Generation of Oracle Components from Message Sequence Charts

Amit Kumar Mondal

August 27, 2016

1 Introduction

In this report, our objective is, given a component architecture and a Message Sequence Chart (MSC), to transform the component architecture by adding a so-called "flag output port" to it with the following expected behavior: during an execution, this output port should carry the value *true* (resp. $false$) at a given instant if the execution confirms (resp. contradicts) the specification of the MSC. This port should carry a special value *unknown* to know whether or not the MSC is satisfied.

To set the value of this port, some new oracle components need to be added to the component architecture: those components observe the communications of the existing components, and set the value of the output port accordingly.

For the component architecture, we will use the FOCUS semantics [1] where the execution of all components is synchronized by a global clock and every channel carries, at every instant, a value of its type or a special signal $\perp$ meaning "absence of signal".

2 Specialization of Syntax for the Problem

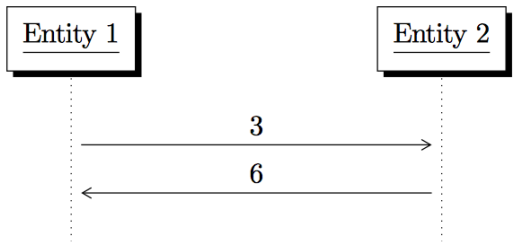
Before generating oracles for component architectures, we should define the *specifications* of such oracles: as mentioned in the introduction, these should set their output to *true* if the execution confirms that the component architecture satisfies the provided MSC. But what does it mean for a component architecture to "satisfy" an MSC? In this section, we investigate various instances of pairs of *MSC – component architecture* in order to characterize the semantics that we should give to the compatibility problem. We will also extend the language of MSC in case these investigations deem it to be necessary.

2.1 Introductory Example

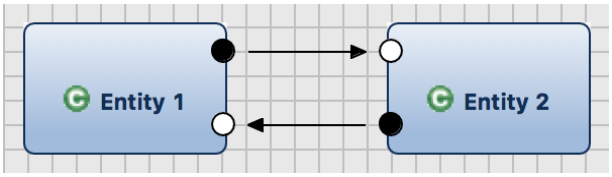
Consider the simple MSC and component architecture in Figure 1. Note that we can already draw a first consideration for the compatibility problem: every entity of the MSC shall have a corresponding component in the component architecture.

We now consider a few simulation traces of the component architecture and analyze whether these traces "satisfy" the MSC or not. Let us first consider the trace (Figure 2a) (Note that this is really a *trace* and not an MSC, i.e., it describes *all* the messages being sent between both entities).

According to the defined MSC (Figure 1a), Entity 1 expects message 6 after message 3 is received by Entity 2. As shown in the trace (Figure 2a), we can clearly see that Entity 1 receives message 6 and message 7 after it has sent message 3. This trace therefore proves that the component architecture indeed satisfies the MSC.

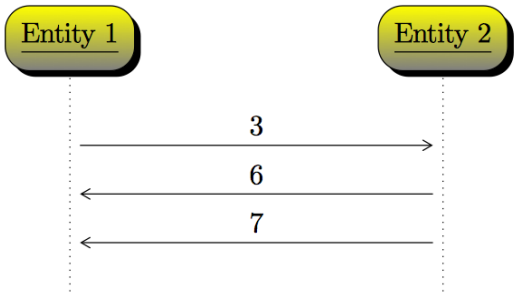


(a) MSC

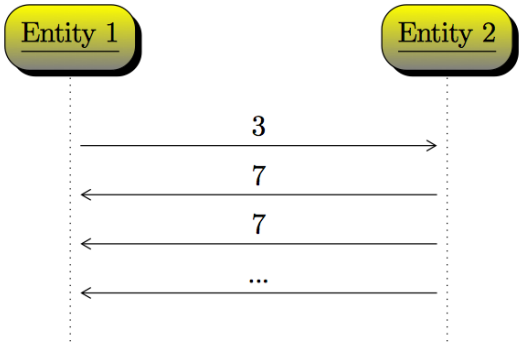


(b) Component Architecture

Figure 1: Message Passing in Component-based System

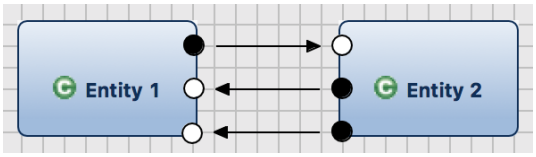


(a) Trace 1

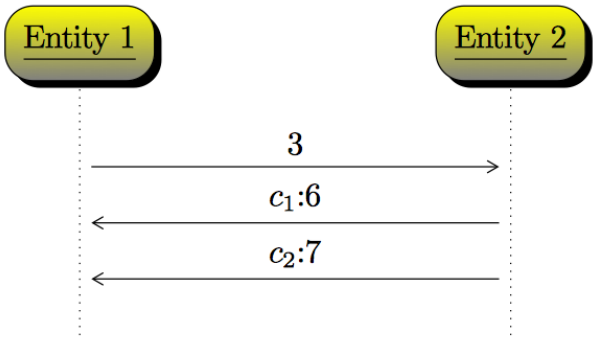


(b) Trace 2

Figure 2: Simulation Traces of defined MSC in Figure 1a



(a) Component Architecture



(b) Trace

Figure 3: Instance of Message Passing in Component-based System

Consider now the trace as shown in Figure 2b. According to the MSC, Entity 1 expects to receive message 6 from Entity 2 after it sends a request for message 3 to Entity 1. Therefore, this trace clearly shows that the component architecture does not satisfy the MSC.

2.2 Ambiguous Instance: Need to annotate MSCs

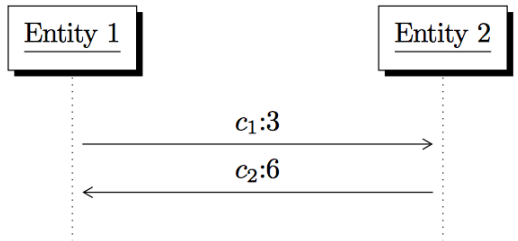
Consider the same MSC as before but the component architecture as shown in Figure 3a and the corresponding satisfying trace (Figure 3b), annotated with channels on which the messages are sent but how can we tell if this trace provides evidence that the MSC is satisfied? The original MSC does not indicate on which channel that message 6 is expected.

This shows that the compatibility problem requires that we extend MSCs by allowing annotating the messages with the channels on which they are expected to be observed. The first MSC becomes then for instance Figure 4a.

There is no possible ambiguity about the satisfaction of the previous trace. From now on, traces will be represented in a tabular fashion showing precisely the corresponding channels and time instants. For instance, the previous trace becomes Figure 4b where $\perp$ denotes absence of message.

2.3 Annotated MSCs

Therefore, in the context of the compatibility problem, it is important that MSCs are defined *with respect to a given component architecture.* For a given a component architecture, an *MSC* is defined as mappings between every entity of the MSC to a sub-component interface of the component architecture and every message of *msc* to a channel of the component architecture where a channel is a pair of two different ports of the same type.



(a) MSC

t	c_1	c_2	c_3
0	3	$\perp$	$\perp$
1	$\perp$	6	$\perp$

(b) Trace

Figure 4: (a) denotes annotated messages with communication channels and (b) denotes a tabular representation of a simulation trace

3 Specialization of Semantics for the Problem

Before defining the compatibility problem, we need to define the precise semantics of MSCs. Before being able to do so, we need to investigate some further potentially ambiguous instances of the problem.

3.1 Ambiguous Instance

Consider the same component architecture and its (now annotated) MSC but assume now that we observe the following trace:

t	c_1	c_2
0	3	$\perp$
1	$\perp$	$\perp$
2	$\perp$	4
3	$\perp$	6

Shall this trace be intuitively considered as satisfying the MSC or not? One might argue that yes, it does because after 3 has been observed on communication channel c_1, c_2 observes 6 at later instant of time. But one might as well disagree because they were expecting a 6 *and only a* 6 to happen. In the context of using MSCs to specify the behavior of the component, our experience is that users expect the latter – even though various semantics can of course be given to an MSC, as pointed out in [2] and [3]. Therefore we will restrict the semantics of MSCs in our context to enforce that a message has to be the *next one*.

Now we have all the necessary ingredients to define the compatibility problem.

3.2 Compatibility Problem

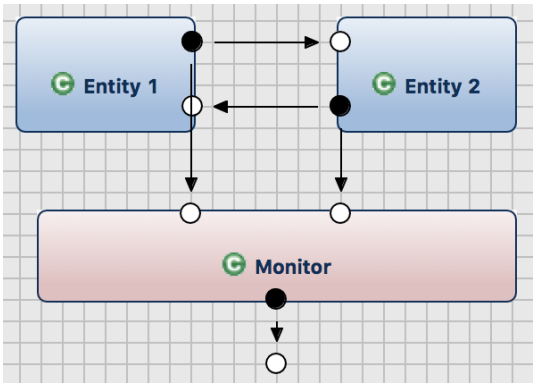
Generation of oracles for component architectures requires concrete specifications to be defined. These oracles must set their output to *true* if the simulation trace ensures the conformance of the component architecture with the specified MSC. But it is imperative to know what ensures such conformance of a component architecture with an MSC. To ensure the conformance, new oracle components are generated to observe the communications of the existing components.

4 Generation of Oracle Components

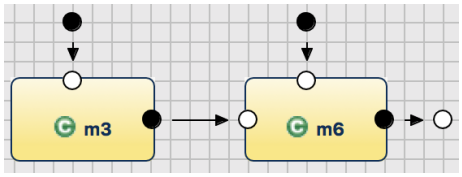
4.1 Supporting Example

Assume the component architecture (Figure 1b) and its corresponding MSC as shown in Figure 4a. Oracle component *Monitor* is generated as Figure 5a.

Monitor is accountable for monitoring output ports from the component architecture to verify conformance with the MSC specification. This is also composed of several oracle sub-component(s) and the number of sub-components varies depending on the message sequences defined in the MSC specification. *Monitor* comprises oracle sub-components as found in Figure 5b and every sub-component observes a specific message from the corresponding MSC.

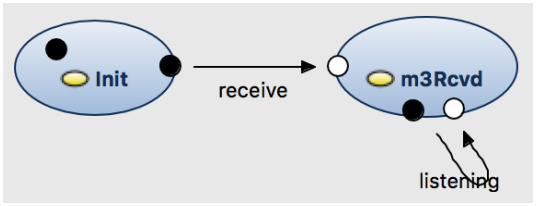


(a) Component Architecture with Monitor oracle

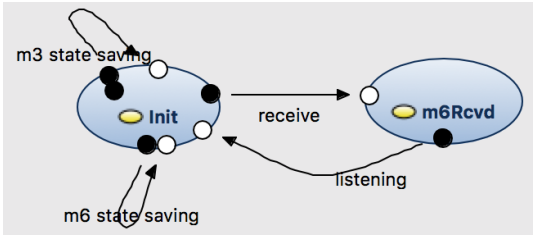


(b) Monitor Decomposition

Figure 5: Oracle Components



(a) m3 Automaton Specification



(b) m6 Automaton Specification

Figure 6: Behavioral Specifications

4.1.1 Behavioral Specifications

Oracle sub-components as shown in Figure 5b are composed of several automaton specifications to perform a predetermined sequence of actions depending on a sequence of messages defined in MSC.

The automaton specification of *m3* to perform observation of message 3 is defined in Figure 6a. This specification denotes observation of message 3 and its corresponding state changes. *init* denotes the initial state of the component. Transition *receive* is followed when the component observes message 3 and the current state (*init*) is changed to *m3Rcvd*, setting output to component *m4* to *true*. Transition annotated *listening* listens for further observations of message 3 thereon.

As stated previously, every oracle component is responsible for observation of a respective message from MSC. In that manner, *m6* is responsible for observations of message 6 (Figure 6b). Transition annotated *m3 state saving* saves the state of observation of message 3 in a respective state variable and similarly, transition annotated *m6 state saving* stores the state of observation of message 6. *receive* transition is followed after observations of both message 3 and 6 and the current state is rendered into *m6Rcvd*. Final observation output is also set to *true* thereafter. State variables as mentioned here are also reset after output has been set to true and transition *listening* is followed to return to its initial state to listen for further instances of both message 3 and 6.

Introduction of such oracle components ensures conformance of an MSC with its corresponding component architecture by observing communications of existing components. Accordingly, this also shows the real expressiveness with respect to the set of *simulation traces*.

4.2 Another Instance

Consider the defined component architecture in Figure 7 and assume the MSC in Figure 8a. The component architecture in Figure 8b incorporates the newly generated monitoring oracle (*Monitor*). *Monitor* comprises several oracle sub-components as in Figure 9 and as stated before, every sub-component observes a specific message from the corresponding MSC.

In accordance with the parallel operations defined in the MSC, oracle sub-components *m3* and *m4* are responsible for observing message sequences of 3 and 4 respectively. Similarly, *m5* and *m6* observe message 5 and 6 respectively. Due to the presence of

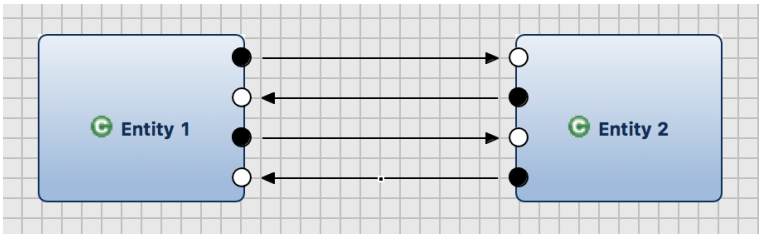
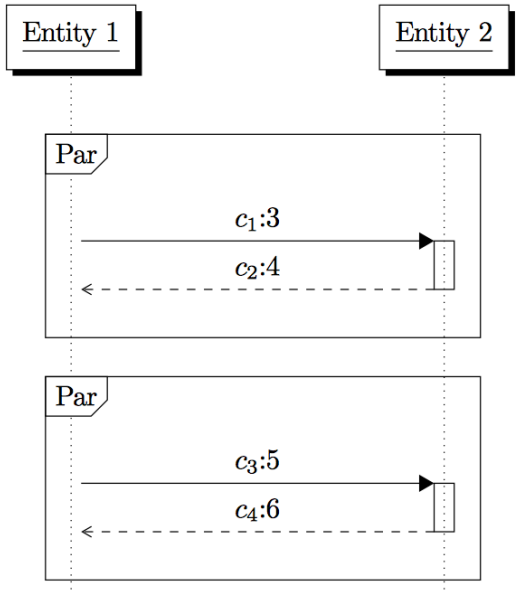
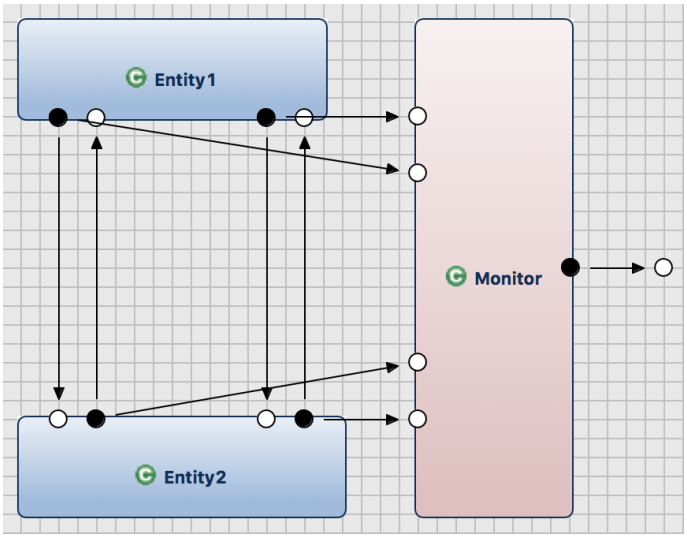


Figure 7: Parallel Message Passing in Component-based System



(a) MSC



(b) Component Architecture with Oracle Component

Figure 8: Instance of top-level oracle component generation, satisfying the MSC in Figure a

parallel operations, *Aggregator* is introduced to observe completion of both parallel operations to substantiate conformance.

4.2.1 Behavioral Specifications

Automaton specification of *m3* observes message 3 as shown in Figure 10a. The *m3* automaton specification denotes observation of message 3 and its corresponding state changes. The component changes its initial state (*init*) to *m3Rcvd* after observation of message 3, setting output to component *m4* to *true*. *listening for m3* is followed for further observations of message 3. Similarly, *m4* observes message 4 and its corresponding automaton specification is shown in Figure 10b.

The *m3 state saving* transition stores the state of observation of message 3 in a respective state variable and similarly, transition annotated *m4 state saving* stores the state of the observation of message 4. *receive* is followed after observations of both messages and the current state is switched to *m4Rcvd*. Output to *Aggregator* is also set to *true* thereafter. State variables are also reset after output to *Aggregator* is set and transition *listening* is followed to return to its initial state (*init*) to listen for further instances of both message 3 and 4.

In a similar fashion, *m5* observes message 5 and its automaton specification is shown in Figure 11a. *m6* sub-component is responsible for monitoring observations of message 6 and its automaton specification (Figure 11b) is in a way very similar to automaton specification of *m4*.

On the other hand, *Aggregator* monitors executions of both the parallel operations. Due to interleaving, any of the parallel operations occur at first and the other one thereafter. Figure 12 shows the automaton specification of *Aggregator* oracle component. *Aggregator* stores the states of the interleaving parallel operations as it completes and monitors completion of both operations to ensure conformity of the MSC. *Aggregator* changes its initial state to *dataRcvd* if it confirms successful executions of both parallel operations.

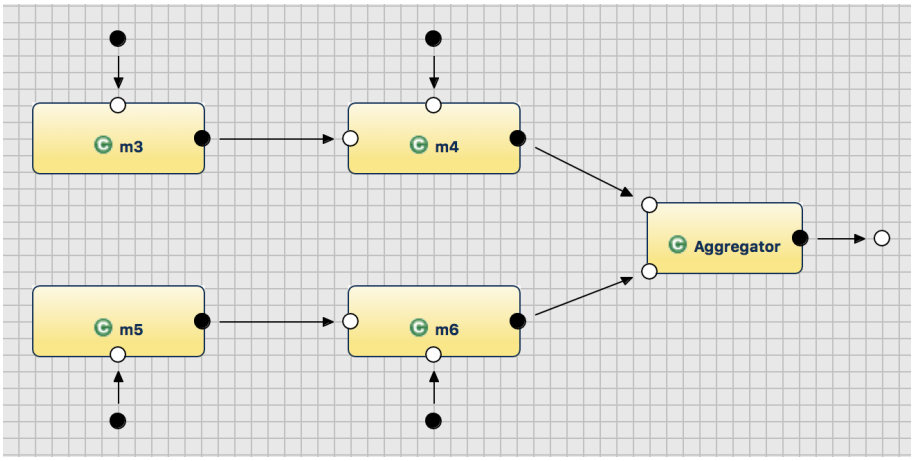
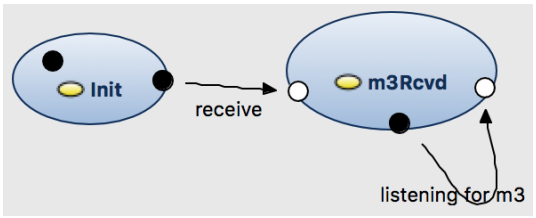
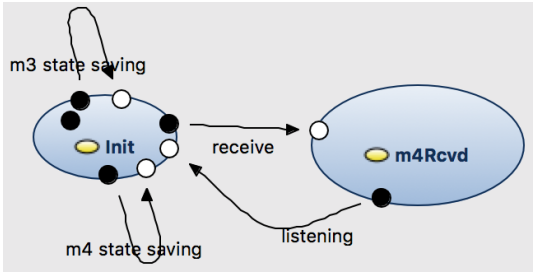


Figure 9: Sub-level Oracle Components

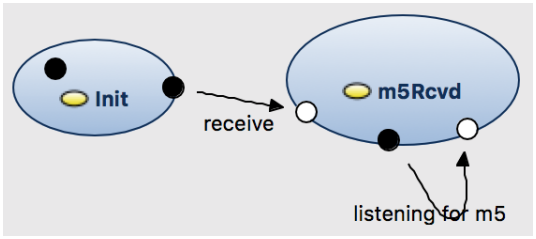


(a) m3 Automaton Specification

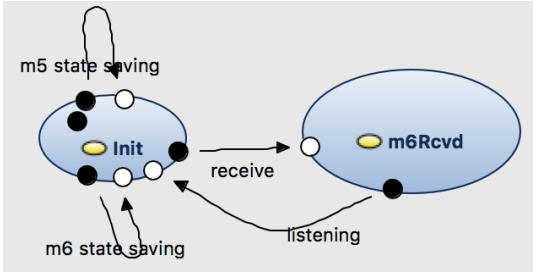


(b) m4 Automaton Specification

Figure 10: Behavioral Specifications



(a) m5 Automaton Specification



(b) m6 Automaton Specification

Figure 11: Behavioral Specifications

4.3 Generalization

So far, we have seen two instances of oracle components generation. The idea behind such generation is to define a semantic behavior. The supporting example in 4.1 shows the sequential message passing in a component-based system for which oracle components are generated. Every oracle component monitors a respective message from the specified MSC.

On the other hand, another instance as shown in 4.2 incorporates parallel message passing. Similar to the previous example, every oracle component is responsible for observation of a respective message from the corresponding MSC. Due to the uncertainty of the order of parallel operations completion, a dedicated oracle component, namely *Aggregator* is generated which monitors completion of the defined parallel operations. It sets the final output port to true if the parallel operations are successfully executed or otherwise false.

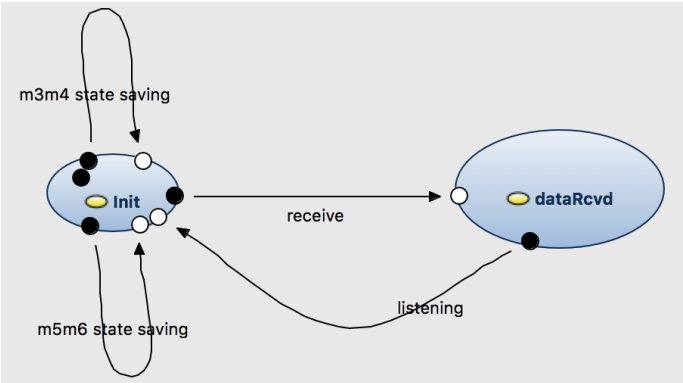
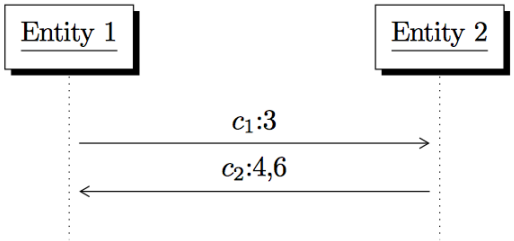


Figure 12: Aggregator Oracle Component Automaton Specification

t	c_1	c_2
0	3	$\perp$
1	$\perp$	$\perp$
2	$\perp$	4
3	$\perp$	6

(a) Trace



(b) MSC

Figure 13: Further instance of MSC and satisfiability check with corresponding simulation trace

5 Extension: Use of Constraints on Messages

It is not yet possible to impose restrictions on additional messages that can be potentially exchanged between a given pair of messages. In this section, we have extended the functionality of MSC by introducing new notion of constraint-based semantics.

5.1 Motivating Example

Given the component architecture (Figure 1b) and its corresponding MSC as shown in Figure 4a, let us try to check the conformity of the trace as defined in Figure 13a. Here c_2 encounters a 4 before 6 and hence it invalidates the conformity. According to the notion of our constraint-based semantics, the default behavior allows only the absence of messages between a given pair of messages.

Let us consider the aforementioned trace again. According to our semantics, to ensure the conformity, the MSC should be defined as Figure 13b. Now, this MSC clearly denotes that c_2 can encounter either absence of any message or a 4 and/or a 6.

5.2 Extension of MSCs with Constraints

In this section, we extend our constraint-based semantics to clearly define the MSC specification in Figure 14 (with the component architecture in Figure 1b).

Consider the trace as shown in Figure 15a. The MSC denotes that after a request message 2, c_2 must not encounter the presence of any other message but multiple instances of 5, 7 and/or 9. This clearly invalidates the trace as c_2 encounters a 8.

Let us assume another trace (Figure 15b) for the previous MSC. As c_2 cannot encounter any other message but multiple instances of 5, 7 and/or 9, we can ensure the validity of this trace.

6 Related Work

The main goal of our work is to propose a way to solve the problem of accurately capturing correctness requirements that have to be verified. For this purpose, we have analyzed existing solutions in order to figure out graphical notations and associated semantics that might be appropriate to extend. For this reason, we have started with analyzing the tools that are commonly used in industries

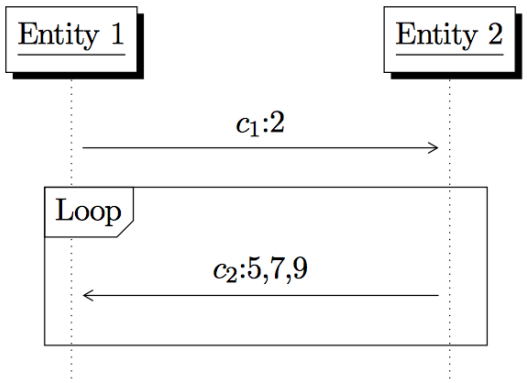


Figure 14: MSC denoting advanced usage of constraints

t	c_1	c_2
0	2	$\perp$
1	$\perp$	8
2	$\perp$	7
3	$\perp$	5

(a) Trace 1

t	c_1	c_2
0	2	$\perp$
1	$\perp$	$\perp$
2	$\perp$	7
3	$\perp$	$\perp$
4	$\perp$	7
5	$\perp$	9
6	$\perp$	5

(b) Trace 2

Figure 15: Further instances of traces for the specified MSC in Figure 14

for specifying component-based systems that interact by message passing. Visual formalisms are scenario-based modeling that are commonly and extensively used within industrial software development practice are *Message Sequence Chart (MSC)*.

In addition, we have analyzed the generic *Message Sequence Chart (MSC)* [4] [5] and also an extended version of the Message Sequence Chart as defined in [2]. A scenario-based visual language called *Property Sequence Chart (PSC)* has been presented that to some extent, fixes the drudgery of model verification using Temporal Logics. It does extend a subset of UML 2.0 Sequence Diagrams. This paper also includes both denotational and operational semantics. The operational semantics is obtained via translation into Büchi Automata. Expressiveness of PSC has also been validated with respect to several *Property Sequence Patterns*. This paper focuses on specifying the execution sequences of a system in terms of message passing by defining an ordering relation among the system messages. As stated earlier, the paper provides a translation algorithm that gets a PSC as input and returns the corresponding Büchi automata. In PSC, different types of messages are provided (*Regular, Required, Fail messages*) subjected to constraint. The translation rule is applied to directly derive the Büchi Automaton corresponding to a single message as found in PSC.

Furthermore, we have also analyzed another solution as presented in [3]. This solution incorporates a seamless integration of MSCs into the system development process. More precisely, it presents a way to translate scenario-based system requirements using MSCs to state-based descriptive techniques like automaton specifications. Due to a seamless integration of MSCs and State Automatons, an unified language has been presented at the semantic level. The unified semantics are presented by means of relational stream semantics. The stream semantics of one component, specified either as MSC or State Automaton reflects its input/output behavior. Dealing with such semantics leverages the opportunity to treat abstract specifications and concrete descriptions. Even though the translation as presented in this solution mainly focused on time synchronous communication, it does not actually depend on

the underlying communication mechanism as it separates every automaton to each component corresponds to the MSC.

We have also examined another extended version of Message Sequence Chart - *Live Sequence Chart* [6]. It presented different shortcomings of Message Sequence Charts and also a novel way to engineer reactive distributed systems. It expanded a serious effort *defining* and *formalizing* modeling languages and relationships between models, hence clearly stated what it means for a design model to *correctly implement* a set of requirements.

During our analysis, we have also looked into an extension of Live Sequence Chart [7] which endeavored to solve the problem of complexities involved in specifying formalisms for runtime verification.

Another probable solution to transform sequence diagrams to state machine has been presented in [8] and [9]. The papers explore how graph transformation based rules can be incorporated for verifying whether the MSC as an early model of the system satisfies a temporal requirement driven by an automation. [9] also introduced tailored transformations for sequence diagrams and a unique graphical operator that allows you to specify the matching and transformation of combined fragments with an unknown number of operands.

In [10] and [11], generation of distributed runtime monitors for each component has also been presented. It makes use of *Aspect Oriented Programming (AOP)* techniques to inject the monitors into the implementation of the components. Thereby, it verifies the adherence of the distributed system interactions with the MSC model.

7 Conclusion and Future Work

In this report, we proposed an approach to verify the conformance of an MSC with a defined component architecture by generating oracle components. In addition, we have proposed a preliminary technique to specify constraint properties aimed at being simple, powerful and user-friendly. After we have examined Message Sequence Charts, we presented an extended notation of a selected subset of Message Sequence Chart operators. More precisely, our notation to define constraints describes both positive and negative scenarios.

As future work, we plan to introduce the formalisms of relevant preliminary definitions, constraint semantics and the extension of the compatibility problem to the MSC with these constraint semantics. We also plan to even validate the expressiveness of our notation of representing constraints with respect to several simulation traces. Furthermore, generation of oracles would be enhanced to incorporate our newly introduced notation for constraint.

Acknowledgement

I wish to express my sincere gratitude to my supervisor PD Dr.habil. Bernhard Schätz and advisor Dr. Vincent Aravantinos who gave me the opportunity to work on this topic under their supervision and guided me with expert advice and encouragement with a freedom to pursue own ideas as well. I am also very grateful for their constant support and valuable suggestions.

At last I would like to thank the research division of Software and Systems Engineering of fortiss GmbH and Technical University of Munich for providing me with the facilities necessary to carry out my work.

References

[1] Manfred Broy, Max Fuchs, Frank Dederichs, Claus Dendorfer, Thomas F. Gritzner, Rainer Weber "The Design of Distributed Systems - An Introduction to FOCUS"

[2] M. Autili, P. Inverardi, P. Pelliccione "Graphical Scenarios for Specifying Temporal Properties: an Automated Approach".

[3] Ingolf Krüger, Radu Grosu, Peter Scholz, Manfred Broy "From MSCs to Statecharts"

[4] B. Genest, A. Muscholl, D. Peled "Message Sequence Charts"

[5] Ekkart Rudolph, Jens Grabowski, Peter Graubmann "Tutorial on Message Sequence Charts"

[6] Y. Bontemps, P. Heymans, P. Schobbens "From Live Sequence Charts to State Machines and Back: A Guided Tour"

[7] Ming Chai, Bernd-Holger Schlingloff "Monitoring Systems with Extended Live Sequence Charts"

[8] Rajeev Alur, Mihalis Yannakakis "Model Checking of Message Sequence Charts"

[9] Roy Gronmo, Birger Moller-Pedersen "From UML 2 Sequence Diagrams to State Machines by Graph Transformation"

[10] Ingolf Krüger, Michael Meisinger, Massimiliano Menarini "Interaction-based Runtime Verification for Systems of Systems Integration"

[11] Ingolf Krüger, Gunny Lee, Michael Meisinger "Automating Software Architecture Exploration with M2Aspects"

[12] Jacques Klein, Franck Fleurey, Jean-Marc Jèzèquel "Weaving Multiple Aspects in Sequence Diagrams"