International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:12, 2009

Verification of Protocol Design using UML - SMV

Prashanth C.M. and K. Chandrashekar Shet

Abstract—In recent past, the Unified Modeling Language (UML) has become the de facto industry standard for object-oriented modeling of the software systems. The syntax and semantics rich UML has encouraged industry to develop several supporting tools including those capable of generating deployable product (code) from the UML models. As a consequence, ensuring the correctness of the model/design has become challenging and extremely important task.

In this paper, we present an approach for automatic verification of protocol model/design. As a case study, Session Initiation Protocol (SIP) design is verified for the property, "the CALLER will not converse with the CALLEE before the connection is established between them ". The SIP is modeled using UML statechart diagrams and the desired properties are expressed in temporal logic. Our prototype verifier "UML-SMV" is used to carry out the verification. We subjected an erroneous SIP model to the UML-SMV, the verifier could successfully detect the error (in 76.26ms) and generate the error trace.

Keywords—Unified Modeling Language, Statechart, Verification, Protocol Design, Model Checking.

I. Introduction

Designing a new protocol, which is reliable and meets all the specifications, is a challenging task. There are many instances of protocol failing (when actually deployed) even after thorough testing. To enhance the correctness of a protocol design, formal verification techniques are used. The formal verification is a complementary approach to testing for detecting subtle design errors. In formal verification, the behavior of the protocol is described using a program like notation, collection of finite state machines or temporal logic formulae and then this description is compared with the specification of the desired behavior. If the verifier detects any flaw in the behavioral description, then it generates a counter example or an error trace that illustrates how the problem has occurred [1].

In this paper, we present a pragmatic semiformal approach for verifying the protocol design. We have considered UML statechart diagrams [2] for describing the behavior of the protocol. The desired behavioral property (safety property) is specified using temporal logic formula [3]. The problem associated with the model verification is "State Explosion" [4]. The state space search technique presented in this paper restricts the exploration of number of states, thereby saves the time and the memory required. We have demonstrated the correctness of our approach by verifying the UML statechart model of the Session Initiation Protocol (SIP). We inducted an error to the SIP model and subjected it for verification. Our prototype verifier "UML-SMV" (UML - State chart Model Verifier) detected this error and generated an error trace.

In the next section we present an overview of the Session Initiation Protocol. The UML statechart model of the SIP is described in the section III. The verification methodology is explained in the section IV and the algorithm for efficient state - space search is given in the section V. The prototype verifier "UML-SMV" is described in section VI. We have presented results obtained from the verification of SIP design in the section VII. We distinguish our work from the other protocol verification techniques in the section VIII. Finally, we have drawn conclusions based on the work carried out in the section IX.

II. Overview of SIP

The Session Initiation Protocol (SIP) is a signalling control protocol on the application layer of a network [5]. It is used for setting up and to terminate the multimedia communication session between the users. The SIP is designed (the latest version of the specification RFC 3261 is from Internet Engineering Task Force - IETF) to enable the functionalities of the network elements designated as Proxy Servers and User Agents. The Fig.1 depicts the SIP call establishment and termination process.

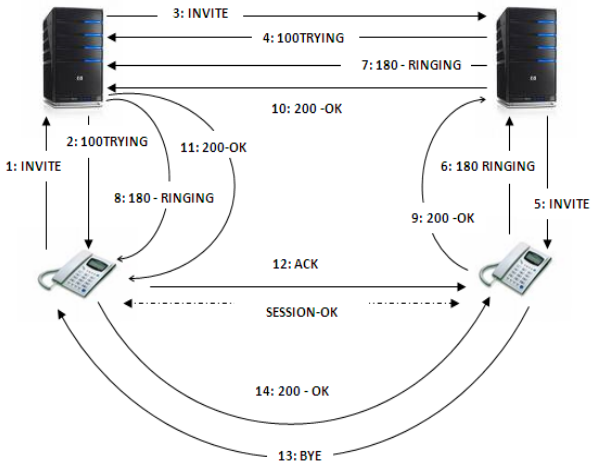


Fig. 1: SIP Call Establishment

Prashanth C.M. is a Research Scholar in the Department of Computer Engineering, National Institute of Technology Karnataka, Surathkal, INDIA, 575025. E-mail: prashanth_bcs@yahoo.co.in

K. Chandrashekar Shet is a Professor in the Department of Computer Engineering, National Institute of Technology Karnataka, Surathkal, INDIA, 575025. E-mail: kcshet@nitk.ac.in

III. UML Statechart Model of SIP

The SIP protocol has four major components "CALLER", "CALLEE", "CALLER Side Proxy" and "CALLEE Side Proxy". These are called objects. In our verification approach, the behavior of each of these objects is modeled using UML statechart diagram. The UML statechart representations of the above mentioned objects are shown in the Fig. 2. The desired property is expressed using temporal logic (symbolically represented as ϕ).

We have verified the property "CALLER will converse with the CALLEE, only after the call being established". This property is described in temporal logic formula as shown below.

$$\text{CR.Conversation} \Rightarrow \text{CL.CallEstablished} \wedge \text{CLP.CallEstablished} \wedge \text{CRP.CallEstablished}$$

IV. Verification Methodology

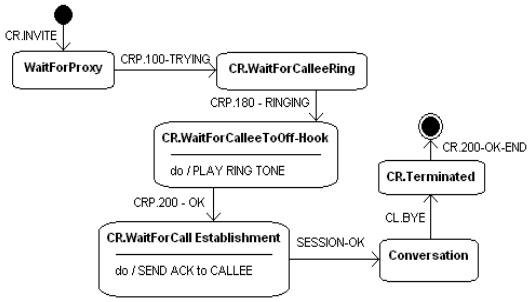
In this section, we introduce the framework proposed for detecting safety violations in UML statechart models. The basic verification process is divided into the following three phases.

A. Preprocessing Phase

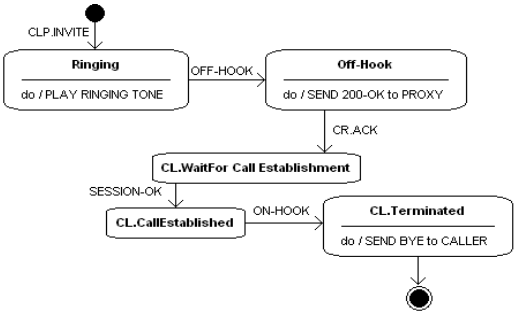
In this phase, the UML Statechart model of the system is first converted into XMI(XML Metadata Interchange) format [6]. The XMI representation of the statechart diagrams are then translated to Intermediate Representation (IR model). The IR model is read and the behavioral aspects of the object are stored using graph data structure. The safety property being verified is written using temporal logic. It is essential to interpret the temporal logic property expression. We translate property expression into AND/OR graph and traverse the graph in such a way that, error states are generated.

B. Verification Phase

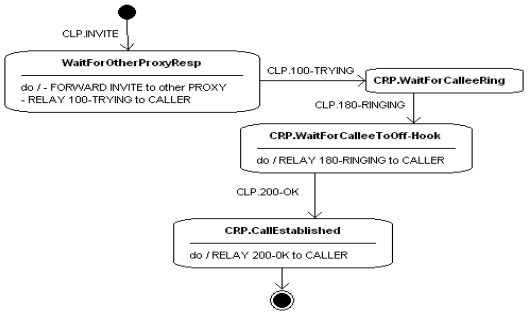
In this phase, the global state space tree is constructed by combining the state transitions of all objects upon occurrence of each event. To begin with, all objects are assumed to be in their respective initial states, during the construction of the state space (on-the-fly), error states are compared with the states that are generated, if the match is found, further exploration of the state space is terminated. We then generate an 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 till all possible states are visited. The complete exploration without error being detected implies model behavior is satisfactory. We have devised an efficient state-space search algorithm (refer section V), which finds set of relevant events associated with each object of the system. The relevant events are computed using the information given in the safety property being checked. The union of all these set of relevant events constitute set of total relevant events (Er_t). The number of events in Er_t will be less than or equal to set of all the events and hence number of states to be explored for detecting the error is reduced.



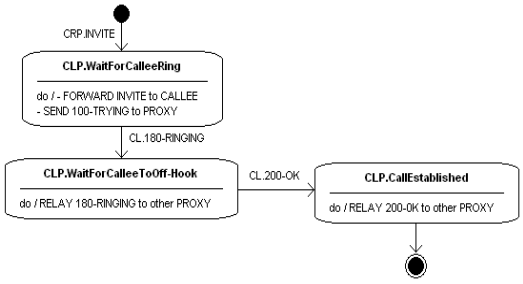
(a) Statechart for CALLER



(b) Statechart for CALLEE



(c) Statechart for CALLER Side Proxy



(d) Statechart for CALLEE Side Proxy

Fig. 2: UML Statechart Model for SIP

C. Analysis Phase

In this phase, the result obtained from the previous phase is examined. If the UML statechart model has any flaw, the previous phase generates an error trace or a counter example. The information provided in the error trace is used to correct the model and the model is again subjected to the verification. This process is repeated till model's correctness is ascertained. The Fig.3 shows all the analysis activities.

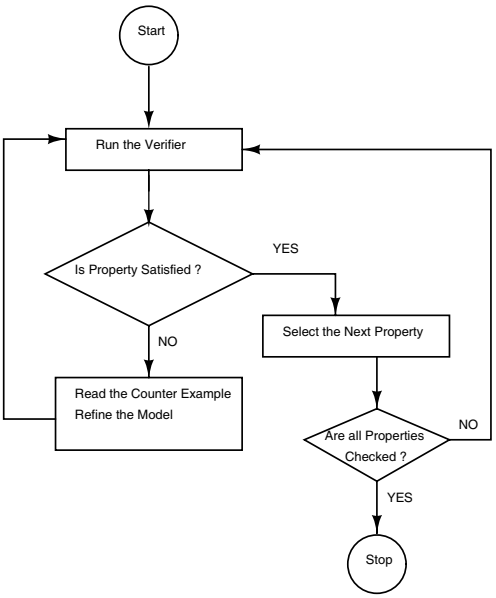


Fig. 3: Analysis Process

V. State space search Algorithm

The algorithm devised for efficient searching the state space of the protocol is shown in Fig.4. The state transition of an object completely depends on externally or internally generated event. Any technique which reduces the number of events to be considered for constructing state space tree, will ultimately reduce the search space. The algorithm devised is based on this idea. This algorithm finds set of relevant events from the UML statechart of each object of the protocol system. The rules listed below are used for computing the set of relevant events (Er_i) associated with an object.

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 ($\rightharpoonup \phi$).
- R 1.2: there is a transition associated with this event 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 with these events 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 the error state.

The union of all these set of relevant events constitutes the set Er_t (i.e $Er_t = \{Er_1 \cup Er_2 \; ..Er_n\}$, where Er_i represent a set of relevant events associated with object i). These events are used by the "Event Manager" for the state space exploration (refer to section VI for the details of "Event Manager").

```
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:   Let flag = false;
12:   Start with state s (all objects are in their initial states);
13:    for (each relevant_event e∈ Er_t enabled in s & s not empty)
14:    {
15:         s* = set of all successor states of s after e_i;
16:         While (s* not empty)
17:         {
18:               If (state s_j ϵ s*, is not in state space)
19:               {
20:                     add s_j to state space;
21:                     push s_j on to stack;
22:                     If (state s_j is same as ¬Ø)
23:                     {
24:                           Set found flag to true;
25:                           Break;
26:                     }
27:               Mark the state s_j as visited;
28:                }
29:          s_j = nextstate (s_j);
30:         }
31:   If (found) Break;
32:   s= pop ();
33:   }
34:   If (! found)
35:         Display "No negative behavior seen in the model";
36:    Else
37:    {
38:         Display "Negative behavior found";
39:         Display Error Trace / Counterexample;
40:   }
```

Fig. 4: Search Algorithm

VI. A Prototype Verifier UML-SMV

A prototype Unified Modeling Language -Statechart Model Verifier (UML-SMV) is developed in C++ language to validate the verification framework. The prototype developed is of size 3655 lines of code, has 13 major header files and a main routine. The UML-SMV system architecture shown in Fig.6. The distinguishable functional components of the UML-SMV are described in the following sections.

A. Preprocessing Components

1) UML Modeler: The BOUML [7], open source modeling tool conforming to UML 2.0 standards of Object Management Group (OMG) has been used for drawing the UML statechart

diagrams. This tool allows us to specify UML models and generate code in Python, C++ or Java languages. The BOUML runs on Linux/Unix/Solaris, Mac OS X and Windows. The tool supports generation of XMI 1.2 or XMI 2.1 representation of UML diagrams. The BOUML tool is integrated to UML-SMV.

2) Translator: The XMI representation of the UML diagrams is translated into Intermediate representation by a C++ Translator routine XMI2IR(). The Translator routine removes all unnecessary characters, tags and information from the XMI document but preserves the logics and hierarchical structure of the original model.

3) IR parser: The intermediate representation (IR) is the textual description of the UML statechart model and is the input to IR Parser (MODPARSE()) module. The IR parser transforms the behavioral description of each object to graph data structure.

4) Property Inference Engine(PIE): The architecture of the Property Inference Engine (PIE) is shown in Fig.**??**. The safety property expressed in temporal logic is input to the property inference engine. The inference engine will produce a AND/OR graph, which has to be traversed in a depth first manner to extract error states.

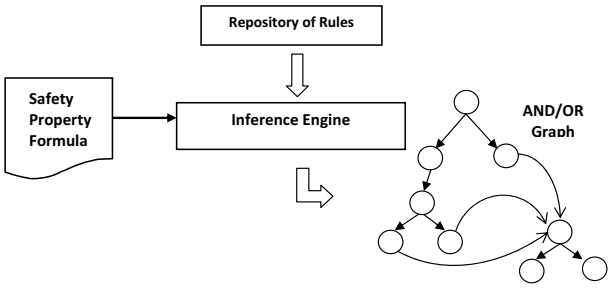


Fig. 5: Architecture of the Property Inference Engine

B. Verification Engine Components

1) Event Manager: The "Event Manger" component of the verification engine is responsible for determining the order of the events (The UML sequence diagram is used for modeling the order of occurrence of the events) and holding the events in the determined order in an "Event Queue" for dispatching. It is also responsible for selecting and de-queuing the event instances from the "Event Queue".

2) State Manager: The "State Manager" has the responsibility of managing the following activities:

- Given the current state of the system, it is the job of the "State Manager" to determine the next possible states on occurrence of each of the event in the "Event Queue".
- The "State Manager" selects a state (if the state is not already visited or added to the state space graph) from the set of next possible states of the system for further exploration.
- If the state selected is not in the state space graph, the "State Manager" adds it to the state space graph. Thus builds the state space tree.
- The "State Manager" also checks whether the selected state is an error state or not by comparing it with the states in the set of error states. If the selected state happens to be an error state, then it asks the "Search Engine" to terminate the search process.

3) Search Engine: The "Search Engine" prunes the state space for detecting safety property violation during the construction of the state space tree. The root node represents the state in which all objects are assumed to be in their local initial state. The "State Manager" determines all possible next states of the system on each of the event dispatched by the "Event Manager". All these states are pushed on to the stack and the first state in the stack is chosen speculatively for further exploration. Next, the "State Manager" checks whether the selected state is in the state space graph or not. If the state is already visited, the state is removed from the stack and state which is next in the stack is selected.

4) Back Tracker: If the "Search Engine" finds a property violation (error), the control is handed over to "Back Tracker". The "Back Tracker" then traces the path back to the initial state from the error state. This then displays the error trail (the counter example).

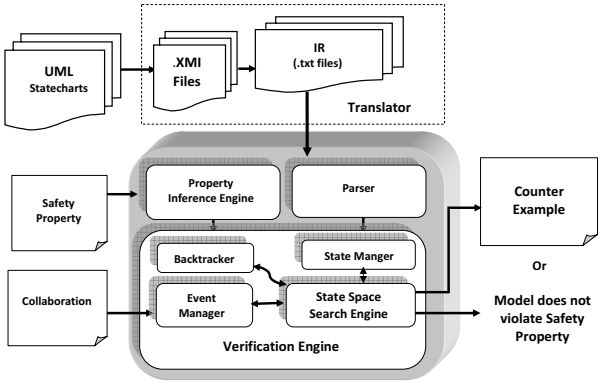


Fig. 6: Architecture of the UML-SMV

VII. Experimental Results

In this section, we describe the results obtained from the verification of UML state chart model of SIP. The results are summarized in the Table I. When the correct model is subjected to the verification, we found that UML-SMV verifier explores 121 states in 321ms to confirm the model's correctness.

We then introduced an error to the SIP model. The "CALLER" is allowed to converse without waiting for the call establishment with the "CALLEE". The verifier successfully detected the error, by exploring only 12 states in 76.26ms. The "Counter Example" generated is of length 10.

VIII. Related work

The protocol verification has been a research issue since last two decades. There are several research articles describing the approaches to verify the protocols. The [8], [9] illustrate

TABLE I: Verification Results of SIP

Input	UML Model complexity		Verification Complexity		
	Max No. of States	No. of Events	No. of States Explored	Length of the error trace (Hops)	Verification time in ms
Incorrect Model	$\leq$360	14	12	10	76.26
Correct Model			121	–	321

verification of cache coherence and authentication protocol respectively. The usual approach is to describe the protocol design in the input language of the model checker, such as PROcess MEta LAnguage (PROMELA), Agros, and Property Specification Language (PSL) and use the model checker like Simple Promela INterpreter (SPIN) [10], Symbolic Model Verifier (SMV) [11] and IBM-RuleBase [12] to verify the desired properties. In our approach, we have used the UML state chart diagrams to model the behavior of the protocols, as UML is a visual modeling language and easy to understand. The state chart diagrams are automatically translated to an intermediate representation and verification is done without the aid of any of the existing model checker.

IX. Conclusions

In this paper, we have presented an automatic protocol design verification technique. This is a complementary approach for testing and this enhances the quality of the end product. We have taken verification of Session Initiation Protocol (SIP) as a case study and following are our findings

- Verification algorithm quickly finds the design errors. In case of SIP design verification, verifier identified error in 76.26 ms.
- As algorithm considers only relevant states, number of states explored would be less. In the specific case which we attempted, we find that 121 states are explored to determine the correctness of the model.
- Modeling the behavior of the protocol using UML statecharts is more convenient compared to modeling using formal languages; UML has rich set of visual notations.

In summary, we conclude that our pragmatic verification approach can act as a valuable design aid to protocol designers.

Acknowledgment

The authors thank IBM Software Lab, Bangalore, India and Mr. Janees Elamkulam, Technical Leader, IBM Software Lab, Bangalore, for supporting us with valuable suggestions while carrying out this work.

References

[1] Edmund M. Clarke,Jr., Orna Grumberg and Doron A. Peled , Model Checking, The MIT press, 1999
[2] D.Harel, Statecharts: A Visual Formalism for Complex Systems, Science Computer Programming 8, pp 231-274, 1987
[3] Z.Manna, A.Pnueli, "The Temporal Logic of Reactive and Concurrent Systems," Springer Verlag, New York, 1992
[4] Valmari,A.: The State explosion Problem, Lectures on Petri Nets I:Basic Models, LNCS 1491, Springer- Verlag (1998) 429-528
[5] Alan B. Johnston, "Understanding the Session Initiation Protocol" Second edition, Artech house, ISBN 1-58053-655-7, 2004
[6] XML Metadata Interchange, http://www.omg.org/technology/documents/formal/xmi.htm,visited on 17/10/2008
[7] BOUML An open source UML modeling tool, available at: http://sourceforge.net/project/, visited on 17/10/2008.
[8] Alan J. Hu, M. Fujita, Chris Wilson, "Formal verification of HAL S1 system Cache coherence Protocol". In proceedings of Int. conference on computer design, IEEE, 1997
[9] T. Y.C. Woo, Simon S. lam, "Design, Verification and Implementation of Authentication Protocol", In proc. of Int. conference on Network protocols, pp 81-90, 1994.
[10] Gerard J. Holzmann, "The Model Checker Spin", IEEE Trans. on Software Engineering, Vol. 23, No. 5, (1997),279-295
[11] Kenneth L. Mc. Millan, "Symbolic Model Checking: An approach to the state explosion problem", Ph.D thesis submitted to Carnegie Mellon University (CMU), 1992
[12] 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.

Prashanth C.M. 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 a life member of Indian Society of Technical Education. He has published several papers in refereed international conference proceedings and journals.

K. Chandrashekar Shet is a Professor in the department 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.