translation process. vUML and HUGO tools use the information contained in UML class diagrams, statechart and collaboration diagrams of a UML model to construct PROMELA specification and uses SPIN as verification engine. The vPROMELA tool is a graphical user interface to the SPIN. It allows us to describe the hierarchies of behavior and of system structure. The visual notations are then translated into PROMELA using translation constructs described in [16]. The TABU (Tool for the Active Behavior of UML) reads the UML specification (statechart, class and activity diagrams) represented in XMI format and translate it to SMV specification for verification.

In a nutshell, there are two prevailing model checking technologies, symbolic and explicit model checking. In symbolic model checking, the state space graph is represented by Binary Decision Diagrams (BDD) and property verification is done by searching the state space. BDDs are directed acyclic graph obtained by removing isomorphic sub trees [19]. The SMV is a BDD based model checker for verifying the properties expressed in temporal Computational Tree Logic (CTL) [20]. On the other hand, in explicit model checking, states are explicitly enumerated in the state space and property verification corresponds to a systematic search of the state space. Explicit model checking has proven to be very successful as it can handle irregularly structured models. The significant question is, can these model checking technologies be effectively used for automatic verification of UML statechart models for reactive systems. As discussed in the previous paragraph there are two predominant steps for verifying the UML models. Translate UML statecharts to the form expected by existing off-the-shelf model checker and use model checker to verify the correctness of the model. We feel that selection of off-the-shelf model checker hinges on capability of the checker to handle huge state space of a complex system. The existing model checkers are not developed with the intent of verifying the UML models and performance of the verification techniques can be improved if the model checking algorithms are modified. In other words, models of complex reactive systems can be efficiently verified. This fact has motivated us to develop efficient algorithms for verification of reactive and concurrent systems.

III. Methodology

A widely known approach for verifying the complex systems is, by modeling them in the input language of the off-the-shelf model checker and passing them on to model checker. The property expected is specified in temporal logic. Subsequently, the need of visual formalism to the models is realized and UML statecharts are used for modeling dynamic behavior of the system. The verification of such models is done by first representing the UML statecharts in Extended Hierarchical Automata (EHA) and then mapping it to input language of the model checker. This approach is well received and successful for less complex systems. As the complexity of the system grows, this technique of flattening (removal of abstraction) the original model during verification would lead to "state-explosion" [21] and hence aborts the verification process. The proposed method for verification of reactive systems does not use off-the-shelf model checker. Fig.1 depicts the architecture of the proposed method. The logics of the UML statechart diagram are captured using suitable data structure and then the state space graph is built. Unlike most of the model checkers, here the data structure preserves the abstraction and limits the state space to be explored for finding safety violations. Thus, memory required to hold the state space is reduced.

IV. Proposed verification techniques

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 and the corresponding guard condition becoming true. The property to be verified is expressed in temporal logic and represented by the symbol ϕ. The

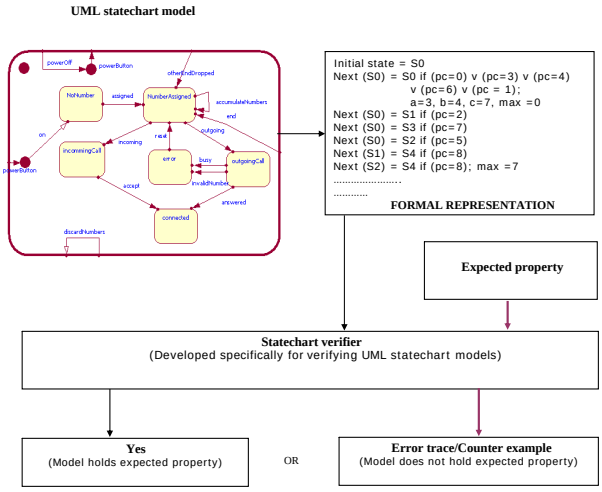


Figure 1: Verification framework

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 and varies from 1 to n
- where n is the number of objects.

S_i is a non empty finite set of states

E_i is a 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

E_t is a set of total events, $E_t = \{E_1 \cup E_2 \ldots E_n\}$

B. Basic verification approach

The basic verification algorithm is shown in Fig.2. The verification process is divided into two phases: preprocessing and state space search. In the preprocessing phase, the logics of the UML statechart diagrams are captured by converting them to formal intermediate representation. The behavior of each object in the reactive system is modeled using a statechart diagram. The UML editor saves formal representation of each of the statechart chart diagram in separate text files. The files are read independently and the behavioral aspects of the object are stored using graph data structure. The safety property being verified is written using temporal logic. It is interpreted and translated into AND/OR graph. The graph is traversed in such a way that, probable error states are obtained from the graph.

In the state space search phase, the tuple $\{S_i, E_i, T_i, I_i\}$ is extracted from the formal representation of behavior of individual objects. Once set E_i for all objects are computed, the union of all these sets represented by symbol E_t is computed. Now, the state space is constructed by combining (cartesian product) the state transitions of all objects upon occurrence of each event in E_t. To begin with, all objects are assumed to be in their respective initial states. During the construction of the state space, we compare probable error states with the states that are generated, if a match is found (observed an invalid behavior), further exploration of the state space is terminated. We then generate error trace, a path from the initial state to the error state. If no invalid behavior is observed, we continue the exploration of the state space until all possible states are visited. The complete exploration without error being detected implies model behavior is satisfactory and as per expectation. The algorithm does explicit checking, when the model is flaw less and no memory is saved. This algorithm can be further improved by finding the set of relevant events and observing the behavior of the system only upon occurrences of these relevant events. This event based technique is explained in the next section.

```
1:  Read ¬Ø (negative behavior or bad state) from the user;
2:  for each object i of the system (model)
3:  {
4:          Get Si set of reachable states;
5:          Get Ei set of all events;
6:          Get Ti set of all transitions;
7:          Get Ii set of initial states;
8:  }
9:          Compute Et;
    // Build the state space (synchronous product of all objects)//
10: Let found = false;
11: Start with state s (all objects are in their initial states);
12:  for (each event e Є Et  enabled in s & s not empty)
13: {
14:          s* = set of all successor states of s after ei;
15:          While (s* not empty)
16:          {
17:                  If (state sj є s*, is not in state space)
18:                  {
19:                          add sj to state space;
20:                          push sj on to stack;
21:                           If (state sj is same as ¬Ø)
22:                          {
23:                                  Set found flag to true;
24:                                  Break;
25:                          }
26:                          Mark the state sj as visited;
27:                  }
28:                  sj = nextstate (sj);
29:          }
30:          If (found) Break;
31:          s= pop ();
32:  }
33: If (found)
34:          Display "No negative behavior seen in the model";
35: Else
36: {
37:          Display "Negative behavior found";
38:          Display Error Trace / Counterexample;
39: }
```

Figure 2: Basic algorithm

C. Event based verification approach

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 verification approach described in this section is based on this idea. The algorithm for the approach is shown in Fig.3.

This modified version of the previous algorithm finds set of relevant events from the UML statechart model of

```
1:   Read ¬Ø (negative behavior or bad state) from the user;
2:    for each object i of the system (model)
3:   {
4:            Get S_i set of reachable states;
5:            Get Er_i set of all relevant events;
6:            Get T_i set of all transitions;
7:            Get I_i set of initial states;
8:   }
9:   for (i=1 to No. of objects)
10:  Compute Er_t= (Er_t U Get_relevant_events ( O_i ) );
     // Build the state space (synchronous product of all objects)//
11:  Start with state s (all objects are in their initial states);
12:   for (each relevant_event e€ Er_t enabled in s & s not empty)
13:   {
14:           s* = set of all successor states of s after e_i;
15:           While (s* not empty)
16:           {
17:                   If (state s_j e s*, is not in state space)
18:                   {
19:                           add s_j to state space;
20:                           push s_j on to stack;
21:                           If (state s_j is same as ¬Ø)
22:                           {
23:                                   Set found -flag to true;
24:                                   Break;
25:                           }
26:                   Mark the state s_j as visited;
27:                   }
28:           s_j = nextstate (s_j);
29:           }
30:  If (found) Break;
31:  s= pop ();
32:  }
33:  If (found)
34:           Display "No negative behavior seen in the model";
35:   Else
36:   {
37:           Display "Negative behavior found";
38:           Display Error Trace / Counterexample;
39:  }
```

Figure 3: Event based algorithm

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 looked for in the model and using the following rules:

R1: An event is relevant if

R 1.1: there is a transition associated and has current sate as part of error state ($\rightharpoonup \phi$).

R 1.2: there is a transition associated and has next state as part of error state ($\rightharpoonup \phi$).

R2: A set of events are relevant if

R 2.1: there is a sequence of transitions associated and takes the object from the initial state to a state, which is part of error state ($\rightharpoonup \phi$). 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 error state.

After the set of relevant events is computed, UML statechart of each object is translated to from of a tuple $\{S_i, Er_i, T_i, I_i\}$, where Er_i is a set of relevant events of an object O_i and Er_t is set of total relevant events, i.e, $Er_t = \{Er_1 \cup Er_2 \cup Er_3 \; ..Er_n\}$. The state space of the system is constructed considering only the events in 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. This approach is very much suitable for verification of safety property of a complex system, having considerable number of non-relevant events.

In the next section, we illustrate verification procedure by applying the above described algorithms to a benchmark case study, the "Generalized Railroad Crossing"(GRC) problem introduced by Heitmeyser et al [22].

V. A CASE STUDY

A. Generalized Railroad Crossing(GRC)

We have validated proposed algorithms by applying them to verify UML statechart model for the "Generalized Railroad Crossing"(GRC) system. The GRC 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) through A on tracks T1 and 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 directions on the same track, at any point of time. There are sensors (S1, S2, S3, S4 & S5) positioned as shown in the Fig.4. The sensors indicate when a train arrives at region A, leaves region A, enters RC and exits RC. The sensor S5 indicate, whether gate is closed or open. The "occupancy interval" is defined as the maximal time interval during which one or more train(s) 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 during all non occupancy interval (Utility)
3. The gate is open as much as possible (Live ness)

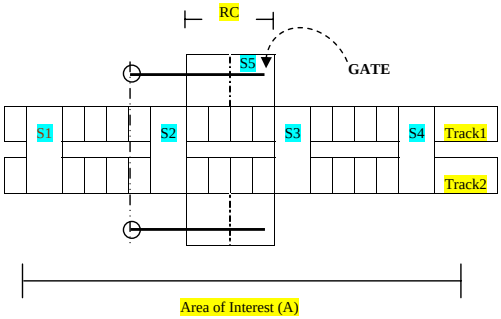


Figure 4: Railroad crossing

The dynamics of the GRC system is described by UML statecharts for the objects Gate and Track. The safety property looked for in the GRC model "When the train is at RC on Track1 or Track2 the Gate should remain closed" is expressed in temporal logic as follows:

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

In our approach, this positive 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 (error state) and counter example is generated. The above stated assertion can be written as follows in the negative form.

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

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

B. UML statechart model of GRC

The UML statechart model for the GRC system is presented in the Fig. 5. The gate and track are the major objects of the GRC system. The UML statechart for Gate in Fig. 5(a) shows four simple states viz., Open, Closing, Closed and Opening. Initially, the Gate is assumed to be Open. The Gate reacts to external signals by opening and closing. The UML statechart for Track in Fig.5(b) shows concurrent composite state consisting of two orthogonal regions for each track (Track1 & Track2), which are again having sequential states (OR state). 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 shown as label on the arrows (see Fig. 5) is triggered. All the events considered are listed in the table I.

TABLE I.: Events associated with GRC model

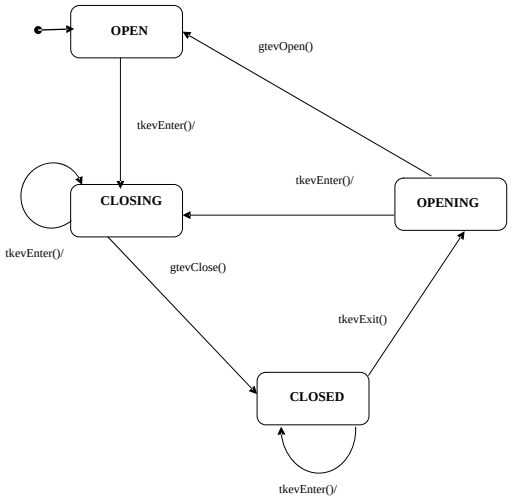
Event	Code	Description
tkevarrive	1	indicates train's arrival
tkeventer	2	indicates that, the train enters RC
tkevexit	3	indicates that, train exits RC
tkevleave	4	indicates that, train leaves
gtevclose	5	indicates that, gate is closed
gtevopen	6	indicates that, gate is opened

C. State space construction

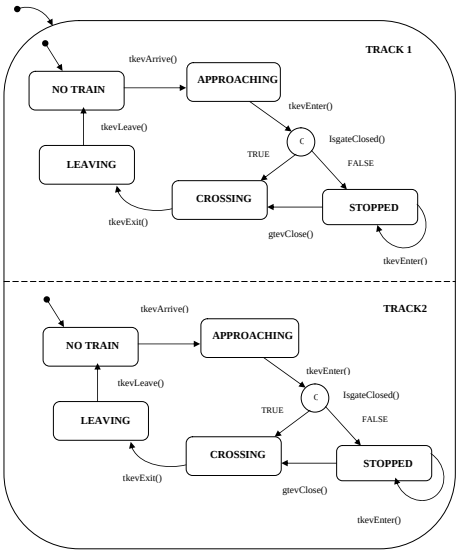
The state space is constructed from the description of the system in UML statechart model. As explained in sub section (IV-B), the dynamic behavior of all objects are combined to generate state space graph. The notion of "Universe" (U) is useful in describing the construction of state space. It is the set of all possible combinations of local states of the objects of a system. The UML statechart model of the GRC system (see Fig. 5) 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 our UML model the state space of the GRC system contains 46 states. The table II shows all possible states.

D. Basic algorithm applied to GRC

The basic algorithm checks the invalid behavior of the system during the construction of the state space. The construction process is terminated immediately when the negative behavior is observed. We have applied the basic verification algorithm to the generalized railroad crossing



(a) Statechart for the object GATE



(b) Statechart for the object TRACK

Figure 5: UML state chart model for GRC

model and observed that the original UML statechart model had an error state.

The Fig.6 shows the state space constructed. The state space is searched for the violation of the safety property "The gate is closed during all occupancy intervals" (i.e, occurrence of any of the state in the set $\{ S_7, S_9, S_{10}, S_{14}, S_{20}, S_{23}, S_{34}, S_{38}, S_{40}, S_{41}, S_{42}, S_{45} \}$ upon an event listed in the table I). The initial state S_1 is a state representing the initial states of Gate, Track1 and Track2 (i.e, Open, No train, No train). The successive states (S_2, S_5, S_6) upon occurrence of the event "tkevarrive" (see table I) are computed. These states are checked for safety violation. If violation is found further

TABLE II.: State space

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

exploration is terminated. Otherwise, a state is randomly selected for further exploration (for example state S_2). This process is continued until we detect a safety violation or all possible states are explored. In the case of GRC, exploration is terminated on reaching the state S_{45}, which is an error state. The state space graph constructed in the afore mentioned way is used to generate counter example or error trace shown in the Fig.7. We would explore all 46 states, if the model does not violate stated safety property.

E. Event based algorithm applied to GRC

In this approach, we compute the set of relevant events from the UML statechart model by applying the rules stated in the section IV-C. The relevant event sets obtained are Er_{Gate} = {gtevopen, gtevclose}, Er_{Track} = {tkevarrive, tkeventer, tkevexit} for objects Gate and Track respectively. The total set of relevant events Er_t = {gtevopen, gtevclose, tkevarrive, tkeventer, tkevexit}. The

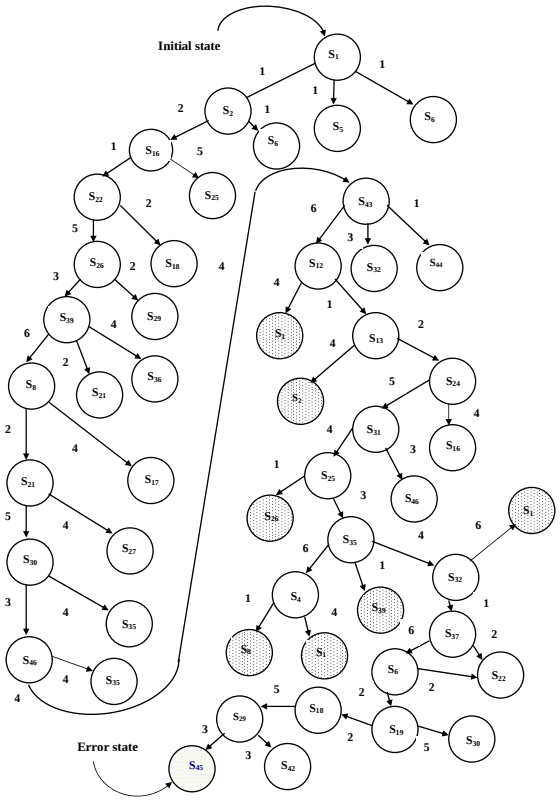


Figure 6: State space exploration

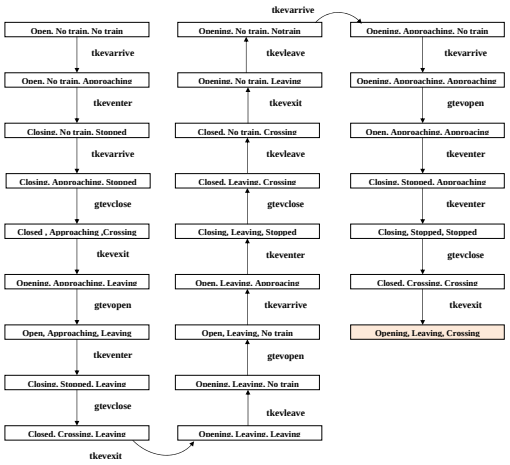


Figure 7: Error trace

"tkevleave"(see table I) is considered as the non relevant event and ignored during the construction of the state space. The Fig.8 shows the exploration of the state space by considering only the events in the set Er_t.

The exploration starts from the initial state S_1 and continues till a state is reached, which does not responds to any of the relevant events. Then we backtrack 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.8, state exploration starts with initial state S_1. The set of successive states (S_2, S_5, S_6) 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 backtrack till the state S_{29} is reached, which leads to state S_{42} 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 trains is at the crossing, the gate starts to open). Once the state exploration is terminated, the counter example or the error trace shown in Fig.9 is generated.

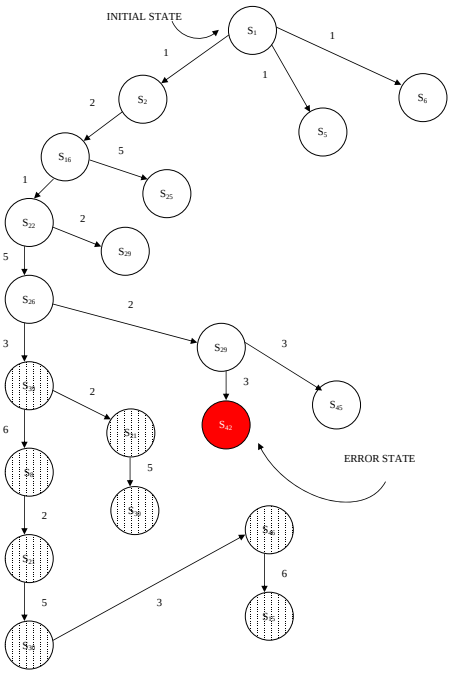


Figure 8: State space exploration

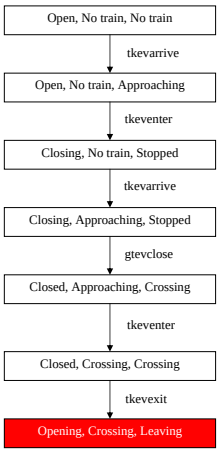


Figure 9: Error trace

VI. Results and Discussion

A. Refinement of GRC model

The error trace in Fig.9 depicts that, the Gate is allowed to open, when one of the trains crosses the RC and 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 the Fig.10. 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 trains are at crossing, when the Gate begins to open. Thus the model's correctness is ensured.

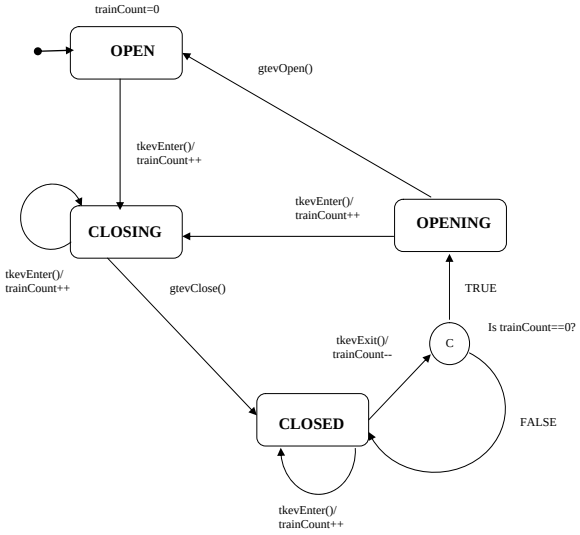


Figure 10: Corrected UML statechart for GATE

B. Performance of basic and event based algorithms

The basic and event based verification algorithms are evaluated based on the ability to reduce the state space during the state space exploration. The results are shown in table III. To detect safety property violation in the UML statechart model of the GRC system with state space of 46, the event based algorithm explores 41% of the total state space. The counter example generated is of length 6. The basic algorithm explores 87% of the total state space and the counter example generated is of legnth 24.

TABLE III.: Performance

Algorithm	**Complete state space**	**States explored**	**Error path length**	**State space reduced**
Basic	46	40	24	13%
Event based	46	19	06	59%

VII. Conclusions

In this article, we have described Basic & Event based algorithms devised for the verification of UML statechart models. The correctness of the verification techniques has been illustrated taking "Generalized Railroad Crossing(GRC) as a case study. The basic algorithm checks the

safety violation during the construction (on-the-fly) of the state space. This leads to the reduction in the state space (13% for GRC example). There will be no reduction in the state space if the verification is done on a flawless model. This algorithm will not generate the error trace of shortest length (24 for GRC).

The event based approach is modified basic approach, which 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). The event based approach gives 4.5 times better reduction in state space for GRC example as compared to the basic approach. We have verified the UML statechart model of the GRC system for the compliance of the safety "The gate is closed during all occupancy intervals" using the proposed techniques and found a flaw in the initial model and 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.

REFERENCES

[1] IBM Rational Rose Real Time (RoseRT) instruction manual,http://www.ibm.com/developerworks/rational/library /797.html, visited on 01/12/2007

[2] Edmund M. Clarke,Jr., Orna Grumberg and Doron A. Peled , Model Checking, The MIT press, 1999

[3] Edmund M Clarke, Ansgar Fehnker, et.al. "Abstraction and Counterexample refinement in model checking of Hybrid Systems", Vol.14, No 4, International journal of foundations of computer science, (2003), 583-604

[4] Gerard J. Holzmann, "The Model Checker Spin, IEEE Trans. on Software Engineering", Vol. 23, No. 5, (1997), 279-295

[5] Kenneth L. Mc. Millan, "Symbolic Model Checking: An approach to the state explosion problem", Ph.D thesis submitted to Carnegie Mellon University (CMU), 1992

[6] The SLAM project, http://research.microsoft.com/slam/ visited on 13/10/2007

[7] I. Beer, S. Ben-David, C. Eisner and Landvar, " RuleBase-an industry-oriented formal verification tool", Proceedings of 33rd Design Automation Conference (DAC), Association for Computing Machinery Inc.,(1996), 655-660.

[8] D. Harel, "Statecharts: A Visual Formalism for Complex Systems, Science Computer Programming",vol.8, no. 3, pp 231-274, 1987.

[9] C.M. Prashanth, Dr. K.C. Shet, Janees Elamkulam, "Verification Framework for Detecting Safety Violations in UML statecharts", Second Asia International conference on Modeling and Simulation (AMS 2008), Kuala Lumpur, Malaysia, 13-15 May 2008,pp 849-854. publisher: IEEE computer society

[10] William Chan, Richard J. Anderson, Paul Beame, Steve Burns et.al. "Model Checking Large Software Specifications", IEEE Transactions on Software Engineering, Volume 24, Issue 7, pp 498-520, July 1998

[11] G.J. Holzmann et.al., "Implementing statecharts in PROMELA/SPIN", proc. workshop on industrial strength formal specification techniques WIFT'98, USA, IEEE computer society, 1998.

[12] Diego Latella, Istvan Majzik and Mieke Massink, "Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model checker", Formal Aspects of Computing, volume 11(6), 1999, pages 637-664.

[13] Stefania Gnesi, et. al., "Model Checking UML statechart diagrams using JACK", The 4th IEEE international symposium on high assurance systems engineering, pages 46-55, 1999

[14] Janees Elamkulam, Ziv Glazberg, et.al., " Detecting Design Flaws in UML State Charts for Embedded Software", Haifa Verification Conference 2006: pp 109-121

[15] Johan Lilus, Ivan pores paltor, " vUML: A tool for verifying UML models", Proceedings of the 14th IEEE international conference on automated software engineering, 225-228, 1999

[16] Stefan leue, Gerard Holzmann, "V-Promela: A Visual object-oriented language for SPIN", Proc. 2nd IEEE international symposium on object-oriented real time distributed computing, 1999, pages 14-23.

[17] Adam Darvas et.al., "Verification of UML statechart models of embedded systems" 5th IEEE design & diagnostics of electronic circuits and systems workshop, April 2002, pp 70 -77.

[18] M. Beato et. al., "UML Automatic Verification Tool (TABU)", 12th ACM SIGSOFT symposium on the foundations of software engineering, 2004

[19] Randal E. Bryant, "Graph based algorithms for Boolean function manipulation", IEEE Trans. on Computers Vol. C-35 No. 8, Aug 1986, Page(s):677 - 691

[20] Computational Tree Logic(CTL), Department of Computer Science, University of Copenhagen - visited in October 2007. http://www-i2.informatik.rwth-aachen.de/Teaching/Course/MC/2005/mc_lec22.pdf

[21] Valmari,A. "The State explosion Problem", Lectures on Petri Nets I: Basic Models, LNCS 1491, Springer-Verlag pp 429-528, 1998.

[22] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw, "Comparing different approaches for Specifying and Verifying Real-Time Systems". In Proceedings of 10th IEEE workshop on Real-Time Operating Systems and Software, (1993), 122-129

C.M. Prashanth is currently a Ph.D. student in the department of computer engineering, National Institute of Technology Karnataka, surathkal, INDIA. His research interests include Software Engineering, Computer Architecture and Operating systems. He is a life member of Indian Society of Technical Education. He has published papers in refereed journals and international conference proceedings.

Dr. K. Chandrashekhar Shet is a Professor in the department of Computer Engineering, National Institute of Technology Karnataka, INDIA. He holds a Ph.D. from the Indian Institute of Technology, Bombay, INDIA. He is a member of Computer Society of India, and ISTE. He is a Fellow of Institution of Engineers (INDIA). His research interests include software testing, Security Solution for Web Services, Wireless Networks, Ad hoc Networks. He has published more than 200 papers in refereed journals and conference proceedings.