JOURNAL OF SOFTWARE ENGINEERING & INTELLIGENT SYSTEMS
ISSN 2518-8739
30 April 2020, Volume 5, Issue 1, JSEIS, CAOMEI Copyright © 2016-2020
www.jseis.org

EARLY SOFTWARE BEHAVIOUR MODELING METHODS: A REVIEW

AWAD ALI

Najran University, Saudi Arabia
Kassala University, Sudan
Email: awadali625@gmail.com

ABSTRACT

Early behaviour models of software systems that generated from requirements specifications have proven useful in early analysis and checking of the design correctness. A number of early behaviours modeling methods are available in the current decade. The most critical part of these methods a scenario description language which participates in dividing system into small parts called scenarios. The behaviour modeling method utilizes these scenarios as input and produce behaviour models in one of the state machine formalisms such as state chart. The complexity of the state machine is a consequence of the scenario description. This paper presents a discussion on scenario description languages. The objective is to discover which language is capable to provide most compact and concise scenario specification.

Keywords: behaviour modeling; scenarios; requirements specification; MSC; LSC;

Academic Editor: Ashraf Osman, Alzaiem Alazhari University, Sudan.
Published: 30 April, 2020
Funding: Authors have no funds.
Article Type: Review Article

1. INTRODUCTION

Basic elements of early behaviour modeling are a scenario description language and a behaviour synthesis method. A scenario language provides formalism that enable describing the system partially in a form that can be processed by the synthesis method. The synthesis method utilizes these scenarios as input and produce behaviour models in one of the state machine formalisms such as labelled transition system (LTS) or state charts. In the behaviour modeling, most of the studies focus on the scenario language enhancement rather than the synthesis method, because it has the major impact in the scalability of behaviour modeling [1-3]. Scalability term refers to the ability of modeling large systems [4]. The synthesis method is just translating the scenario to state machine formalism without affecting the number of scenarios reduction which in turn leads to the scalability enhancement. However, the synthesis method has to be able to deal with the language notation, which may be complex [5]. As this research work addresses the modeling at the early design stage, this work concentrates more on the behaviour modeling methods that were focused use the scenario description languages as main input for behviour modeling.

Historically, several scenario description languages such as UML-SDs [6] and MSCs [7] have been used by the early behaviour modeling methods. As reported in the current literature, motivated by the target of enhancing scalability of the behaviour modeling, more expressive scenario description language is required. The expressive language helps in producing concise and compact scenarios, which leads to a smaller number of scenarios for system description. In this section, a historical background on scenario description languages is provided, and then an extensive discussion about existing triggered scenario-based behaviour modeling methods is given. Triggered scenario languages are a popular scenario description language use today in behaviour modeling methods to improve scalability.

2. A HISTORICAL BACKGROUND OF SCENARIO DESCRIPTION LANGUAGES

A scenario is a description of how the elements of the system interacting to perform functionality. The importance of scenario documentation has motivated researchers to develop many scenario notations, as basic parts of any scenario description language. This section of this paper starts with sequence charts as common graphical notations for describing scenarios. Most of scenario languages, such as MSCs, UML-SDs, and LSCs are adopting sequence charts.

As shown in Figure 1 below sequence charts define a finite interaction between numbers of components' instances. Every vertical arrow in the sequence chart is named instance (or lifeline). Messages between component instances that refer to the interactions are represented by horizontal arrows. Interactions between components can be synchronous or asynchronous communication. In synchronous communication, message represents an

Corresponding Author: Awad Ali
Email Address: awadali625@gmail.com

instantaneous event on the both communicated instances. While in asynchronous communication, message represents two instantaneous events, sending event pertaining to the arrow source and the receiving event pertaining to the arrow destination. Figure 1 shows a basic syntax of sequence chart that describe a synchronous communication, and as shown no message arrows cross each other.

Most of the existing scenario description languages [1-3, 6, 7] were derived from the sequence chart shown in Figure 1. These languages were derived by adopting the syntax of sequence chart which is named labeled partial orders (LPO) and developing additional constructs. The LPO syntax is defined below in Definition 1. Example of these constructs are the combined fragments that were introduced in MSC and modified in UML 2 to enable description of complex scenarios [8]. The fragments comprise many constructs such as: alt constructs which is used to specify branches execution; par constructs, used to specify parallel behaviours and loop constructs that defines looping behaviour.

Due to the common adoption of sequence chart, its syntax was used in the definition of most scenario languages'[9], including the language defined in our previous research work [10].

Definition 1 (Labeled Partial Order): A labeled Partial Order is a tuple ($L, \sum, \leq_i, \lambda$) where: L is a finite set of locations. $\sum$ is event alphabet. $\leq_i \subseteq L \times L$ is a partial order relation over L corresponding to the top-down ordering of L, if l , l' , and l'' are locations $\in \leq_i$, then $\leq_i$ is: reflexive (e.g. either $l \leq l'$ or $l' \leq l$) and transitive (e.g. if $l \leq l'$ and $l' \leq l''$, then $l \leq l''$). And $\lambda : L \rightarrow \sum$ is a labeling function that defines the event following a location.

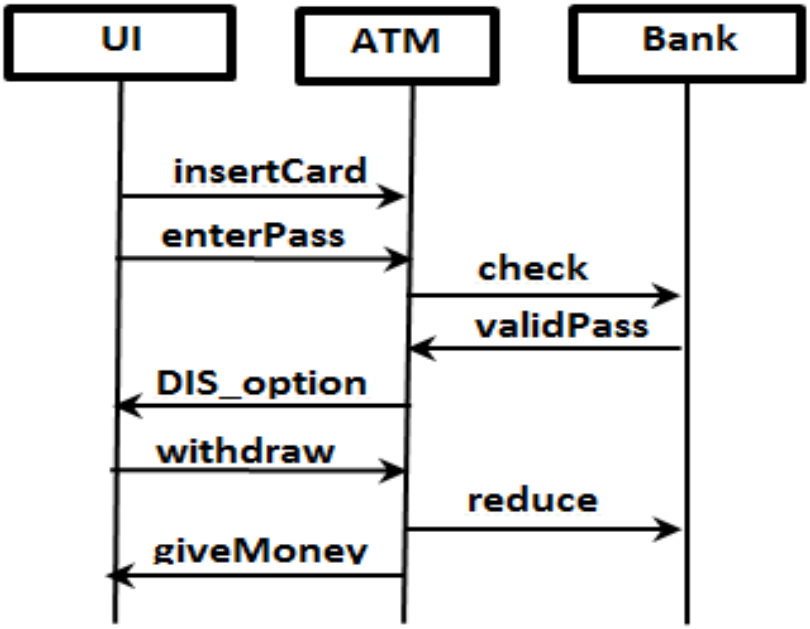

Figure. 1 Sequence chart

a) Message sequence charts

Message sequence charts (MSCs) are diagrams adopting the previous syntax of the sequence chart to represent scenarios. They have been standardized by the ITU [7] and also adopted in UML Interaction Diagrams. A Message Sequence Chart displays various component instances involved in the scenario as "lifelines", shown as vertical lines labeled by the component name.

For example, the MSC of Figure 2, showing an ATM system, describes a scenario involving three component instances: a user interface (UI), an ATM and a bank component. These components interact by sending messages to each other, shown as arrows. The scenario of Figure 2 reads naturally as "when the user inserts a card and enters a password, the ATM asks the Bank component to check the password. The Bank checks the password and may inform the ATM that the entered password is valid. Then the ATM display options to the user." An MSC describes a partial order between events, according to the following rule: an event may occur if all events higher up on the same lifeline already happened. This presents concurrency between events. For instance, in Figure 2, events wait and checking are not ordered and may occur in any order.

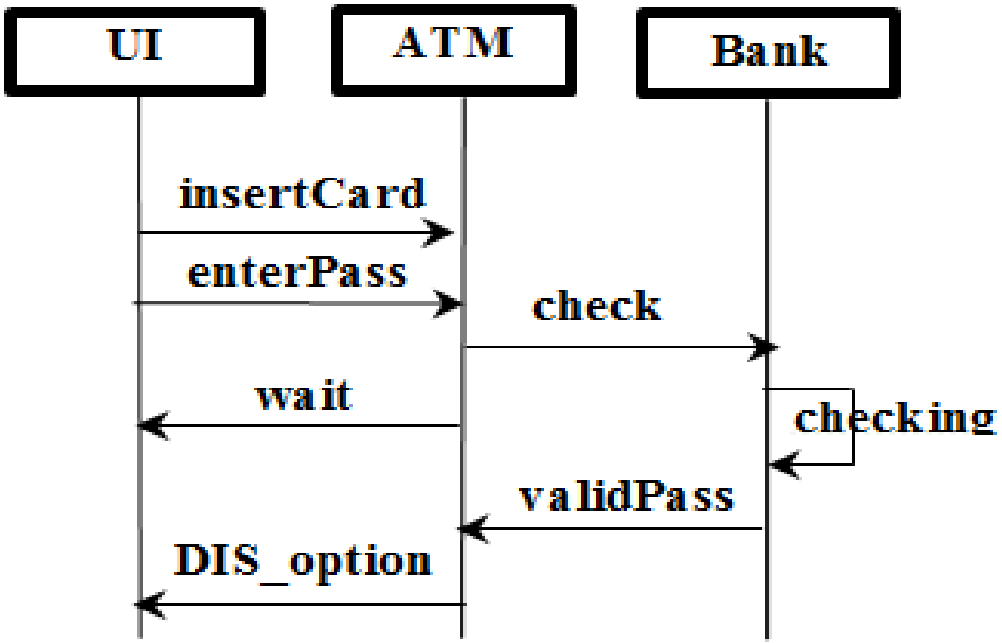

Figure. 2 Message sequence chart

Complex scenarios can be described in MSC by using set of composition operators such as concatenation, choice, unbounded loop and interleaving. There are two ways to use these operators: combined fragments and High-Level MSCs (hMSCs). In the former, boxes containing MSCs are included in the MSC. Later, a graph whose nodes contain MSCs presents all possible sequences of MSCs.

Though the graphical representation of the MSCs is much popular, still five shortcomings have been found in case MSCs is being used as input for early behaviour synthesis methods [11]:

- The semantics of MSCs are weak and aims at describing execution samples only. This is very different from specifying the expected behaviour of the future system.
- Composition of MSCs causes an enormous amount of extra work to be carried out by the software engineer. This is because the semantics of MSCs are silent about the meaning of a set of scenarios, i.e. how they impose requirements on the future system.
- MSCs miss syntactic constructs to express the scope of a scenario, i.e. whether events not appearing in this scenario can occur at will or are forbidden by their mere absence.
- MSCs do not distinguish between events that trigger the scenario and events that occur in response to this activation, even though this is a usual implicit distinction in scenarios.
- MSCs offer no syntactic means to distinguish between universal rules and examples.

b) Live sequence charts

Live sequence charts (LSCs) language is introduced to tackle the shortcoming of UML sequence diagrams and MSCs by adding liveness. LSC describes the scenarios of the system by two types of charts: an existential (eLSC) and a universal (uLSC). eLSC defines a non-triggered scenario which is more like MSC and SD [12]. A scenario in eLSC is defined as an example of system behaviour, and must be satisfied by at least one system run. While a scenario described by the uLSC is defines a rule that expected to satisfy by all system behaviour. Thus, each conditional behaviour in uLSC is encapsulated as action and reaction, using a pre-chart as a trigger and main chart as response to that trigger: once the pre-chart occurs, the main chart must occur. The reason behind using of two charts in LSC is related to the stage of requirements elicitation. In this stage there is a progressive movement from existential statements that represents examples and use-cases, to universal statements that represent final confirmation of the requirements, which are documented as declarative properties [12].

An example of a universal statement is “if the user inserts a valid card into the ATM, and then enters invalid password, then she/he must see password incorrect message”. Figure 3 shows description of such statement using universal chart of LSCs language.

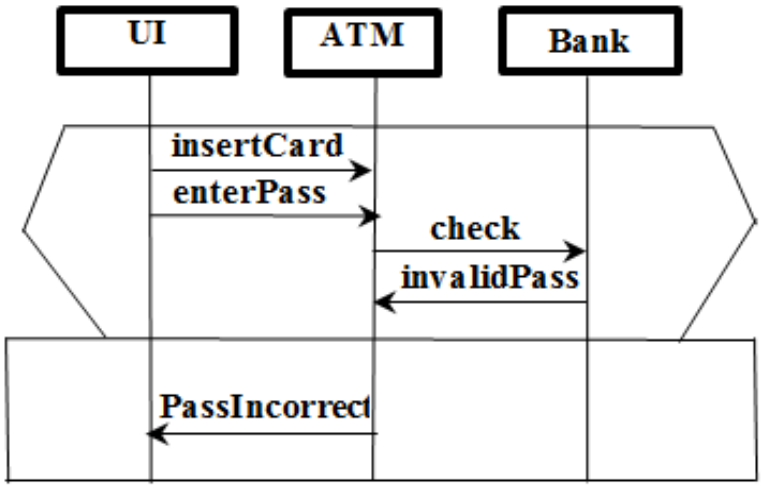

Figure. 3 LSCs chart

c) Triggered scenario languages

Existential statements depict by LSC language in a chart same like Figure 4 which describes the scenario: "if the user of ATM inserts a valid card into the ATM, followed by valid password, then may see ATM options and then may complete operation of withdraw successfully". The introduction of universal scenarios in LSC has inspired to a new type of scenario languages called triggered scenario languages. Main difference between these new languages and the other scenario languages is the description of scenarios in two different forms of statements that are existential and universal.

Triggered scenario language (TSs) can be defined as a scenario language that enables express reactive behaviours of system with clear distinction between existential, branching, and universal events [2]. A number of triggered scenario languages have been developed as an extension to MSCs, while others as an extension to LSCs. Unlike LSC existential statements in triggered languages described using a triggered formalism. Figure 4 shows an example of existential triggered scenario representation (this scenario has been depicted previously in Figure 1).

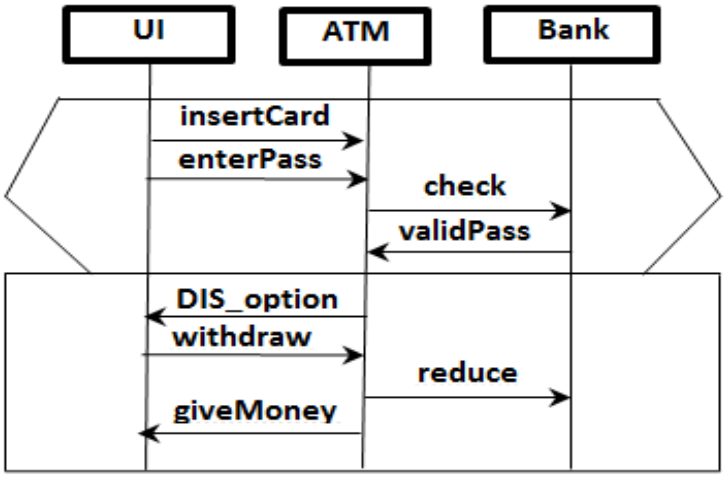

Figure. 4 Example of existential triggered scenarios

Although there are many scenario language extended from MSC such as Hybrid Sequence Charts (HySCs) [13],the popularity of triggered language became very high and the evidence of that it extended from both MSC [14] and LSCs [1]. The main reason for the triggered language's popularity is, the first form of requirements specification is use cases, and the use cases are interpreted as existential events in conjunction with triggers, thus triggered scenarios with existential semantics provide a good fit with use case based formalisms [12]. Later after the requirements confirmation, scenarios need to be rewritten in stronger statements, known as universal statements. Triggered languages provide support for documenting these two types of statements [12]. As the scalability problem is planned to tackle by the use of triggered language, the subsection below discusses triggered scenario-based early behaviour modeling methods.

3. TRIGGERED SCENARIO-BASED BEHAVIOUS MODELING METHODS

Unlike classical behaviour modeling methods which use scenario description languages such as UML-SD and MSCs, new type of behaviour modeling methods named triggered scenarios-based modeling is presented in the studies [1, 2, 8, 12, 14-18]. These methods use triggered scenario languages which can differentiate between existential, branching, and universal actions. For example, part of these languages use two types of chart to distinct existential statements from the universal, while others define different kind of arrows to distinguish between events. To address scalability problem a number of triggered scenarios-based methods have been proposed in literature [1, 2, 12]. Practically, our proposed work in [10] supports part of these methods via proposing a new mechanism extending and adopting their assumptions.

A main group of triggered scenarios-based methods, for instance the works in [2, 14, 16], use triggered languages built upon extension of MSC, while a another group [1, 12, 18] extended LSC. Damm and Harel [15] introduced the concept of triggered scenario in LSC via describing the scenario by two charts eLSC and the uLSC. The uLSC contains a pre-chart and a main chart. A pre-chart defines an event sequence whose execution activates the main chart. For the methods that build based on MSC similar to the trigger of uLSC, where extended MSC with trigger and universal semantics interpretation [16]. In the same direction, the study in [14] was also extended MSC with triggers and universal interpretation. the triggers in this study were used to describe component behaviour while the triggers in the study in [16] capture abstract progress properties of the system. Most the methods of that build based on MSC define High-Level MSCs as global behaviour model to describe inter-scenario dependencies. Each scenario is displayed by basic-MSCs, where each node in the High-Level MSCs denotes to one basic MSC. However, the methods of LSC do not need global model to describe inter-scenario dependencies, each scenario semantically symbolizes to the following scenario. The use of High-Level MSCs as global behaviour model may lead to state space explosion problem. That can be happening exactly in case of larger software systems where the number of scenarios is large.

In the LSC's group, the study presented by Sibay et al. [12] suggested use of trigger with existential chart. This chart in Sibay's work is named existential LSC with pre-chart (epLSC). The epLSC scenario description language uses one type of chart to describe the scenarios of the system. The aim of this work is to enable the progressive move from existential statements, in the form of examples and use-cases, to universal statements as a last confirmation in the requirement elicitation stage. Therefore, this type of scenario description languages combines the universal and existential scenarios in one chart, which can reduce number of scenarios. Sibay et al. in the study presented in [1], extended their study presented in [12] (epLSC language) via adding uLSC to convert previous model from existential to universal statements using independent charts for the universal scenarios. This extended scenario language is named e&uTS [10]. However, the scenarios that represent the whole system behavior can be less in case of using s-TS scenario description language presented in [10].

Conditional scenario specification language (CSSL) [18] is an another type of triggered scenario language that has been emerged. Linear-time logic is used in CSSL to represent conditional existential statements. CSSL is introduced to support reasoning about heterogeneous requirements specification that holding universal and existential statements. In the study presented in [17], M. Autili et al. defined a new scenario language called property sequence chart (PSC). PSC language represents an extension to Interaction Sequence Diagrams of UML 2.0. PSC presented to describe system temporal properties in an accurate and correct manner. PSC characterized by the simplicity of use and concentration at the system level. Harel and Maoz [8] extended UML 2.0 sequence diagrams by adding the universal and existential semantic of LSC. However, as critics to the work presented by Harel and Maoz, scenarios number that produced by the language do not taken into account [2].

In a similar spirit to our work presented in [10], I. Krka and N. Medvidovic [2] proposed a method for behaviour modeling consist a triggered scenario language named component-aware triggered scenario language (caTS). The presented approach concentrates on revealing the component-level behaviour. The number of scenarios in caTS is less comparing to others scenario languages. The reason is the combination of existential and universal scenarios in one chart. caTS adopted MSC's basic notations and defined additional constructs for example to existential and universal event, branching event, and alternative event. However, proposing a scenario language uses fewer constructs adopted from UML, from the usability perspective it will be easier than caTS to system developers, because as it is evident that software developers would prefer use of UML (as familiar notations) [19]. Figure 5 shows classification of early behaviour modeling methods that summarizes the previous discussion.

4. DISCUSSION

Combined fragments which have been introduced in MSC and modified in UML 2, can be adapted into new scenario language to enable description of complex scenarios [8]. The fragments comprise powerful syntactic constructs. Examples of these constructs are: alt constructs which use to specify branches execution, par constructs

use to specify parallel behaviours and loop constructs that defines lopping behaviour. Although these constructs have highly improved the expressiveness of scenario languages still there is a lack in the capturing of inter-scenarios dependencies and avoidance of scenarios overlapping. However, coding of scenario's messages as variables can enrich the scenarios in order to solve this problem. As the behaviour synthesis method is a main concern in this research, scenarios and scenario description languages need to be evaluated from behaviour synthesis perspective. Hence, that the following points conclude the required features to make scenarios specification suitable for behaviour synthesis method.

- Pre/post conditions that define the system state before and after each message. These conditions which annotate each component instance independently are better to define the system state from the component perspective. This means state variables will limit to the instance scope. Such way of annotation avoids difficulties which may arise due to scenarios composition specially in the case of constructing component behaviour models. Commonly pre/post conditions are defined by the object constraint language OCL [20] or linear temporal logic [21].
- Message ordering mechanism. The mechanism can be more useful if it is integrated with pre/post conditions that annotate component instances.
- Conditions are classified in three categories: 1) local conditions which cover and annotate one instance. 2) Non-local condition which cover and annotates more than one instance. 3) Global condition covering all instances within the scenario.
- Messages are defined as variables in order to participate in the capturing of inter-scenarios dependencies and avoidance of scenarios overlapping. Two types of message should be defined clearly: trigger messages and branching messages. Trigger message determine starting point of the scenario, while the branching message with combined fragments' operators such as ***alt*** and ***loop*** determine exit points.

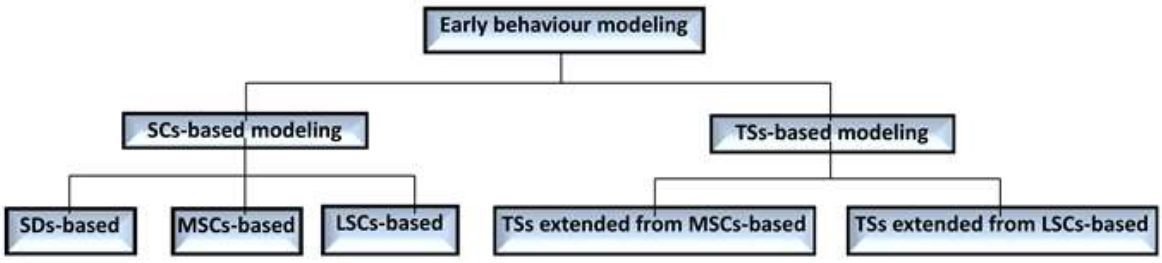

Figure. 5 Classification of early behaviour modeling

5. CONCLUSION

Behaviour modes that constructed based on system scenarios facilitates requirements elicitation and verification, test case automatic generation, code generation, and early analysis of software quality attributes. In this paper a detailed discussion on scenario description languages as an important part of behaviour modeling methods are presented. A discussion on the triggered scenario languages is given, as most of the scenario languages have been used to tackle problem of modeling large software systems.

ACKNOWLEDGEMENT

We are very grateful to the Deanship of Scientific Research at Najran University, Kingdom of Saudi Arabia, for their support of this research. We would also like to thank Embedded & Real-Time Software Engineering Laboratory (EReTSEL) members in Universiti Teknologi Malaysia (UTM) for their thoughtful and constructive feedback.

REFERENCES

1. Sibay, G.E., et al., *Synthesizing modal transition systems from triggered scenarios.* Software Engineering, IEEE Transactions on, 2013. **39**(7): p. 975-1001.
2. Krka, I. and N. Medvidovic. *Component-aware triggered scenarios*. in *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*. 2014: IEEE.
3. Whittle, J. and P.K. Jayaraman, *Synthesizing hierarchical state machines from expressive scenario descriptions.* ACM Transactions on Software Engineering and Methodology (TOSEM), 2010. **19**(3): p. 8.

4. Ali, A., D. Jawawi, and M.A. Isa, *Scalable Scenario Specifications to Synthesize Component-centric Behaviour Models.* International Journal of Software Engineering and Its Applications, 2015. **9**(9): p. 79-106.
5. Heinrich, R., et al. *Infrastructure for modeling and analyzing the quality of software architectures*. in *Proceedings of the 2nd International Workshop on Establishing a Community-Wide Infrastructure for Architecture-Based Software Engineering*. 2019: IEEE Press.
6. OMG, *UML: Unified modeling language superstructure specification v2.0, formal/05-07-04*. August 2005, OMG specification, OMG.
7. ITU, *Message sequence charts.* 2004.
8. Harel, D. and S. Maoz, *Assert and negate revisited: Modal semantics for UML sequence diagrams.* Software & Systems Modeling, 2008. **7**(2): p. 237-252.
9. Keller, K., et al. *A Comparative Analysis of ITU-MSC-Based Requirements Specification Approaches Used in the Automotive Industry*. in *International Conference on System Analysis and Modeling*. 2018: Springer.
10. Ali, A., D.N. Jawawi, and M.A. Isa, *Scalable Scenario Specifications to Synthesize Component-centric Behaviour Models.* International Journal of Software Engineering and Its Applications, 2015. **9**(9): p. 79-106.
11. Bontemps, Y., P. Heymans, and P. Schobbens, *From live sequence charts to state machines and back: A guided tour.* Software Engineering, IEEE Transactions on, 2005. **31**(12): p. 999-1014.
12. Sibay, G., S. Uchitel, and V. Braberman. *Existential live sequence charts revisited*. in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. 2008: IEEE.
13. Grosu, R., I. Kruger, and T. Stauner. *Hybrid sequence charts.* in *Object-Oriented Real-Time Distributed Computing, 2000.(ISORC 2000) Proceedings. Third IEEE International Symposium on*. 2000: IEEE.
14. Sengupta, B. and R. Cleaveland, *Triggered message sequence charts.* Software Engineering, IEEE Transactions on, 2006. **32**(8): p. 587-607.
15. Damm, W. and D. Harel, *LSCs: Breathing life into message sequence charts.* Formal methods in system design, 2001. **19**(1): p. 45-80.
16. Krüger, I.H., *Capturing overlapping, triggered, and preemptive collaborations using MSCs*, in *Fundamental Approaches to Software Engineering*. 2003, Springer. p. 387-402.
17. Autili, M., P. Inverardi, and P. Pelliccione, *Graphical scenarios for specifying temporal properties: an automated approach.* Automated Software Engineering, 2007. **14**(3): p. 293-340.
18. Ben-David, S., et al. *CSSL: a logic for specifying conditional scenarios*. in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011: ACM.
19. Becker, S., H. Koziolek, and R. Reussner, *The Palladio component model for model-driven performance prediction.* Journal of Systems and Software, 2009. **82**(1): p. 3-22.
20. Warmer, J.B. and A.G. Kleppe, *The object constraint language: getting your models ready for MDA*. 2003: Addison-Wesley Professional.
21. Giannakopoulou, D. and J. Magee. *Fluent model checking for event-based systems.* in *ACM SIGSOFT Software Engineering Notes*. 2003: ACM.

AUTHORS PROFILE