A New approach to Detect Safety Violations in UML Statechart Models

Prashanth C.M.[†] **Dr. K. Chandrashekar Shet**[††],

Dept. of Computer Engineering,
National Institute of Technology Karnataka , Surathkal, INDIA

Summary
The model based development is a widely accepted phenomenon to build reliable software. This has prompted development of tools capable of generating code from the model. Such rapid software development tools are handy in development of embedded systems. The code generated using tools can be deployed directly on to target hard ware, provided the model correctness is ensured. In this paper, we present an efficient procedure to verify UML (Unified Modeling Language) statechart models of reactive and concurrent systems. The algorithm checks for safety property violation during the construction (on-the-fly) of the state space graph and generates counter example if any violation is found. The exploration of the state space is terminated, as soon as safety violation is found and hence search space is reduced. We prove the correctness of the approach by taking a benchmark case study of Generalized Railroad Crossing (GRC) system. The dynamic behavior of the gate & track, two concurrent objects of the GRC system are modeled using UML statecharts and the safety property "when train is at the crossing, the gate always remain closed" is verified. We could detect property violation in the initial UML statechart model of GRC and later it is corrected with the help of the counter example generated by the algorithm. The case study results show that the verification algorithm yields 13% reduction in the state space for the GRC example.

Key words:
UML Statecharts, Software verification, Reactive Systems

1. Introduction

1.1 Preamble

The development of reliable software has been the major goal for the advent of software engineering discipline. 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. 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 in high level languages like C, C++ or JAVA from the model (IBM's Rational Rose RT [1] is one such tool used for the development of embedded real time systems).

As deployable binaries are generated from the model, ensuring model's correctness becomes highly essential. The commonly used model verification technique is model checking. Model checking [2] is a pragmatic technique that, given a finite-state model of a system and a logical property (expected system property), systematically checks whether model holds the property or not. If the model does not hold the expected property, an error trace (also called as counter example) is generated. The original model can be refined by leveraging information generated 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], BLAST (Berkeley Lazy Abstraction software verification Tool) [7] and Rule Base [8] are in existence.

The major drawback of using afore mentioned model checking tools for verification is that, they expect system to be modeled using their proprietary input language. The input languages of most of these tools are text based and lacks advantages of visual representation. Numerous researchers have tried to address this issue. We have surveyed the earlier works (see our published papers ([9],[10]) and found that, though they suggest modeling the dynamic behavior of the system using UML (Unified Modeling Language) statechart diagrams (provides visual representation to the models), subsequently these statechart diagrams are translated to the input language of the model checker before verification. The translation process removes the abstraction of the models and exponentially increases the state space of the complex systems. This could lead to state explosion [11].

We in this paper, present verification algorithm which avoids the usage of off-the-shelf model checker and

Manuscript received July 5, 2008
Manuscript revised July 20, 2008

translation of UML statechart models to input language of the model checker. The algorithm presented is memory efficient and successfully handles the complex reactive systems.

In the section 2, we present algorithm devised to verify safety properties of reactive systems. In section 3, we describe generalized rail road crossing problem, UML statechart model for the GRC and also discus about the verification of safety property of the GRC model using the proposed technique. The results and performance of the proposed verification technique is discussed in the section 4. We draw conclusions in the section 5.

1.2 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 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" and hence aborts the verification process. The proposed algorithm for verification of reactive systems does not use off-the-shelf model checker. The 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. Thus, memory required is reduced. This methodology is explained in detail in our earlier papers [9, 10 and 12].

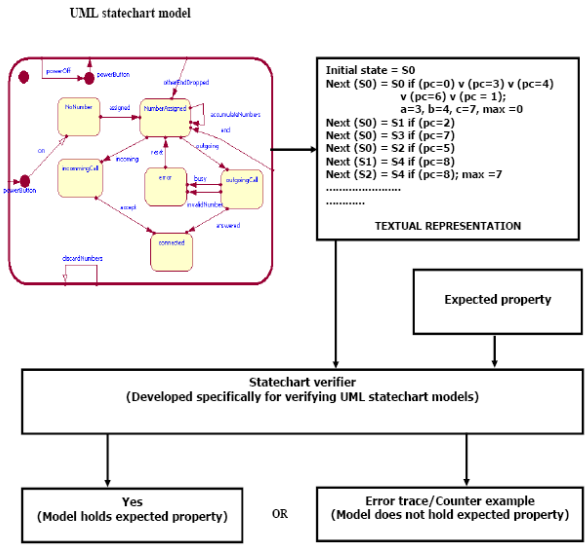


Figure 1: Proposed verification method

2. Proposed verification technique

2.1 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 events & the corresponding guard condition becoming true. 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, i varies from 1 to n, where n is number of objects
- S_i is a non empty finite set of states of an object
- E_i represent set of events associated with an object
- $T_i \subseteq S_i \times S_i$ is a set of total transitions
- $I_i \subseteq S_i$ is a set of initial states
- Let E_t be set of total events,

  i.e $E_t = \{E_1 \cup E_2 .... \cup E_n\}$

The property to be verified is expressed in a temporal logic.

2.2 Verification approach

In our approach, the state space of the system is built by combining (Cartesian product) the state transitions of all objects upon occurrence of each event $e_i \in E_i$. 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. This approach limits the search space and memory usage thereby. The flowchart and algorithm are shown in Fig. 2 and Fig. 3 respectively. The algorithm does explicit checking, when model is flaw less and no memory is saved. This algorithm can be further improved by finding the set relevant events and observing the behavior of the system only upon occurrences of these relevant events. In the next section, we illustrate verification procedure by applying the described algorithm to a benchmark case study, the "Generalized Railroad Crossing" (GRC) problem introduced by Heitmeyser et al [13].

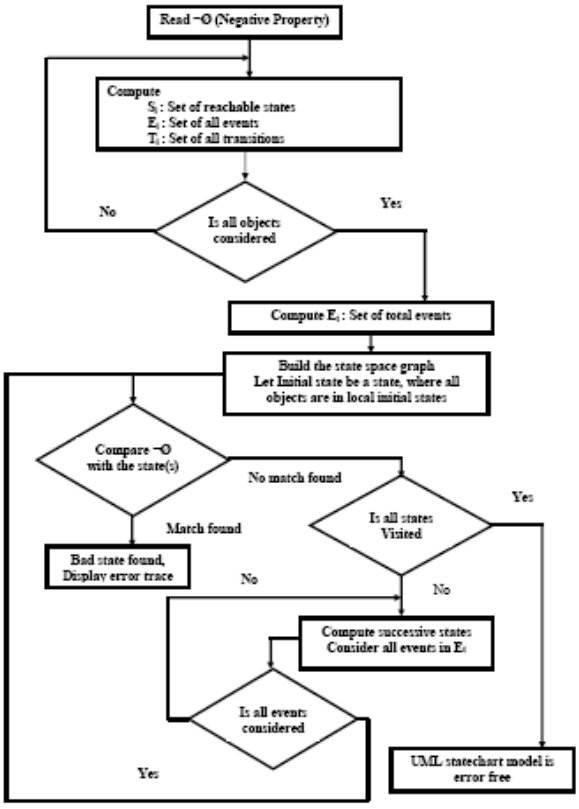


Figure 2: Approach

```
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 ∈ 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 ϵ 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 3: Verification algorithm

3. A case study

3.1 The Generalized Railroad Crossing (GRC)

In this section we describe the process of verifying 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 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.4. The sensors indicate when the train 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, maximal time interval during which one or more trains are in railroad crossing (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 as much as possible (Live ness)

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:

$$(T1.Crossing \ V \ T2.Crossing) \Longrightarrow G.Closed$$

(V represents logical OR)

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 and counter example is generated (path from the initial state to error state). The above stated assertion can be written as follows in the negative form.

$$(T1.Crossing \ V \ T2.Crossing) \Longrightarrow \neg (G.Closed)$$

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

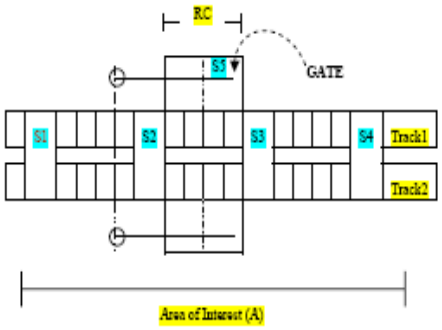


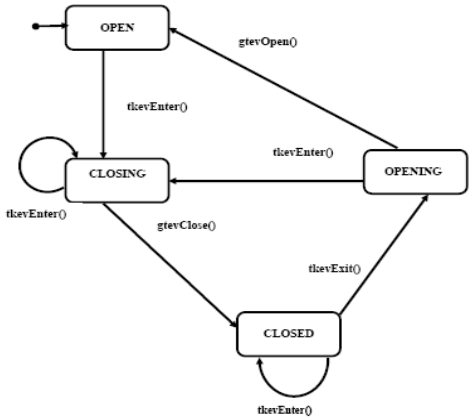
Figure 4: Railroad crossing

3.2 UML statechart model of GRC

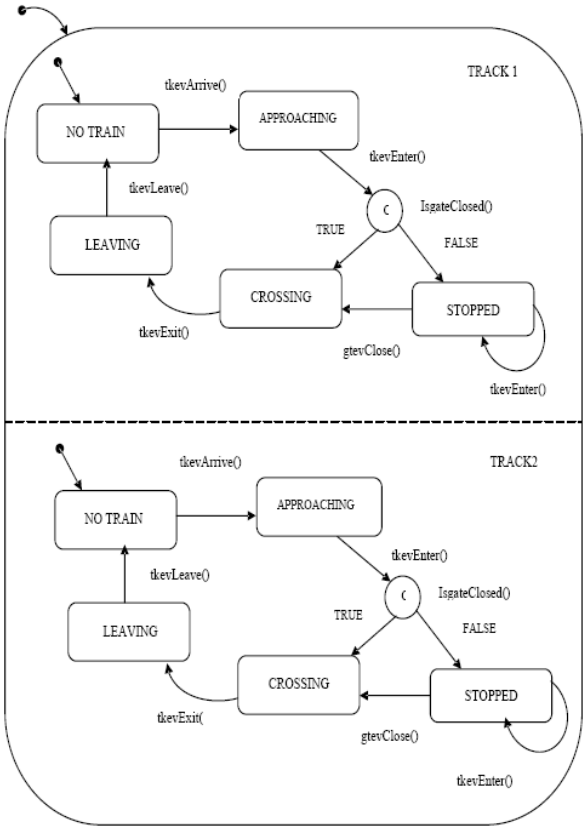
The UML statechart model for the GRC system is presented in the Fig. 5. The gate and track are the major objects of the GRC system. The UML statechart for Gate in Fig. 5(a) shows an initial state and four simple states viz., Open, Closing, Closed and Opening. The gate reacts to external signals by opening & closing of gate. The UML statechart for Track in Fig. 5 (b) shows concurrent composite state consisting of two orthogonal regions for each track (Track1 & Track2), which are in turn 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 given as label on the arrows (see Fig. 5) is triggered. All the events responsible for state transitions of objects are listed in the table 1.

Table 1: Events associated with GRC

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



(a) UML statechart for the object GATE



(b) UML statechart for the object TRACK

Figure 5: UML statechart model of GRC

3.3 State space construction

The state space is constructed from the description of the system in UML statechart model. As explained in section 2, the dynamic behaviors 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 2 shows all possible states.

Table 2: All possible states

Sl.No.	Gate status	Track1 status	Track2 status
1.	Open	Notrain	Notrain
2.	Open	Notrain	Approaching
3.	Open	Notrain	Crossing
4.	Open	Notrain	Leaving
5.	Open	Approaching	Notrain
6.	Open	Approaching	Approaching
7.	Open	Approaching	Crossing
8.	Open	Approaching	Leaving
9.	Open	Crossing	Notrain
10.	Open	Crossing	Approaching
11.	Open	Crossing	Leaving
12.	Open	Leaving	Notrain
13.	Open	Leaving	Approaching
14.	Open	Leaving	Crossing
15.	Open	Leaving	Leaving
16.	Closing	Notrain	Stopped
17.	Closing	Stopped	Notrain
18.	Closing	Stopped	Stopped
19.	Closing	Stopped	Approaching
20.	Closing	Stopped	Crossing
21.	Closing	Stopped	Leaving
22.	Closing	Approaching	Stopped
23.	Closing	Crossing	Stopped
24.	Closing	Leaving	Stopped
25.	Closed	Notrain	Crossing
26.	Closed	Approaching	Crossing
27.	Closed	Crossing	Notrain
28.	Closed	Crossing	Approaching
29.	Closed	Crossing	Crossing
30.	Closed	Crossing	Leaving
31.	Closed	Leaving	Crossing
32.	Opening	Notrain	Notrain
33.	Opening	Notrain	Approaching
34.	Opening	Notrain	Crossing
35.	Opening	Notrain	Leaving
36.	Opening	Approaching	Notrain
37.	Opening	Approaching	Approaching
38.	Opening	Approaching	Crossing
39.	Opening	Approaching	Leaving
40.	Opening	Crossing	Notrain
41.	Opening	Crossing	Approaching
42.	Opening	Crossing	Leaving
43.	Opening	Leaving	Notrain
44.	Opening	Leaving	Approaching
45.	Opening	Leaving	Crossing
46.	Opening	Leaving	Leaving

3.4 The algorithm applied to GRC

The 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 verification algorithm to the generalized railroad crossing model and observed that the original UML statechart model had bad state or error state. The Fig.7 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", occurrence of any state in the set $\{S_7, S_9, S_{10}, S_{14}, S_{20}, S_{23}, S_{34}, S_{40}, S_{41}, s_{42}, S_{45}\}$ is treated as safety violation.

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) on event “tkevarrive” (see table 1) are computed. These states are checked for safety violation. If violation is found further exploration is terminated. Otherwise, a state is randomly selected for further exploration (for example state S_2). This process is

continued till we see 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 afore mentioned way is used to generate counter example or error trace shown in the Fig. 8.

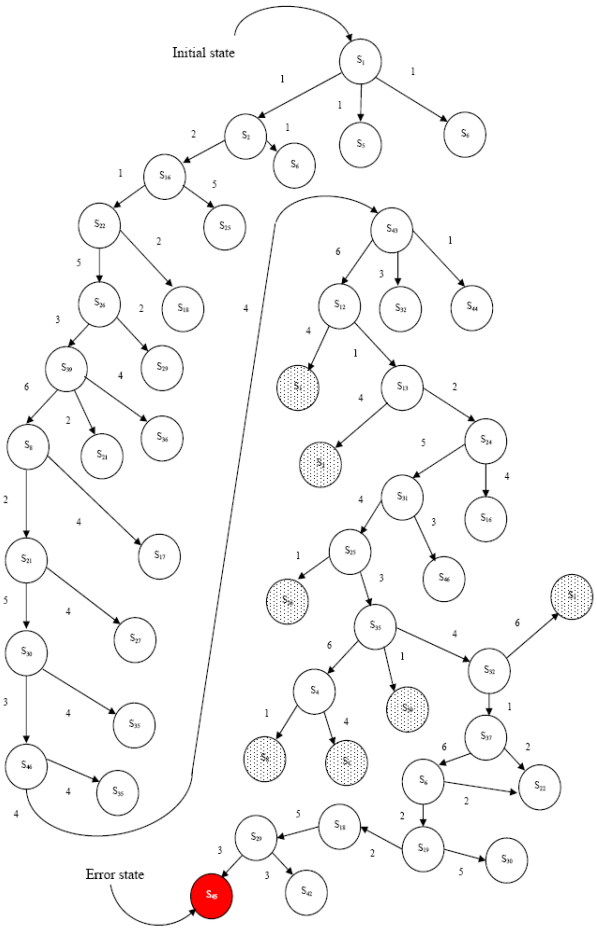


Figure 7: State space exploration

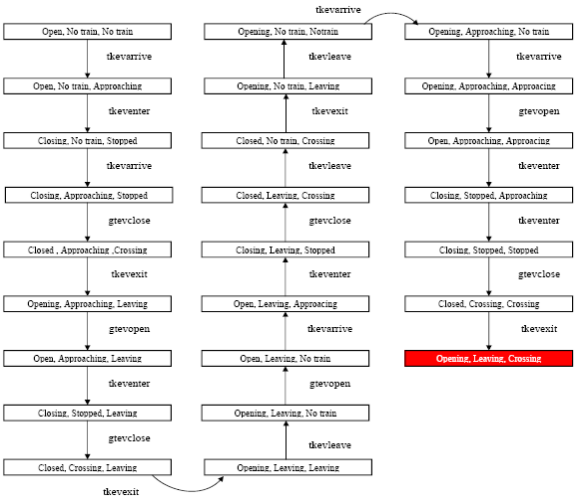


Figure 8: Error trace/counter example

4. Results and discussion

4.1 Correcting the UML statechart model of GRC

The error trace shown in Fig. 8 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. 9. 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. The value of this train count is checked by the Gate object before changing its state from closed to opening. If the train count is 0 then the Gate starts opening, other wise it remains closed. There by we ensure that no trains are at crossing, when the Gate begins to open. Thus the model correctness is ensured.

4.2 Performance of the algorithm

The verification algorithm is evaluated based on the ability to reduce the state space during the state exploration. The results obtained by applying the algorithm to GRC system is shown in table 3.

Table 3: Performance

Complete state space	States Explored	Error path length	State space reduction
46	40	24	13%

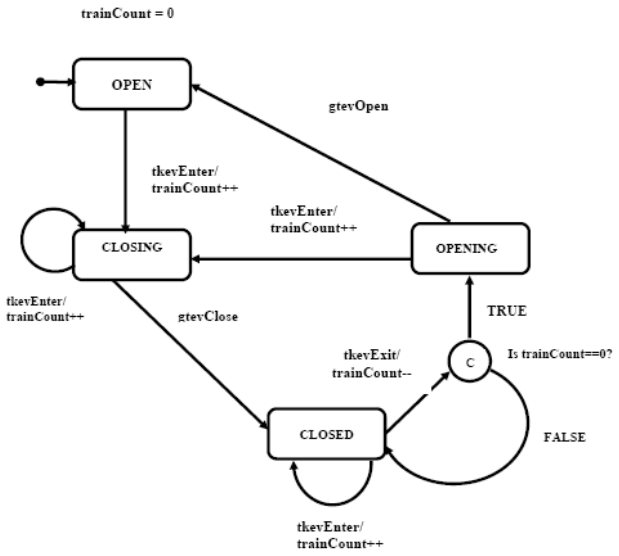


Figure 9: Corrected UML statechart for the object GATE

5. Conclusions

A majority of the existing approaches translate UML statechart model into text based modeling language which can then be 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 algorithm for the verification of safety property violations in UML statechart models of reactive systems. The correctness of the verification technique has been illustrated taking “Generalized Railroad Crossing (GRC)” as a case study. The 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).

We have verified the UML statechart model of the GRC system for compliance of the safety “The gate is closed during all occupancy intervals" using the above mentioned technique 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.

Acknowledgment

We would like to thank all those employees of IBM India private limited, Bangalore who have given fruitful suggestions and assistance in carrying out this work.

References

[1] IBM's Rational Rose Real Time (Rational Rose RT) tool 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 of microsoft laboratory, http://research.microsoft.com/slam/ visited on 13/10/2007

[7] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Grégoire Sutre, Software Verification with BLAST. 235-239, Electronic Editions (Springer LINK)

[8] 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.

[9] C.M. Prashanth, Dr. K.C. Shet, Janees Elamkulam, " A survey of model checking the UML statechart model of embedded systems", National Conference on Emerging Trends in Engineering \& Technology, Frontier (2007),India, 149-155

[10] C.M. Prashanth, Dr. K.C. Shet, Janees Elamkulam, “A Reality chek of model checking the UML statechart diagrams and research directions", International conference on Computers, Communication, Control systems and Instrumentation (3CI-2007), Bangalore, India, pp 16-22

[11] Valmari,A.: The State explosion Problem, Lectures on Petri Nets I: Basic Models, LNCS 1491, Springer-Verlag (1998) 429-52

[12] 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

[13] 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 an Assistant Professor in the department of Computer Science & Engineering, Adichunchanagiri Institute of Technology, India. He has received the B.E. degree in Electronics & Communication from Adichunchanagiri Institute of Technology, India in 1996 and M.E. degree in Computer Science & Engineering from Vellore College of Engineering, India in 2002. He is currently pursuing Ph.D at National Institute of Technology Karnataka, India. His research interests include Software Engineering, Computer Architecture and Operating system. He is life member of Indian Society of Technical Education. He has published papers in refereed international conference proceedings and Journals.

Dr. K. C. SHET is a Professor in the dept., of Computer Engineering, National Institute of Technology Karnataka, India. He has more than 36 years of experience in teaching and research. He holds a Ph. D. from IIT Bombay, India. He is a member of Computer Society of India, and Indian Society of Technical Education. He is a Fellow of Institution of Engineers (INDIA). His research interests include software testing, Security Solution for Web Services, Cyber Laws, Anti spam solutions, Wireless Networks, Mobile Computing, Ad hoc Networks. He has published more than 200 papers in refereed conference proceedings and journals.