BEHAVIORAL VERIFICATION OF BOM BASED COMPOSED MODELS

Imran Mahmood[(a)], Rassul Ayani[(b)], Vladimir Vlassov[(c)] , Farshad Moradi[(d)]

[(a)] [(b)] [(c)]Royal Institute of Technology (KTH), Stockholm, Sweden
[(d)]Swedish Defense Research Agency (FOI), Stockholm, Sweden

[(a)]imahmood@kth.se, [(b)]ayani@kth.se, [(c)]vladv@kth.se, [(d)]farshad.moradi@foi.se

ABSTRACT

A verified composition of predefined reusable simulation components such as BOM (Base Object Model) plays a significant role in saving time and cost in the development of various simulations. BOM represents a reusable component framework and posses the ability to rapidly compose simulations but lacks semantic and behavioral expressiveness required to match components for a suitable composition. Moreover external techniques are required to evaluate behavioral verification of BOM based components. In this paper we discuss behavioral verification and propose an approach to verify the dynamic behavior of a set of composed BOM components against given specifications. We further define a Model Tester that provides means to verify behavior of a composed model during its execution. We motivate our verification approach by suggesting solutions for some of the categories of system properties. We also provide a case study to clarify our approach.

Keywords: Composability, model verification, deadlock detection, BOM, SCXML, model execution

1. INTRODUCTION

Verification is typically defined as the process of determining whether a model is consistent with its requirement specifications and whether it will satisfy the requirements of the intended application (Petty 2009). Verification is concerned with the analysis of accurate transformation of the requirements into a conceptual model. Model Verification deals with building the model right (Balci 1998) and conceptually correct. Composability on the other hand is an important term used in modeling and simulation. It is the capability to select and assemble simulation components in various combinations into simulation systems to satisfy specific user requirements (Petty & Weisel 2003). One conceptual approach to increase the efficiency and effectiveness of complex model development is, based on reusable model components (Lehmann 2004). A composeable simulation model component thus can be defined as a reusable, self-contained and independently deployed software unit that conforms to a component model (Bartholet et al. 2004), has well defined functionality and behavior and is usable in a variety of contexts (Moradi 2008). Base Object Models (BOMs) provide a framework to define and characterize these components at a conceptual level. BOM is a SISO standard and encapsulates information needed to formally represent a simulation component (Gustavson 2006).

The term Composability carries varied meanings and views in research literature that differ primarily by its different levels. It is essential to consider these different "levels" of composability, in order for them to be meaningfully composed. Medjahed et al (Brahim & Athman 2006) introduces a multilevel composability model in which the composability of Semantic Web Services is checked in four levels: Syntax, Static Semantic, Dynamic Semantic and Qualitative level. In Modeling and Simulation (M&S), these levels are also brought into consideration during the composition process of the model components. *Syntactic Composability* means that the components have the ability to fit together as is concerned with the matching of syntactic information, such as message name and number of parameters etc. *Static Semantic Composability* refers to a meaningful and computationally valid coupling of components whereas *Dynamic Semantic Composability* deals with the behavioral correctness of the composition. Composition of models becomes more challenging when models are heterogeneous in terms of their formal specifications i.e., when they have different structural and behavioral specifications (Sarjoughian 2006).

Various approaches have been developed to evaluate different levels of composability. An interesting approach has been proposed in (TEO & SZABO 2008) that deals with the Syntactic and Semantic level of composability and proposes an integrated approach for model reuse across multiple application domains. In a similar work we suggested a rule-based seven-step process (Moradi et al. 2007) to calculate the composability degree of a particular composition. It

suggests that based on a given scenario, a set of BOM components can be discovered from a BOM repository and matched to analyze their composability degree at three different levels. It was however mainly focused on Syntactic and Static Semantic level of composition. We further proposed another method in (Mahmood et al. 2009) which was mainly focused on matching the structure and behavior of the BOM components at Dynamic Semantic level. Our approach suggested in (Mahmood et al. 2009) was an elementary framework to match BOM state-machines by analyzing their structure and execute them in a runtime environment.

In this paper we revisit and extend our state-machine matching process and apply it to perform Behavioral Verification analysis on a given model composition. In this extended framework we introduce a *Model Tester* to define and compare the requirement specifications for verification and use this Tester during the model execution to verify that the behavior of components being composed match each other and that they can correctly interact with each other to meet their collective objectives. Subsequently we aim to convene our methods with Case Study scenarios in order to provide the proof of concept. Essentially behavior verification problem looks at a goal oriented correct execution of a given composition of components. Solutions to such problem would enable simulation modelers to select and compose various reusable components and verify that their composition would work correctly and satisfy their requirements and intended objectives. We summarize the primary contribution of this paper as follows:

We introduce behavioral verification of BOM based model composition and propose a model tester to represent requirement specification in form of states. We suggest using this tester with our revised state-machine matching process to verify the composed model during execution through an instrumentation technique. Based on this approach, we further suggest the design of a behavioral verification framework that takes candidate BOMs and Tester as input and perform automatic verification. We also discuss different system specification properties and as an example contribute a solution for the verification of deadlock freedom and apply it in a case study.

The rest of the paper is organized as follows: Section 2 formulates the discussion of Behavioral verification and its different methods. Section 3 contains our proposed behavioral verification process for BOM composition. Section 4 provides the details and implementation of the verification framework. A case study is presented in Sections 5 and Section 6 concludes the paper.

2. BEHAVIORAL VERIFICATION

In this section, we discuss Behavioral Verification in detail, highlight different methods for verification and describe various classes of system properties and their representation as requirement specifications. We further propose design of our model tester and its use.

As previously defined, behavioral verification is a process through which we identify that a given set of model components possess correct behavior such that when they are composed they satisfy a given criteria.

2.1. Methods of Verification

Based on the techniques used, the methods of verification can be classified into four groups: I) Informal, II) Static, III) Dynamic & IV) Formal (Balci 1998). For each group various tools and techniques have been suggested within M&S community. Informal techniques are mainly based on inspections, domain expert reasoning and comparison with the similar existing verified models. Static verification techniques primarily focus on the assessment of static model design including structural, syntax and semantic analysis. They are called static because they can be performed without the execution. Dynamic verification techniques require the execution of the model and attestation of its behavior. These techniques are based on instrumentation, abstract level execution, cause-effect graphing and reachability analysis. Formal verification techniques present high degree of difficulty while trying to prove correctness of the model using mathematical methods including induction techniques, inference models, deduction logic, predictive calculus, correction proofs, bisimulation, and model checking (Balci 1998).

In this paper our focus is on two verification methods from the Static and Dynamic groups namely Structural Analysis and Model Instrumentation. Structural Analysis is used to examine the model structure using rule-based evaluation and to verify if it adheres to the principles of Finite State Automata. Model Instrumentation involves insertion of a “Tester” instrument in the composed model to observe the behavior during abstract level execution. We refer it as Dynamic verification.

2.2. System Properties

The system properties can generally be classified into following groups (Berard & Bidoit 2001):

Reachability Properties: state that some particular situation can be reached.

Safety Properties: express that under certain conditions, something never occurs.

Liveness Properties: express that under certain conditions, something will ultimately occur.

Fairness Properties: state that under certain conditions, something will (or will not) occur infinitely often.

For each group, various cases and their solutions are commonly discussed during the verification process. Reachability and safety properties are usually the most crucial to the system correctness. Safety properties represent characteristics of the system such that undesirable event will not occur. Deadlock freeness is a special property commonly categorized as safety property however theoretically its classification as safety or liveness property is debatable (Berard & Bidoit 2001). In this paper we discuss deadlock-freeness as a safety property. We also propose solution to this common issue of verification in the later section.

For verification purposes, the composed model under consideration needs to be accompanied with a set of requirement properties described in form of a *model tester*. Figure 1 illustrates this relationship:

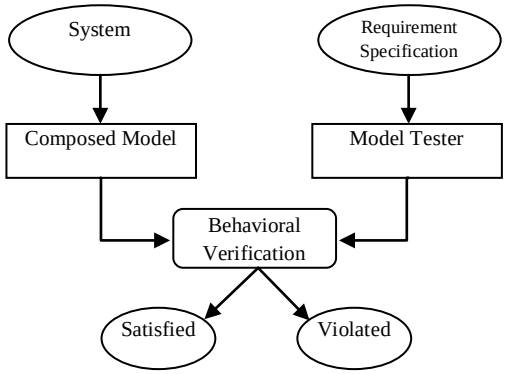

Figure 1: Behavioral Verification Relationship

2.3. Model Tester

In order to describe the requirement specifications, we define a "Model Tester". The Model Tester should be conceptualized as a tester device which is attached to the system during its execution to detect faults. The Model tester mainly consists of the specifications of behavioral properties representing desirable or undesirable situations and incidences. The Model Tester design is specified by the modeler in the form of a state-machine and consists of *good* or *bad* states that express a particular property and represent any of the families of system properties. Each state is guarded by a conditional trigger representing an event or a sequence of events (or global system change) which is sought to occur during the model execution. Good states are expected to be reached while bad states are not expected to be reached and thus the result of behavioral verification depends on the reachability (or un-reachability) of these states in the model tester.

Model tester is basically a passive component that only observes the interaction of the components of the composed model during the abstract level execution and changes its respective states once a desired or undesired behavior has been displayed by the composition hence verify the overall behavior.

Most states in the tester are exitable and are accompanied with an internal state reachability counter. Once they are reached the counter is incremented and then the tester is reset to its initial state. This is useful to record the number of times a specific behavior has been displayed by the composition and thus can be eventually utilized to collect statistics. Some states in the tester are however un-exitable and once reached halts the tester. This means either the composition has reached a successful completion of its goal(s) or it has crashed abnormally or deadlocked. Presence of such states means that the system has one or more terminating conditions.

We suggest the following general guidelines to create a Model Tester:

- A Ready state is defined to initialize the model tester in a neutral state.
- The desirable goals or objectives of a system usually expressed by reachability properties are defined as good states.
- Unwanted situations are expressed as negative reachability properties and are defined as bad states. This should be noted that Model Tester doesn't represent states of the member components of a composition; instead it contains states representing a global change or an effect of the combined behavior of the composition.
- An event or a sequence of events from the composed model is defined as condition for reaching each of these states.
- Commonly known correctness standards such as freedom of Deadlock as safety, absence of Livelock as liveness and Starvation freeness as fairness are general issues in verification process and are modeled in the tester as correctness standards depending on the domain of application whereas problems related to a specific business model are usually described as reachable properties and thus the objective of the verification is to test their satisfaction or violation.

Overall structure of the model tester is a set of good or bad states representing common safety, liveness and fairness properties and scenario specific reachability properties. The trigger to reach states of common properties is the external signal caused by the detection module integrated in the framework whereas the trigger to reach scenario specific reachability states is caused by probing the trace of the sequence of events exchange during the interaction of the members of the composition. Because for each correctness problem there are different detection algorithms and each varies based on the nature of the system however reachability can be detected by simply looking at the traces of the interaction.

The verification of the generic properties involves external solutions or integration of 3rd party model checking tools. Our proposed verification framework incorporates perpetual development and integration of similar external solutions and is open to alternatives because a variety of methodologies and techniques exists and each has its pros and cons in terms of suitability, accuracy and performance measures. Some solutions use Symbolic Model checking or Temporal Logic where as some use Graphs or Petri Nets. Our aim is to provide a generic runtime verification environment where a composed model can automatically be verified with the help of a model tester for context specific properties, while leaving the choice to the modeler to choose and integrate any external solution for system correctness verification. We however suggest a solution for deadlock detection as an example in the later section.

3. BEHAVIORAL VERIFICATION PROCESS

In this section we discuss our proposed Behavioral Verification Process and the framework in which this process has been implemented. A pre-condition for entering this process is that the components share the same semantic classes of the concepts used in the composition i.e., they have passed the static semantic matching phase as proposed in (Moradi et al. 2007).

Our proposed process is an aggregation of previously proposed state-machine matching process(Mahmood et al. 2009) with an extension of features in the framework to facilitate verification. In (Mahmood et al. 2009) we aimed to propose an initiative for developing a runtime environment for state-machine matching where as in this paper we intend to apply it for verification purpose.

Our verification process is based on a W3C compliant State-machine language and runtime environment called SCXML (State Chart Extensible Markup language) (State Chart XML (SCXML) 2009). This state-machine runtime environment has been used to perform abstract level execution of the composed model. The process consists of four steps as shown in Figure 2. First three steps in this process are preparatory and the fourth step is the execution step in which the state-machines are executed and their interaction is monitored with respect to the Model Tester. If the Model Tester doesn't reach any bad state throughout the execution and has reached all (or some) of the good states then we conclude the composition to be positively verified with respect to the given requirement specifications.

This process is briefly described as follows:

3.1. Parsing

As BOM state-machines are event driven, they are required to exchange events with each other to exit from current state and move to the next state, thus the events provide as guard to exit states. In the first step, BOM behavioral data (including state-machines, their states and corresponding events as exit conditions) of all the participating components are ingested by Parsing and their structure is loaded in the system. This step will produce a list of:

- State-machines for each component participating in the composition
- States for each state-machine
- Events (Send/Receive) of each component

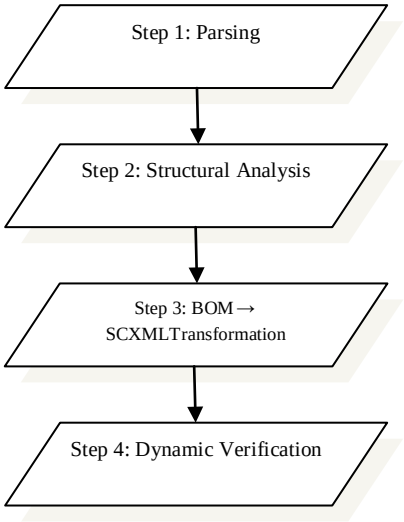

Figure 2: Behavioral Verification Process

3.2. Structural Analysis

In the second step all state-machine objects are sequentially passed through a set of Rules for structural

analysis. These rules are used to check whether a particular state-machine is structurally suitable for the composition or not. If a state-machine is verified by these rules, only then the verification process proceeds otherwise the composition becomes invalid for the given set of components.

Following is the proposed set of rules:

Rule1: Existence of Exit Condition
Each state in a state-machine must have an exit condition or otherwise it should be declared as final state.

Rule2: Existence of a sender for each receiver
In any of the participating state-machines for every Receiver waiting for an Event E there must exist a Sender that is supposed to send E.

Rule 3: Terminal Condition(optional)
If the composed model is terminating then there must exist at least one state marked as final in at least one state-machine among all the participants, such that at least one exit condition (event) leads the composed model to this state.
Violation of Rule-1 means that the execution path of a participating state-machine is broken and does not lead to a final state. Breach of Rule-2 means that there may be a situation when the state-machine will wait for an event endlessly as no corresponding sender is present to send the desired event. Rule 3 optionally evaluates the terminating condition of the model and if satisfied expresses that the model has a termination point. There may be cases where an execution is non-terminal and the system runs for an indefinite time but for simplicity we do not consider them. If all three rules are validated, we can continue to the next step.

3.3. Transformation (BOM to SCXML)
In the third step, BOM objects are transformed to SCXML format. Each state-machine will be transformed to a separate SCXML document. The term transform refers to the fact that the objects are converted from BOM to SCXML format. Here we also provide Model Tester in SCXML format so that it can also be injected in our runtime framework.

3.4. Dynamic Verification
The fourth step deals with the abstract level execution of the state-machines and their verification with respect to the model tester. In this step all components are subjected to an execution environment using verification framework. We discuss the details of our proposed Verification Framework in the next section.
In this step, all state-machines are set to their initial states. When the execution begins, each state-machine participates in a series of event exchange and as a result moves to its next state until it reaches its final state (if any, as not all members may possess final state). During this execution the verification framework uses external methods (if provided) to verify generic system properties and also monitors overall system behavior by observing events or sequence of events and trigger signals to the model tester for state change. As an example we suggested a method to detect deadlock in our framework. If the execution does not encounter any deadlock we verify that the composition satisfy the required safety property i.e. deadlock freedom. If both generic properties and context specific properties are satisfied during the entire execution time we infer that the given composition is verified with respect to behavior.

4. BEHAVIORAL VERIFICATION FRAMEWORK
In this section we discuss our proposed Behavioral Verification Framework design and its execution in detail. This framework uses SCXML to input and execute XML structure of a set of participating event driven state-machines. A finite state machine (FSM) is event driven if its inputs and outputs are modeled in the form of events or messages.

4.1. Framework Design:
Our framework consists of the following modules:

- Message Controller (MC)

This module is used as a communication platform. It is an asynchronous Message Controller used to send and receive multiple messages at a time. MC follows “Post Office” protocol and consists of an address space to accommodate each component for communication. Every component needs to register its name to be used as a unique identifier for the address so that it can be allocated with an INBOX. Each INBOX is a queue of Messages so that all the messages addressed to a component are stored in the INBOX in form of a FIFO queue and the component can process them one by one. Also there is a common OUTBOX for all outgoing messages, shared by all the components where they can place their outgoing messages addressed to each other. These outgoing messages are dispatched periodically and MC is responsible to place them in the pertaining

recipient's INBOX from where the receivers can retrieve and process them.

- **Component Executor (CE)**

A Component Executor module represents one instance of a component in the system and has a unique ID (i.e., the name of the component). Each CE has a SCXML engine object (State Chart XML (SCXML) 2009) to transact state-machine and can also communicate with the "Message Controller" to send and receive messages. CEs are implemented in form of Java Threads in order to perform parallel interaction between the components. They use SCXML document to initialize their internal SCXML engine and set the current state to their initial state. This is how a particular state-machine model is assigned to an executor thread in the system. Then an Event listener of each component is invoked and ready to fire events. Each time a suitable event is fired CE transacts its internal state-machine to the next state. A CE will stop its thread if it reaches its final state (if it has one as described by the SCXML model). If all CEs having final states are stopped, the process will successfully terminate.

- **Tester Executor (TE)**

This module is similar to Component Executor as it also has an SCXML state-machine model (representing our Model Tester) and SCXML engine. But it is passive and only receives events and changes its internal state. It also has a state reachability counter to count number of times a particular state (good or bad) has been reached. It can be modeled to have exitable or non-exitable, good or bad states. In case an exitable state if reached, the tester increments the counter and automatically resets back to the ready state to wait for any other event.

- **Event lookup Table**

This module contains a list of all the Events used in the composition and are parsed from BOM. This list of Events is accessible to each Component Executor so that being at a particular state they can locate the information of the next occurring event expected to be sent or received. This information is used to let a component either send an event to a specified recipient or wait for an event from a sender during each logical time step.

- **Monitor**

A monitor is used to observe global behavior of the composition. A monitor acts as a comparer between the member state-machines and the Model Tester. It monitors the overall execution, waits for the important exchange of events or sequences of events and fire necessary events for the Model Tester to update its state.

- **Deadlock Detector**

We propose this module as an example solution to one of the generic system property i.e., deadlock freedom.

A deadlock can be defined as: A set of components are said to be in a deadlock if each of them is waiting for an event that only another component in the same set can cause.

Deadlock is of two types: i) Total Deadlock ii) Partial Deadlock. If all components of the composition are in the waiting set then it leads to a total deadlock whereas if only some components are in the waiting set then it is a partial deadlock.

We develop deadlock detector component to observe any possible detection of deadlock and notify the monitor. This detector periodically collects the list of those components which are waiting for any event to proceed and apply our proposed deadlock detection method.

- **Deadlock Detection Method**

This method assumes a set Q of tuples of waiting components called "Receivers" which are waiting for their corresponding "Senders". This arrangement is used to establish a wait-for relationship for each waiting component.

$$Q = \{(r, s) \mid r \in \text{Receiver}, s \in \text{Sender}\}$$

Based on the information collected from the set Q, we construct a "Wait-for Graph" and fill it with vertices representing all the components from the waiting list at a particular instance of time. First a receiver will be inserted then its corresponding sender will be inserted. Then a directed edge is connected from the receiver to the sender showing "Wait-For" relation. When all the components from the waiting set have been inserted in the graph, we apply a standard Depth First Search (DFS) algorithm to find any possible cycles within the graph.

A cycle is defined as a closed loop of two or more vertices connected in a closed chain. If there exists a cycle, it means that some components are waiting for each other in a chain and thus there is a deadlock. In practice cycle detection in a Graph is done using Depth First Search (DFS) coloring algorithm. Color markers are used to keep track of which vertices have been discovered. White marks vertices that have yet to be discovered, gray marks a vertex that is discovered but still has vertices adjacent to it that are undiscovered. A black vertex is discovered vertex that is not adjacent to

any white vertices. Edges that lead to a new unvisited vertex are called Tree Edge whereas the edges that lead to an already visited vertex are called back-edges. By definition: "A directed graph G is acyclic if and only if a depth-first search of G yields no back edges". So DFS essentially detects any back-edge for finding a cycle(Skiena July, 1998). If we detect any cycle in the wait-for graph of a set of waiting components then we have found a deadlock which means there exists a cyclic chain of two or more components that are waiting for each other. If the number of waiting components in the set is equal to the total number of components participating in the composition then it will be a Total deadlock.

4.2. Framework Execution:

Figure 3 illustrate the verification framework:

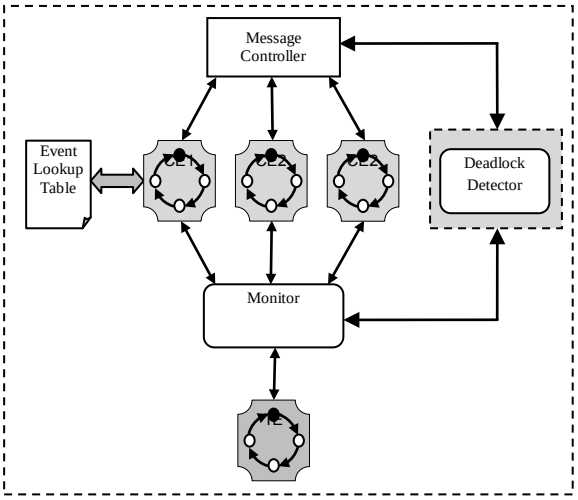

Figure 3: Behavioral Verification Framework

When the execution begins all Component Executors (CE) initialize their SCXML state-machine model objects to their initial states and also register their IDs in the Message Controller (MC) for address space allocation. When all the participating components are ready, they start dynamic interaction with each other by sending or receiving event messages using MC. Based on their current state each CE identifies the next event it is supposed to send or receive using Event Lookup table. From this table each CE can fetch information about the event which is required to exit its current state. If it is an outgoing event (i.e., the source of this event is the CE itself) then the CE will act as a Sender whereas if it is an incoming event (i.e., CE is the target of this event then it has to wait as a receiver).

In case of being a Sender a CE will prepare a message object by stamping its ID as a source, its recipient's ID as target and message parameters taken from the Event Object (previously fetched from the Event lookup table) and transmit it using Message Controller. MC will place this message in the Outgoing queue which will be dispatched in the next time generation. This CE will also fire that event in its internal state-machine and go to the next state.

If the CE is a receiver it will wait until it has received any message from the Message Controller. In case of multiple arrivals the first message from the INBOX is retrieved (as Queue is FIFO) and processed by firing the event internally and stepping to the next state. Each time there is a Sending or Receiving of an event, based on which a component transacts its state-machine to the next state, we let the system advance to next logical time step.

Deadlock detector performs its routine on each time step and checks possible deadlock occurences. In case of detection, it alerts the monitor which in turn sends a signal to the Tester Executor (TE). TE moves to the corresponding bad state and halts the execution. Monitor is also responsible to observe the behavior which corresponds to context specific reachability properties in the Model Tester and in case of finding valid sequence of events; it triggers an alert to the Model Tester. There is a performance bottleneck when we perform deadlock detection routine at each time step thus we propose to schedule them after an N interval of time where N is defined by the modeler based on the size of the composition and system resources.

The successful completion of the abstract level execution means that there is no violation of generic system properties and the context specific properties are reachable (or unreachable in case of negative reachability properties) and thus the behavior of the member components is verified.

5. CASE STUDY

In order to test our verification approach, we have considered a Restaurant Case Study. We discuss two scenarios in this case study.

5.1. Scenario A

The basic theme of the scenario is that customers arrive to a restaurant, order food, eat, pay their bills and then leave. There are 3 components in this scenario: *Customer, Waiter and Kitchen.* A sequence diagram in Figure 4 represents the pattern of interplay between these components.

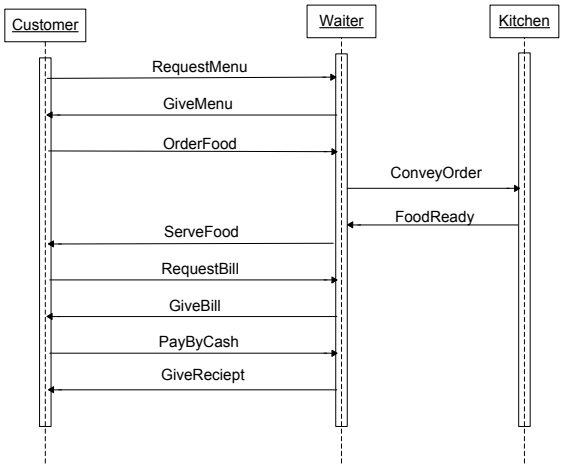

Figure 4: Restaurant sequence diagram

Figures 5, 6 & 7 represent the individual state-machines models of each component involved in the scenario. Green text with an up arrow represents a send event (as an exit condition) whereas blue text with a down arrow represents a receive event.

In the first step of the verification of composed restaurant model the state-machines are parsed from the BOMs and state-machine objects are produced. In step 2 the corresponding data is passed through structural analysis, which checks the state-machines against the rules and if passed the static analysis phase is declared as successful.

In the next step SCXML documents are generated, each representing the corresponding state-machines of the customer, waiter and kitchen components. Also a Model Tester is defined in the form of SCXML as represented by figure 8. Each SCXML document is then executed in runtime environment. When the Component-Executors are initialized to their initial states they identify their next action The first CE which is responsible to send an action in the Message controller is Customer and the action is *RequestMenu.* Waiter component is waiting for this event and as soon as it receives *RequestMenu,* it moves on the next state and Send *GiveMenu.* On receiving *GiveMenu,* the Customer starts to *OrderFood.* Thus each component sends and receives events in the same manner until the Customer component (which is the only component having a final state) reaches "Leaving" state and thus the abstract level execution is successfully completed and this is how the three BOMs are verified and validated using dynamic analysis.

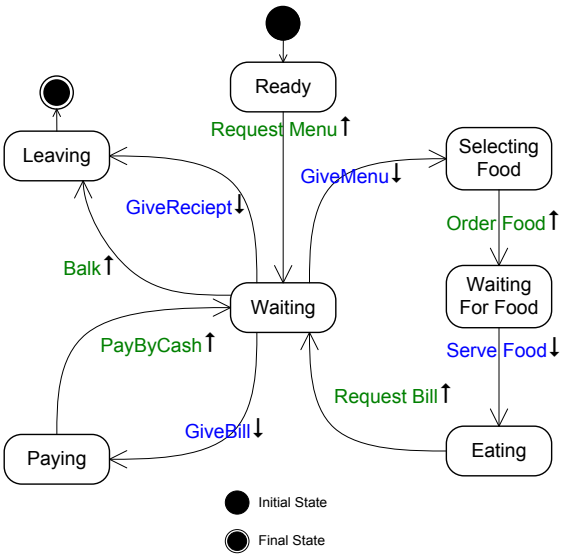

Figure 5: Customer state-machine

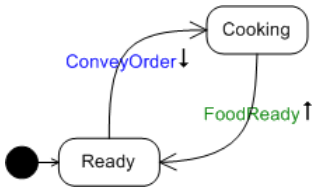

Figure 6: Kitchen state-machine

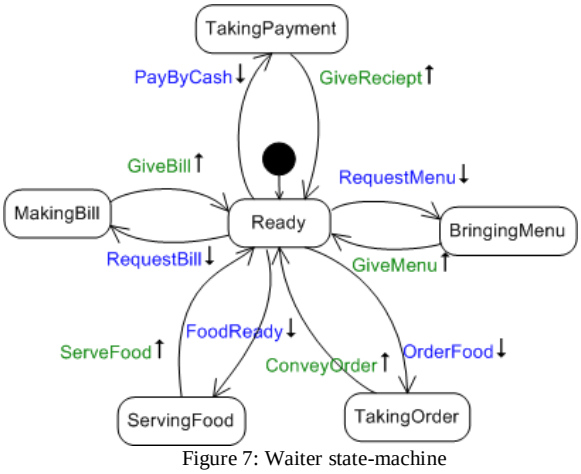

Figure 7: Waiter state-machine

Sales and Dinning are considered to be good states as they promote business. Leaving the restaurant without payment is a bad state as it incurs loss. Deadlock is a bad-state marked in gray and is un-exitable which means if reached halts the execution.

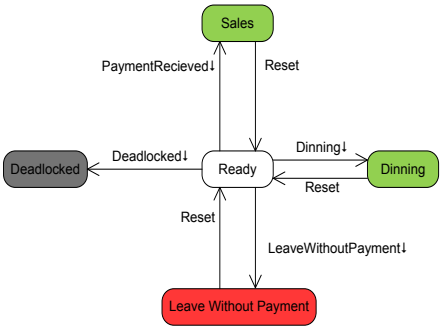

Figure 8: Model Tester

The state transition in Model Tester could occur due to a sequence of events exchange between the components e.g., in order to reach "dinning" state, the correct sequence of events is:

Customer!OrderFood → Waiter!ConveyOrder →
Kitchen!FoodReady → Waiter!ServeFood →
Customer!RequestBill → Waiter!GiveBill →
Customer!PayByCash → Waiter!GiveReciept

When this sequence is noticed by the monitor, it will fire "Dinning" event to the tester. Another sequence of events is:

Waiter!ServeFood → Customer!RequestBill→ Customer!Balk

When this sequence is noticed by the monitor, it will fire "Leave without payment" event which is a bad state because Waiter *serves food* but Customer after *requesting bill* goes to waiting state and leaves the restaurant without paying. Note the *Balk* event that can be fired from the waiting state.

If during the entire execution of the restaurant scenario, no bad states have ever been reached and the number of times the good states are reached is greater than zero, then the composed restaurant model is said to be dynamically verified in terms of behavior and with respect to the given specifications.

5.2. Scenario B

In this scenario, we modify the behavior of waiter, i.e., after taking the order, he makes a Bill and gives it to the customer and waits for the payment. Once Customer pays the bill, only then he conveys the order to the kitchen. Figure 9 represents the modified waiter.

This modification passes Structural Analysis because all the events are same only their order is changed. However when execute it in the dynamic verification step, it detects deadlock because the customer waits for the food to be served where as the waiter waits for the bill to be paid. Since two waiting components wait for each other so our deadlock detector detects a closed cycle in the wait-for graph and thus notifies the detection of deadlock to the monitor which in turn puts the Model Tester in the "Deadlocked" state and thus the composition is not verified due to deadlock.

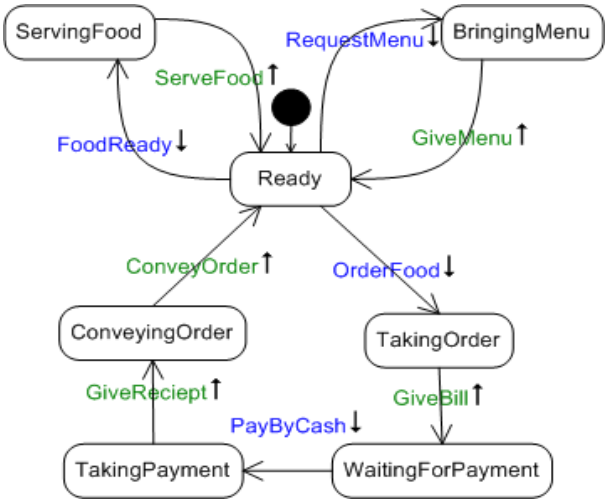

Figure 9: Waiter state-machine

6. CONCLUSIONS AND FUTURE WORK

In this paper we discussed Behavioral Verification and its different methods. We proposed a *Model Tester* to be used as an *Instrumentation Technique* for the dynamic behavioral analysis and proposed a verification framework using this technique. We also discussed different classes of the system correctness properties and suggested to represent them as good or bad states in the model tester. Deadlock Freedom being a commonly accepted correctness standard is suggested to be included in the model tester as a key requirement beside other scenario specific reachability properties. We further proposed a method to detect deadlock and implemented it as an integrated module in our verification framework. Similar modules can be developed and added to the framework to detect other safety or liveness properties thus increasing the credibility of the verification process. At the end we provided a restaurant case study as a proof of concept.

With the help of Behavioral Verification Process we can verify any BOM composition and essentially help the modeler to study and analyze the behavior using model tester which is based on system properties. These properties from different classes cover a variety of verification aspects and evaluation standards. By verifying the behavior of a set of BOM we can assert that a necessary condition in the process of BOM composition is fulfilled.

We are further interested to use formal methods such as Petri Nets and Linear Temporal Logic to verify the

composition based on more complex requirement specifications including livelock freedom and starvation freeness. We further strive to generalize our verification approach for other component frameworks such as DEVS (Discrete Event System Specification) to match and verify model compositions.

REFERENCES

Balci, O 1998, 'Verification. Validation and Testing', in *Handbook of Simulation: Principles, Methodology, Advances, Applications and Practice*, John Willey & Sons.

Bartholet, RG, Brogan, DC, Reynolds, PF & Carnahan, JC 2004, 'In Search of the Philosopher's Stone: Simulation Composability Versus Component-Based Software Design', *Simulation Interoperability Workshop*, Orlando.

Berard, B & Bidoit, M 2001, *Systems And Software Verification*, Springer.

Brahim, M & Athman, B 2006, 'A Multilevel Composability Model for Semantic Web Services', *Journal of IEEE Transactions on Knowledge and Data Engineering*, vol VOL. 17, No. 7.

Gustavson, P 2006, 'Guide for Base Object Model (BOM) Use and Implementation', Simulation Interoperability Standard Organizations (SISO).

Lehmann, A 2004, 'Component-Based Modeling and Simulation – Status and Perspectives', *Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications*.

Mahmood, I, Ayani, R, Vlassov, V & Moradi, F 2009, 'Statemachine Matching in BOM based model Composition', *13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, Singapore.

Moradi, F 2008, 'Framework for Component Based Modeling and Simulation using BOMs and Semantic Web Technology', PhD Thesis, KTH/ICT/ECS AVH-08/05—SE, 2008, Stockholm.

Moradi, F, Ayani, R, Mokerizadeh, S, Shahmirzadi, GH & Tan, G 2007, 'A Rule-based Approach to Syntactic and Semantic Composition of BOMs', *11th IEEE Symposium on Distributed Simulation and Real-Time Applications*.

Petty, MD 2009, 'Verification and Validation', in *Principles of Modeling and Simulation*, John Wiley & Sons.

Petty, MD & Weisel, EW 2003, 'A Composability Lexicon', *Proceedings of the Spring Simulation*, Orlando, FL.

Sarjoughian, HS 2006, 'Model Composability', *Proceedings of the Winter Simulation Conference*.

Skiena, SS July, 1998, *The Algorithm Design Manual* , Springer.

'State Chart XML (SCXML)' 2009, W3C.

TEO, YM & SZABO, C 2008, 'CODES: An Integrated Approach to Composable Modeling and Simulation', *41st Annual Simulation Symposium*, Ottawa.