Statemachine Matching in BOM based model Composition

Imran Mahmood,
Rassul Ayani,
Vladimir Vlassov
Royal Institute of Technology (KTH),
Stockholm, Sweden
{imahmood, ayani, vladv}@kth.se

Farshad Moradi
Swedish Defense Research Agency (FOI),
Stockholm, Sweden
farshad@foi.se

Abstract

Base Object Model (BOM) is a component-based standard designed to support reusability and Composability. Reusability helps in reducing time and cost of the development of a simulation process. Composing predefined components such as BOMs is a well known approach to achieve reusability. However, there is a need for a matching mechanism to identify whether a set of components are composable or not. Although BOM provides good model representation, it lacks capability to express semantic and behavioral matching.

In this paper we propose an approach for matching behavior of BOM components by matching their statemachines. Our proposed process includes a static and a dynamic matching phase. In the static matching phase, we apply a set of rules to validate the structure of statemachines. In the dynamic matching phase, we execute the statemachines together at an abstract level on our proposed execution framework. We have developed this framework using the State Chart Extensible Markup Language (SCXML), which is a W3C compliant standard. If the execution terminates successfully (i.e. reaches specified final states) we conclude that there is a positive match and the behavior of these BOMs is composable. We describe the matching process and the implementation of our runtime environment in detail and present a case study as proof of concept.

Keywords: *Statemachine matching; BOM Composition; SCXML; Abstract Level Execution;*

1. Introduction

The ability to compose reusable simulation components in a simulation environment efficiently and effectively is a key need recognized by the Modeling and Simulation (M&S) community. The basic requirement for this is a set of meaningfully composable components that can be coupled together to model and develop interoperable simulations [1]. In order to be able to reuse the simulation components, we have to first check whether they are composable or not. In this section, we will discuss basic factors involved in the composition of simulation components.

In a previous work [2], a rule-based seven step process was proposed. It was suggested that a set of BOM components can be passed through this process in order to match them and analyze their Composability degree. With the help of this process a simulation modeler can evaluate the syntactic, static semantic and dynamic semantic composability of a set of input BOMs. These three composability properties are the different perspectives of a composition. As the name suggests syntactic composition is concerned with the matching of syntactic information, such as message name, mode of action and number of parameters. Static semantic checks the entity and data types and their meaningful interconnection, while dynamic semantic handles matching of the components behaviors, i.e. state machines [7].

In this paper, we have mainly focused on the Dynamic Semantic Matching part of the composition as this problem needs to be addressed further. Basically this part of the composition deals with the behaviors and involves a deeper study of statemachines. It also demands a suitable platform for the implementation.

Although an approach for statemachine matching was mentioned in the previous work [2]; however, based on our current research, we propose a different approach using a World Wide Web Consortium (W3C) standard and introduce a new process to perform dynamic statemachine matching of BOM components. Our proposed approach divides the matching process of BOM components into two main parts. In the first part, each statemachine of the BOM is validated against a set of rules to ensure that they adhere to a finite statemachine (FSM) standard as prescribed by automata theory [3]. This part helps to check that all the participating statemachines meet a given criteria for execution. In the second part, statemachines are executed at an abstract level using our statemachine runtime environment. This environment is built using the State Chart language (SCXML) and an execution engine, which is a W3C standard [10]. The purpose of the abstract level execution is to check if all the statemachines actually behave as they were intended by traversing through their states on exchanging events and finally reach the finish state. If the execution is terminated successfully we can assert that the statemachines have a positive match. The rest of the paper is organized as follows. In section 2, we review some related material, including SCXML. In section 3, we detail our proposed approach in its implementation. In section 4, we present a case study and section 5 concludes our work.

2. Background

Before we discuss our main contribution and elaborate our solution for the dynamic matching of BOM components, we will cover a brief background of different concepts and technologies involved in this research.

2.1 Composability

The key element of this work is the term *Composability*, which is defined by Mikel D. Petty in theory of Composability [4] as follows:

"Composability is the capability to select and assemble simulation components in various combinations into simulation systems to satisfy specific user requirements"

Composability is further divided into three different categories: i) Syntactic Composability, ii) Semantic Composability and iii) Pragmatic Composability[7]. *Syntactic Composability* means that the components fit together, whereas the *Semantic Composability* means that the components work together in a meaningful way [4]. Moreover Syntactic Composability is concerned with the compatibility of implementation details, such as parameter passing mechanisms, external data accesses, and timing mechanisms [4]. In simpler words Syntactic Composability determines whether the components can be connected or not whereas Semantic Composability is concerned with the behavioral validity of the composition. The latter refers to a composition where combined coupling of two or more simulation components is considered meaningful and computationally valid [6]. Pragmatic composability is yet another type which is concerned with the context of the simulation, and that whether the composed simulation meets the intended purpose of the modeler [7].

There have been some significant development in syntactic Composability both within software engineering industry and Simulation communities, but Semantic Composability is still an open ended problem and has inspired many researchers to contribute theoretical and experimental research in this regard [7].

2.2 Base Object Model (BOM)

BOM is a SISO standard [8] defined in form of an XML document. It encapsulates information needed to describe a simulation component. BOM has four main parts: i) Model Identification ii) Conceptual Model iii) Model Mapping and iv) HLA Object Model. The first part is meant to store the metadata of the component, which is basically the general metadata information about the component itself. The second part is the Conceptual Model, which contains information about the pattern of interplay, statemachine, entities and the events of the component. Entities and Events represent data about the real world object models and their interaction in form of Entity types and Event Types [8] whereas the pattern of interplay and statemachine collectively represents the dynamic behavior of the component. The other two parts Model Mapping and HLA Object Model relate to the BOM HLA assembly. Figure 1 represents the Conceptual Model of a BOM:

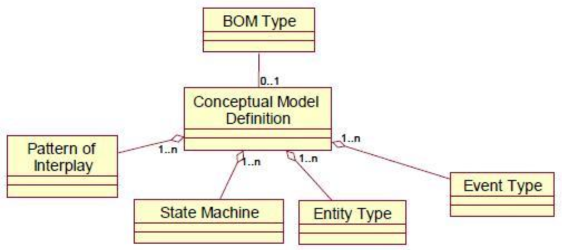

Figure 1 BOM Conceptual Model *(Courtesy to [9])*

Pattern of interplay describes the type and sequence of events and actions that take place among components. Statemachine defines the behavior model of a component and represents the dynamic element of BOM component. The BOM Statemachine provides means to formalize the change in the state of an entity with respect to its corresponding actions, thus in a way it depicts the abstract model of the behavior of the BOM towards each action. Our main focus in this work is indeed the statemachines. A typical BOM Statemachine refers to a particular entity and lists all possible states of that entity, and the exit conditions. An exit condition must be satisfied in order to exit a current state and enter the next state as specified in a state transition table. Each entry in the table is a triple identifying current state, exit condition and next state.

BOM statemachines are basically event driven as they iterate through the states by sending or receiving events. When a statemachine transits to a new state, it performs an action. Each action correlates to a particular event specified in the event type group of the conceptual model. An example of a BOM statemachine is described in Figure 2:

```
<stateMachine>
        <name>NAME</name>
        <conceptualEntity>ENTITY</conceptualEntity>
        <state>
                <name>STATE</name>
                <exitCondition>
                    <exitAction>ACTION</exitAction>
                    <nextState>NEXT STATE</nextState>
                </exitCondition>
        </state>
</stateMachine>
```

Figure 2: BOM Statemachine

2.3 State Chart Extensible Markup Language

The State Chart Extensible Markup Language (SCXML) is a W3C compliant standard [10] which provides a generic statemachine execution environment [5]. SCXML is a general purpose event-based statemachine language and can be used as a standard statemachine framework. SCXML is an extension to CCXML (Call Control Extensible Markup Language) and it was initially designed for voice applications however its usage is open to a wide range of problems. Especially it offers a clean and well-thought out semantics for sophisticated constructs representing finite statemachines [10]. The SCXML Java API implementation consists of the following components:

i) SCXML Document

A typical SCXML document represents the basic structure of a given statemachine. This document is used to input a statemachine model in the runtime environment for an abstract level execution [11]. Figure 3 illustrates the XML structure of this document:

```
<SCXML initialstate= STATE1>
        <state id = STATE1 final = True>
                <transistion event=EVENT NAME
                        target=STATE2/>
        </state>
        <state id = STATE2 final = false>
                <transistion event=EVENT NAME
                        target=STATE1/>
        </state>
</SCXML>
```

Figure 3: SCXML Format

ii) SCXML Parser

Apache Common's SCXML also provides an implementation of an XML Parser that can parse SCXML documents into Object Model. This Object model is required by yet another component called SCXML Executor, which is the core component of the runtime environment. This parser not only parses the elements of SCXML document but also validates its syntax and structure, thus proves to be very useful tool for validating any statemachine according to the given standard [11].

iii) SCXML Engine

Java SCXML engine is capable of executing statemachines defined by SCXML documents, while abstracting out the environment interfaces. The most important of all is the *Executor* which is mainly responsible for initiating the engine. SCXML Engine also provides an *Event Dispatcher* interface for wiring the behavior of SCXML <send> actions so as to receive callback implementation provided to the executor. SCXML also provides an implementation of *Listener* to be registered within the engine, which is informed about the progress of the statemachine via notifications when transitions are followed. This listener is useful to carry out statemachine execution, by triggering events (time based or using input consoles). A *TriggerEvent* class is responsible for firing events. The *Error Reporter* interface is used by the engine for reporting errors to the host environment or logger. The SCXML Engine specification further allows implementations to support multiple expression languages so that SCXML documents can be used in varying environments. Apache's Commons SCXML currently supports different expression languages for expression evaluation, which proves to be very useful for evaluating conditions during the statemachine transactions. [11]

Once the SCXML engine has been initialized, the statemachine progress is based on the events that are fired on it. When an event is fired, if the current set of states has transitions waiting for that event, the statemachine is said to "follow" that transition, which may possibly yield a new set of current states. Most statemachines will ultimately reach a "final" state, wherein the statemachine has said to have executed to completion. Moreover the received events are also logged using Commons Logging. [11]

3. Statemachine Matching

In this section, we will discuss our main contribution and implementation of the proposed solution for the statemachine matching of BOM components.

We define Statemachine Matching of BOMs as: A process by which we can identify that the statemachines of a given set of component models (with a given set of initial states) can correctly interact with each other to perform a joint activity

Statemachine matching provides means to ensure the causality order of events in the composed model [7]. If we find a positive match among a group of state-machines in question, we can say that their pertinent BOMs are dynamically composable It should be noted that the matching of an abstract model is a necessary condition and not a sufficient condition for BOM composition.

In our proposal for matching the statemachines of various interoperating BOM components, we define a process which is divided into four steps:

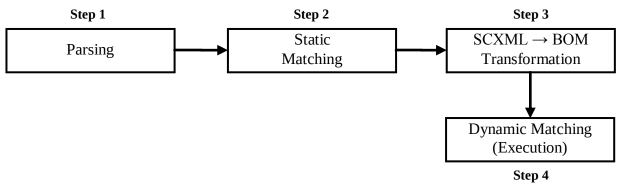

Figure 4 Overview of statemachine matching process

This four step process takes BOM XML document as an input. In the first step, we parse the BOM xml and collect statemachine objects. In the next step, we subject these statemachines to a set of static rules. When all the rules are sequentially validated, we apply a transformation procedure to convert BOM XML to SCXML format. Then in the last step, we finally execute the SCXML statemachines and infer the results from its executing.

Step 1: BOM Parsing

In this step, a BOM XML document is parsed and its corresponding Java objects are generated. Classes of these objects are shown in Figure 5

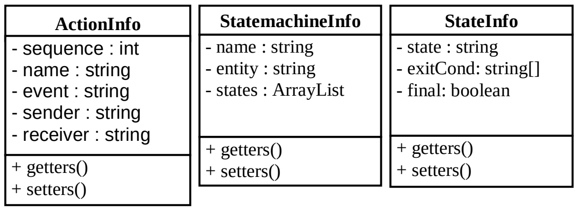

Figure 5: Classes of Java Objects generated by the parser

A StateInfo object represents data of a single state. It has a state name and a string array of Exit Conditions. Each Exit Condition is stored in the following format:

Exit Condition = {Exit Action: Next State}

A StatemachineInfo object stores the metadata of a statemachine such as Name and the Corresponding BOM Entity and a collection of states. An ActionInfo object represents the data structure of the actions involved in the transitions of the states.
Getters() and Setters() functions are used to set or retrieve the data like in Java Beans.

The states are parsed from BOM's statemachine and stored in *StateInfo* objects. An Array List of *StateInfo* objects is then stored in *StatemachineInfo* object. The actions are parsed from BOM's pattern of interplay separately and each action is stored in *ActionInfo* object, an *ArrayList* of which is maintained to lookup for sender and receiver of each action.

Step 2: Static Statemachine Matching

In the next step, the statemachine objects are sequentially passed through a set of rules. We call them ***Static Matching Rules.*** These rules are used to check whether a particular state-machine fits for the matching process. If a statemachine is passed by these rules, only then the matching process can continue otherwise the composition becomes invalid for the given set of components.

Following is the proposed set of rules:

RULE 1: Existence of Exit Condition

All states in a statemachine must have an exit condition.

This rule ensures that a statemachine cannot enter into a state and stay there forever.

The exit condition in Rule 1 is either a send action or a receive action. According to BOM specification [8], "Actions" are responsible to trigger exit conditions in the statemachines. An action is described in the pattern of interplay block. Each action is mapped to an event described in the *EventType* block.

RULE 2: Existence of a send action for each receive action

In any of the participating statemachines for every receive action, which causes an exit condition of a state, there must exist a state that has the corresponding send action.

For all those states in a statemachine, whose exit conditions are based on actions that are expected to be received (called Receive actions) from some other participating statemachines, there must exist a corresponding state in any of other statemachines, that has the same action which is of type Send (called Send action). If both Send and Receive actions are matched then the statemachines can inter-operate otherwise there will be a situation when a statemachine enters a state and waits endlessly for an event to receive whose sender is absent in the composition. The satisfaction of this rule ensures, that there is no state present in any of the participating statemachines, that is depended on an event (Receive action), which will never occur because there is no sender. We have termed the pair of two actions (Receive action & Send action) as Couple.

RULE 3: Terminal Condition

There must exist at least one state marked as final in at least one statemachine among all the participants, such that at least one exit condition leads the statemachine to this finish state.

If there is no final state, we cannot be certain that a joint activity has been completed; instead there is a possibility that statemachines are switching their states in an infinite loop and are stuck in a live lock. Hence reaching to a final state can only tell that an activity has been completed successfully. The problem with the BOM is that there is no provision to mark a final state, as current BOM specification [8] does not support any XML Tag for marking final states. Tough the SCXML specification requires that the final state should be declared. So one way to resolve this problem would be to allow the modeler to select a final state during the matching process and another option would be to compute it by looking at the last action in pattern of interplay and retrieve its corresponding state thus we can assume that state to be the final state.

Rule 1 & 3 are necessary to be fulfilled as they comply with the statemachine standard given by SCXML specification. Whereas Rule 2 ensures that each receiver has a corresponding sender. When all the rules are validated, we can continue to the next step.

STEP 3: Transformation

In this step, the statemachine objects are transformed to SCXML format. Each statemachine will be transformed to a separate SCXML document. The term transform refers to the fact that the statemachines are transformed from BOM to SCXML

STEP 4: Dynamic Statemachine Matching

This step deals with an abstract level execution of the statemachines. We will discuss the internal details of the structure of our proposed runtime environment in this section. Our runtime environment is using SCXML Executor API as an underlying layer for execution. We have extended this layer, by implementing a multi-threaded synchronization of executor instances, each initialized by inputting the SCXML document. Also we have introduced a mutually exclusive shared variable for simulating send and receive of actions. The purpose of this execution is to initialize all the statemachines to their initial states and simulate the send and receive of events to observe the transaction of the statemachines until they reach their final state. This will tell us that all the statemachines have valid and ordered inter communication and thus match each other.

In this step all SCXML documents collected in step 3 are dispatched to the run-time environment for execution. Each document represents a statemachine and carries internal information about its states, their transitional conditions and next states as prescribed by the SCXML structure. Each SCXML document is parsed, and the runtime spawns an **Executor** thread to execute the statemachine described by the corresponding SCXML document. The Executer instance is initialized by the **SCXML Engine.** Multiple Executor threads (one thread per statemachine) are our proposed multithreaded implementation of statemachine execution on top of the SCXML executor interface. Using multiple threads our runtime environment allows concurrently executing and synchronizing instances of different statemachines. Each Executor thread is responsible for the dynamics of the statemachine associated with it, i.e., it performs state transitions, sends and receives events according to the model of the statemachine executed by the Executer. Another component in our execution framework is the **Event Controller,** which is meant to simulate communication of events between the statemachines. Events are passed via a synchronized shared object **Action** which represents a FIFO event channel. The Action object is accessed by Executers (statemachines) by means of synchronized Put() and Get() methods to send and receive events passed over the channel, respectively. Figure 6 represents our proposed statemachine matching process:

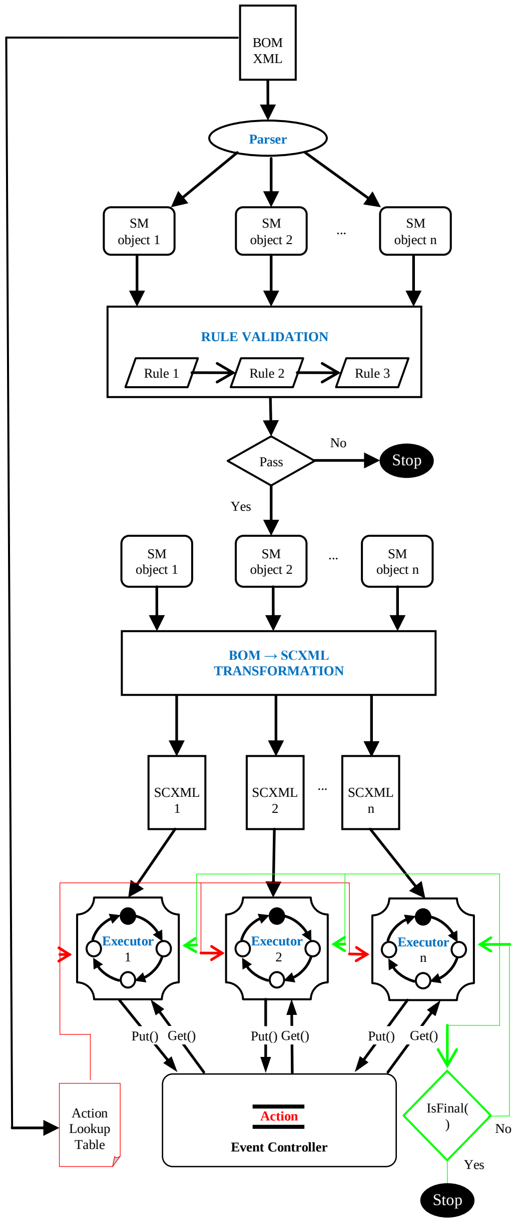

Figure 6: Statemachine Matching Process

When the execution begins, each Executor computes its next expected event (either send or receive) by considering the **Exit_Actions** of the current state. If the Exit Action is of type **"Send"** then the Executor will fire a corresponding event in the Engine internally and move on to next state. It will also execute the Put() method to pass the event to the Event Controller (as if it was actually sent by the BOM component). If the Exit Action is of type **"Receive"** then the Executer will wait until some other Executer (statemachine) sends the expected event, i.e. puts the event in the Event Controller.

This is how each the Executor will simulate send and receive events and cycle through its corresponding states. Each time a state is traversed, it is checked against **IsFinal()** and if the final state is reached then that particular Executor thread will stop the execution and report the successful dynamic match.

In Step 4, all the participating statemachines execute simultaneously. They traverse through their internal states by sending or receiving events and transit to their next states. If any of the statemachine reaches the marked final state, we can conclude that there is a positive match. The execution framework allows detecting deadlocks. A set of statemachines is deadlocked when each statemachine in the set is waiting for an event which can only be caused by another statemachine in that set. The statemachines executed in our framework are in a (total) deadlock when all statemachines Executors are executing the Receive exit action (i.e. waiting for an event) and the Event Controller channel is empty. This will essentially help us to determine the problem and help the modeler to resolve the deadlock by modifying the statemachine model.

4. CASE STUDIES

In order to test our matching approach, we have considered a Case Study with two scenarios. One will represent a successful scenario while the other will represent a scenario where even though the statemachines will pass the static matching phase, but they will face deadlock during the execution. These case studies are simulation of a Restaurant.

4.1 SCENARIO A:

The basic theme of this scenario in the Case Study is that customers arrive to a restaurant, order food, eat pay their bills and then leave. There are five entities in this scenario: *Customer, Waiter, Queue, Table and Chef.* A sequence diagram in Figure 7 represents the pattern of interplay between these entities. The purpose of this figure is to give an overview of the interaction between the entities.

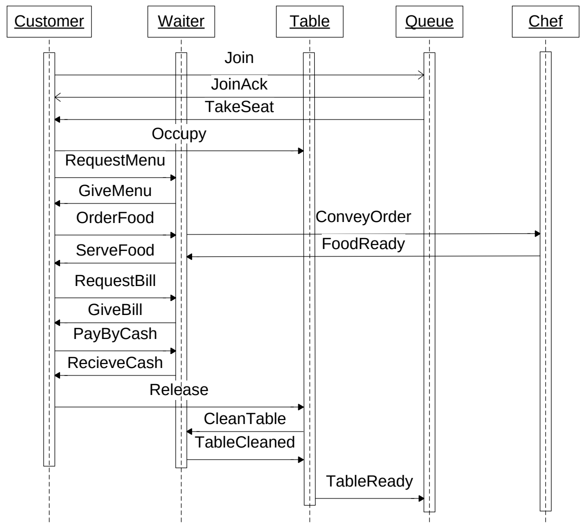

Figure 7 Sequence Diagram

All of the statemachines are developed in a Restaurant BOM. This BOM was subjected to our Statemachine matching process. In step 1, the BOM was parsed and the objects *(StatemachineInfo, ActionInfo and StateInfo)* were populated in a data collection. Each *StateInfo* is a triple consisting of *current state, exit condition* and *next state.* Whereas each *ActionInfo* is a triple consisting of *Event name, Sender* and *Receiver.*

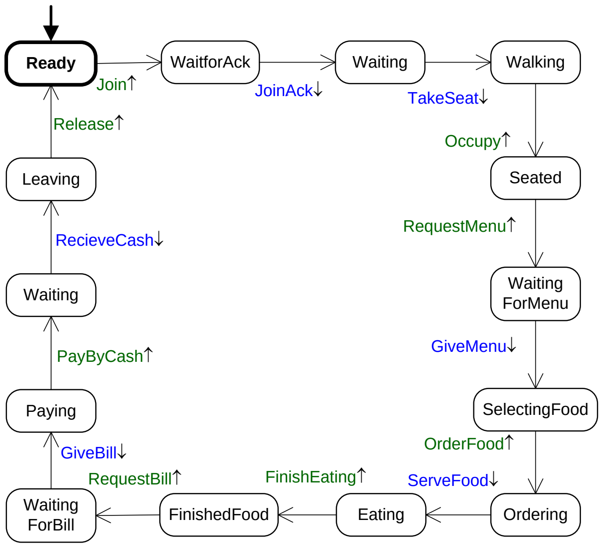

Figure 8 Customer Statemachine

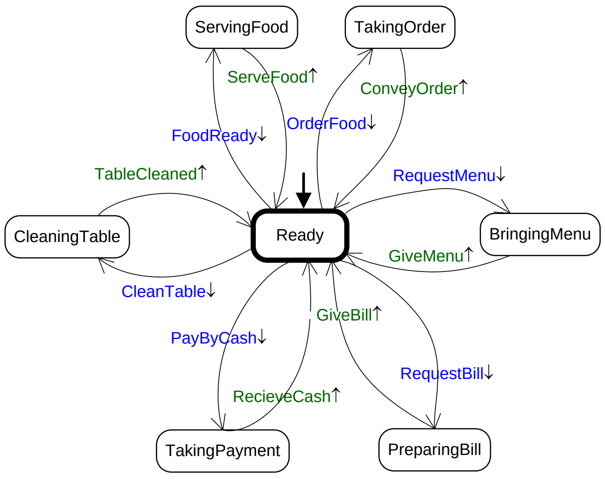

Figure 9 Waiter Statemachine

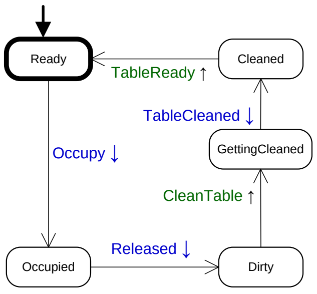

Figure 10 Table Statemachine

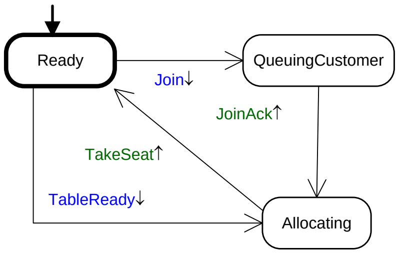

Figure 11 Queue Statemachine

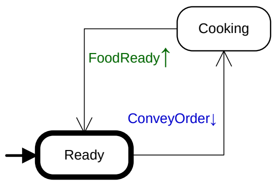

Figure 12 Chef Statemachine

Figure 8-12 represent the individual statemachines of each entity involved in the scenario. Here ↑ arrow means Sent Event where as ↓represents a received event.

In step 2 the corresponding data was then injected to Rule Validation Module, which checked all the three rules on the five statemachines. Since each statemachine contains valid exit actions, so they all passed Rule 1. Then all the Receive actions were compared to their respective senders and as they exist, so they also passed Rule 2. For Rule 3, we manually assigned Customer's **Leaving** state to be the final state of the scenario, as when Customer leaves the restaurant, we can say that the scenario is successful. When all three rules were passed the BOM was qualified to be transformed to SCXML documents. In step 3 five SCXML documents were generated, each representing the corresponding statemachine of the components.

In step 4, each SCXML document was then executed in our runtime environment. When the threads were initialized, each statemachine was reset to its initial state. Then they identified their next action. The first thread which was responsible to put() an action in the Event controller was **Customer** and the action was **Join**. (See pattern of interplay in Figure 7). So the customer proceeds to its next state by firing **Join** in its internal statemachine and place that action in the event controller. This simulated the sending of an action. This sent action was expected only by the **Queue** statemachine, so when the **Queue** got a chance to synchronize get(), it picked the event fired the event in its internal statemachine and then proceeded to the next state. This simulated the receiving of an event. This is how all the statemachine exchange the actions and proceed to their next states. In each transaction of a state, we are comparing if the current state is a Finial state or not. In the latter case we continue the execution and in the former case we terminate the loop and a message is printed out by the event logger which shows that the BOM statemachine were matched.

4.2 SCENARIO B:

This scenario is similar to the previous one, but the only difference is that we have introduced a different waiter component. The behavior of this waiter is such that he takes order from a customer and then waits for the customer to pay. Only when the customer has paid, he serves food. Now this waiter component will pass the static matching phase because it fulfills all the requirements. However during the dynamic execution, there will be a deadlock because the customer orders food and wait for it whereas the waiter expects the customer to pay before he eats his food. When the execution will reach the state **TakingOrder** the waiter will give bill to the customer and wait for the payment. But on the other hand the customer will wait for the waiter to ServeFood (See figure 8), so there will be a deadlock. Thus the statemachines do not match. Figure 13 represents the statemachine of the modified waiter

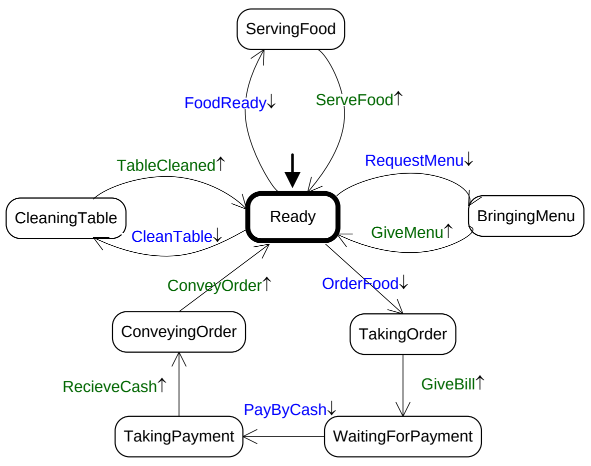

Figure 13 Modified Waiter Statemachine

4 Conclusions and future work

In this paper, we have defined statemachine matching and proposed a process that includes a static and a dynamic matching phase. In the static phase, we apply a set of rules to validate the structure of statemachines. The purpose of static matching is to detect possible structural problems in the participating statemachines. If a statemachine fulfills these rules, we transform it to a standard W3C SCXML format. In the second phase, we execute all the participating statemachines in the runtime execution environment that we have built on top of SCXML. The purpose of dynamic matching is to execute statemachines at an abstract level and detect possible deadlock. If the execution terminates successfully we conclude that the statemachines match each other. It should be noted that the matching of an abstract model is a necessary but not a sufficient condition for BOM composition. We have also discussed a case study to support our proposed method.

This matching approach will execute the statemachines in our runtime environment, help us to analyze the behavioral composition of BOMs and detect possible deadlock in the composed model.

This work has several limitations. First, it considers deterministic state machines only. Second, for simplicity, we have only considered single instance execution of each component participating in the composition i.e., in our case study we have only used one instance of customer, waiter, table, queue and chef. However in future we will extend our solution by introducing parallel execution of multiple instances of all participants in the composition and reevaluate our matching approach. Third, in our implementation we have not considered expression evaluation which is normally coupled with the transition rules of statemachine to model complex behaviors e.g., a customer may only *join* a queue when it is not full. So evaluating expressions along with the exit rules will be more realistic abstraction of the statemachines. There is a provision of different expression evaluation languages in SCXML framework which we intend to exploit to improve our matching technique.

5 References

[1] P. Gustavson, T. Chase, *"Using XML and BOMS to rapidly compose simulations and simulation environments"*, Proceedings of the 2004 Winter Simulation Conference.

[2] F. Moradi, R Ayani, S. Mokarizadeh, G. H. Shahmirzadi, G. Tan, *"A Rule-based Approach to Syntactic and Semantic Composition of BOMs"*, 11th IEEE Symposium on Distributed Simulation and Real-Time Applications, October, 2007.

[3] A. Gill, *Introduction to the theory of finite state-machines*, Book, McGraw Hill, 1961 New York.

[4] M. D. Petty, E. W. Weisel, R. R. Mielke, "*Overview of Theory of Composability"*, Virginia Modeling Analysis & Simulation Center, Old Dominion University, 2004.

[5] J.L. Martin, S. Mittal, B. Zeigler, J. Manuel, "*From UML Statecharts to DEVS State Machines using XML"*, IEEE/ACM conference on Multi-paradigm Modeling and Simulation, Nashville, September 2007.

[6] Yong Meng Teo , C. Szabo, *CODES: An Integrated Approach to Composable Modeling and Simulation*; 41st Annual Simulation Symposium, Ottawa Canada, 2008

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

[8] Base Object Model (BOM) Specification, SISO-STD-003-2006, Prepared by: SISO Base Object Model Product Development Group. http://www.boms.info/standards.htm (Last visited May, 2009).

[9] Guide for Base Object Model (BOM) Use and Implementation, *Simulation Interoperability Standard Organizations (SISO)*, 31 March 2006.

[10]State Chart XML (SCXML), A Statemachine Notation for Control Abstraction, http://www.w3.org/TR/scxml (Last visited May, 2009)

[11]State Chart XML (SCXML), http://commons.apache.org/scxml/ (Last visited May, 2009)