



**Wu, X., Li, C., Wang, X. and Yang, H. (2018) 'A creative approach to reducing ambiguity in scenario-based software architecture analysis', *International Journal of Automation and Computing*. doi: 10.1007/s11633-017-1102-y.**

The final publication is available at Springer via <http://doi.org/10.1007/s11633-017-1102-y>

## ResearchSPAce

<http://researchspace.bathspa.ac.uk/>

This pre-published version is made available in accordance with publisher policies.

Please cite only the published version using the reference above.

Your access and use of this document is based on your acceptance of the ResearchSPAce Metadata and Data Policies, as well as applicable law:-

<https://researchspace.bathspa.ac.uk/policies.html>

Unless you accept the terms of these Policies in full, you do not have permission to download this document.

This cover sheet may not be removed from the document.

Please scroll down to view the document.

# A Creative Approach to Reducing Ambiguity in Scenario-based Software Architecture Analysis

Xiwen Wu\*

\*Department of Computer Science  
and Engineering  
Shanghai Jiao Tong University  
Shanghai, China  
Email: jxnuwxw@gmail.com

Chen Li†

†Department of Computing  
Imperial College London  
London, United Kingdom  
Email: Chen.li1@imperial.ac.uk

Xuan Wang and Hongji Yang‡

‡Center for Creative Computing  
Bath Spa University  
Bath, United Kingdom  
Email: xuan.wang13,h.yang@bathspa.ac.uk

**Abstract**—In software engineering, a scenario describes an anticipated usage of a software system. As scenarios are useful to understand the requirements and functionalities of a software system, the scenario-based analysis is widely used in various tasks, especially in the design stage of software architectures. Although researchers have proposed various scenario-based approaches to analyse software architecture, there are still limitations in this research field, and a key limitation lies in that scenarios are typically not formally defined and thus may contain ambiguities. As these ambiguities may lead to defects, it is desirable to reduce them as many as possible. In order to reduce ambiguity in scenario-based software architecture analysis, this paper introduces a creative computing approach to scenario-based software requirements analysis. Our work expends this idea in three directions. Firstly, we extend an ADL-based language - Breeze/ADL to model the software architecture. Secondly, we use a creative rule - Combinational Rule (CR) to combine the vector clock algorithm for reducing the ambiguities in modeling scenarios. Then, another creative rule - Transformational Rule (TR) is employed to help to transform our Breeze/ADL model to a popular model - UML model. We implement our approach as a plugin of Breeze, and illustrate a running example of modeling a poetry to music system in our case study. Our results show the proposed creative approach is able to reduce ambiguities of the software architecture in practice.

**Keywords**—Creative Computing, Vector Clock Algorithm, Scenario-based Analysis of Software Architecture, Sequence Diagram, Breeze/ADL

## I. INTRODUCTION

For a given software system, its software architecture defines its structure, communication and interrelation among its components [1], and its scenarios describe its usages [2]. As scenarios are useful to improve the quality of an architecture [3], the scenario-based analysis of software architecture has been a hot research topic in recent years [3], [4], [5], [6]. Although this research direction is intensively studied, Sibertin-Blanc *et al.* [7] complain that even the basic diagram notation, sequence diagrams, may contain ambiguities, since existing work typically does not define scenarios formally. For example, a sequence diagram may show that component  $A$  receives two messages  $m_1$  and  $m_2$  from components  $B$  and  $C$  respectively, and component  $A$  delivers another message  $m_3$ . From the sequence diagram, it is tricky to determine the partial order between  $m_1$ ,  $m_2$  and  $m_3$ , since sequence diagrams do

not define the conditions of messages. Here, component  $A$  may need one message ( $m_1$  or  $m_2$ ) or both messages ( $m_1$  and  $m_2$ ) to deliver  $m_3$ , and different conditions lead to different message orders. The ambiguities in the design phase may lead to defects in the latter phase of software development. As it takes much effort to fix those defects in software, it is desirable to reduce the ambiguities as many as possible.

Thus, we introduce the creative computing combining with popular algorithm and modeling language into our approach. In software architecture, the architecture description language (ADL) is commonly used to formally define architectures. Although researchers [8], [9], [10], [11] have proposed various ADLs, it is still a blind spot to model scenarios with ADLs. In practice, programmers typically use the sequence diagram of Unified Modeling Language (UML) [12] to model scenarios. As the state of the art, UML is a general-purpose modeling language in the field of software engineering, and it provides a standard way to visualize the design of a system. Despite of its popularity, UML is not suitable for automated analysis (*e.g.*, verification and validation), since its constructs lack formal semantics. As a result, Pandey [1] criticizes that the informal UML can lead to ambiguities and inconsistencies. As far as the definition of scenarios is concerned, ADL has advantages over UML [1], since ADL presents a formal way to define scenarios, and thus allows automating architecture-level scenario-based analysis (*e.g.* [3], [4], [5], [6]). However, it is challenging to reduce such ambiguities in scenario-based software architecture analysis with ADLs:

**Challenge 1.** Although it is relatively easy to introduce new features for modeling scenarios, it is tricky to detect and reduce possible ambiguities in scenarios.

**Challenge 2.** To make our approach more general and compatible with existing works, we need rules to translation ADL into a general model. It is tricky to choose a popular modeling language and define such rules and to ensure the correctness during the translation.

To address the above challenges, in this paper, we present a creative computing approach to reducing ambiguity in scenario-based software architecture analysis and propose combinational rule and transformational rule. Based on the creative rules, we extend our previous tool, called Breeze. Our

extension supports the formalization of scenarios, the elimination of possible ambiguities, and the translation from ADLs to UMLs. This paper makes the following key contributions:

- Based on combinational rule, we propose an extended ADL, called Breeze/ADL, that introduces new features for modeling scenarios, and an algorithm to reduce possible ambiguities in scenarios. We leverage the combination rule to borrow ideas from the vector clock algorithm [13] that produces positive results in determining event orders in distributed systems.
- Based on transformational rule, we choose UML as our target model and define mapping relations between Breeze/ADL and the UML sequence diagram. The mapping relations allow us to translate Breeze/ADL into UMLs, which allows integrating our approach with existing tools. A user may design more accurate architectures with our tool, and then translate these architectures to UML to gain the benefits of existing industry tools.
- A plugin of Breeze that is implemented for our approach. The latest version of Breeze is now available at Github<sup>1</sup>. With the support of the extended Breeze, we conducted a case study on modeling an online shop. The results show the effectiveness of our approach.

The rest of the paper is organized as follows. Section II introduces the related work. Section III presents combinational rule and Breeze/ADL. Section V presents transformational rule and the mapping between Breeze/ADL and UML. Section VI presents a case study. Section VII concludes.

## II. RELATED WORK

**Creative Computing** With the rapid development of information technology, a great deal of novel computing emerges, such as Google search engine and Facebook, which enrich the human life much more convenience and colorful. This kind of computing could be considered as Creative Computing. The nature of creative computing, grammatically, focuses on the term 'creative'. It can be seen that the meaning of creativity is the core of creative computing.

Creativity can be defined as the ability to generate novel and valuable ideas [18]. If we look carefully at many examples of human creativity that surrounds us, we can see that there are three different ways in which creativity happens. Novel ideas may be produced by combination, by transformation, or by exploration [19].

Combinational Creativity means to combine familiar ideas to produce unfamiliar ones, through making associations between ideas [19]. Examples include many cases of poetic imagery, collage in visual art, and mimicry of cuckoo song in a classical symphony. Analogy is a form of combinational creativity that exploits shared conceptual structure and is widely used in science as well as art.

In transformational creativity, the space or style itself is transformed by altering (or dropping) one or more of its defining dimensions. As a result, ideas can now be generated

that simply could not have been generated before the change [19]. For instance, if all organic molecules are basically strings of carbon atoms, then benzene can't be a ring structure. In suggesting that this is indeed what benzene is, the chemist Friedrich von Kekule had to transform the constraint string (open curve) into that of ring (closed curve). This stylistic transformation made way for the entire space of aromatic chemistry, which chemists would explore [sic] for many years.

Exploratory creativity rests on some culturally accepted style of thinking, or conceptual space [19]. This may be a theory of chemical molecules, a style of painting or music, or a particular national cuisine. The space is defined (and constrained) by a set of generative rules. Usually, these rules are largely, or even wholly, implicit. Every structure produced by following them will fit the style concerned, just as any word string generated by English syntax will be a grammatically acceptable English sentence. In exploratory creativity, the person moves through the space, exploring it to find out what's there (including previously unvisited locations) and, in the most interesting cases, to discover both the potential and the limits of the space in question.

In this paper, we propose related rules based on combinational creativity and transformational creativity.

**Scenario-based Analysis of Software Architecture.** Kazman *et al.* [3] analyse relations between quality attributes and scenarios. Lassing *et al.* [14] work on a similar research problem, with an emphasis on the impacts of complex scenarios. Lung *et al.* [15] present an approach that estimates the reusability of software architectures through scenarios. Bose [4] models scenarios with finite-state machines and verifies their consistency. Tekinerdogan *et al.* [2] propose an approach that analyses the reliability of software architecture. Yacoub *et al.* [5] propose a reliability analysis method that is based on scenarios of component interactions. Rodrigues *et al.* [16] propose a reliability prediction approach that is based on scenarios. Cheung *et al.* [6] employ scenarios to predict the reliability of concurrent systems. Williams and Smith [17] present a scenario-based approach that estimates performance at architecture level. The above approaches typically do not formally define scenarios. Although Bose [4] uses finite-state machines to model scenarios, it is not helpful to detect many ambiguities in sequence diagrams (*e.g.*, the sample ambiguous sequence diagrams shown in Section I), since finite-state machines do not define conditions for transitions either. Our work complementing the preceding approaches, since it presents a formal language to define scenarios and an algorithm to reduce such ambiguities.

**Sequence Diagram.** In practice, the sequence diagram of UML is widely used to describe scenarios [20]. Figure 1 shows an example sequence diagram, and we present more details on UML sequence diagrams in Section V. Although it is popular, Pandey [1] complains that UML suffers the incapability of automated analysis, since it is informal. As a part of UML, sequence diagrams also have the same limitation. Furthermore, as shown by Sibertin-Blanc *et al.* [7], sequence diagrams may suffer from ambiguity. Our work complements the notation of

<sup>1</sup><https://github.com/BreezeCSA/Breeze>

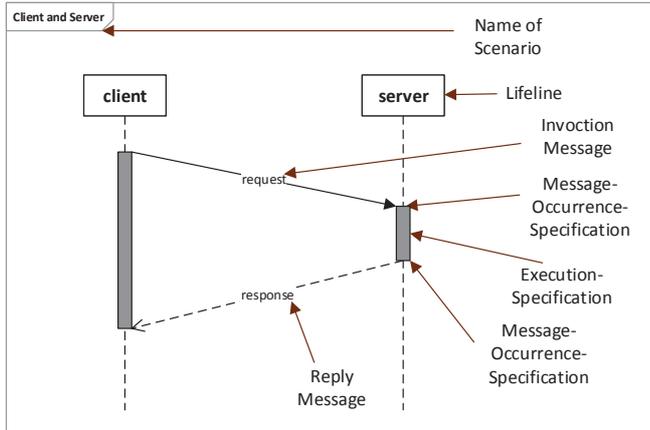


Fig. 1. An example of basic interaction

		Modeling Element	Description
Breeze.xsd	Style	Component template, Connector template, Interface template	Templates for elements.
		Style Constraints	Constraints for specific style.
		Reconfiguration Operations	Defining the changes
		State Options	Allowed state transformation
	Definition	Component, Connector, Interface,	Definitions of the elements according to the templates
	Configuration	Instances (Component, Connector and Connection)	The configuration of architecture instances.
	State	State Transformation Rules	The transformation rules for elements state

Fig. 2. The schema for Breeze/ADL

sequence diagrams, since our extended Breeze/ADL is formal and it allows our algorithm to reduce ambiguities in scenarios. **Breeze/ADL.** As the underlying ADL of our Breeze tool [8], [21], Breeze/ADL models software architecture in the formats of both XML and graphs. Figure 2 shows the schema file of the existing Breeze/ADL. To support modeling scenarios, our work in this paper extends the Breeze/ADL in Figure 2 with more elements. Furthermore, we propose an algorithm to reduce ambiguities of scenarios, and implement a translator from Breeze/ADL to UML sequence diagrams.

### III. SCENARIO MODELING IN BREEZE/ADL

The specification of UML<sup>2</sup> defines the concrete and abstract syntaxes of sequence diagrams. After inspecting elements of sequence diagrams, we find that the basic modeling elements in the *BasicInteractions* package of UML are quite useful. For example, most approaches in Section II use only these basic modeling elements. As a result, in this paper, we focus on these basic modeling elements.

In this section, we present new elements to model scenarios (Section III-1), and the vector clock algorithm to ensure that scenarios are defined without ambiguities (Section IV-B).

<sup>2</sup><http://www.omg.org/spec/UML/>

1) *Elements of Scenario Modeling:* From the viewpoint of software architecture, scenarios are interactions among the components of a software system. Here, components can be many, and their interactions may be in parallel. To define scenarios formally, we need to focus on the two aspects of scenarios such as the definition of scenarios and the interactions among their components. In particular, we introduce four additional elements to define a scenario, and the formal definition of a scenario is as follows:

**Definition 3.1** *Scenario* :=  $\{Description, ComponentList, MessageList, ParallelScenario\}$  represents a scenario, where:

- *Description* is a brief introduction of this scenario.
- *ComponentList* is a set of components that involve in a scenario  $S$ . Components in *ComponentList* follow the traditional definition of software architecture.
- *MessageList* denotes the messages among components, and each *Message* is defined in the next definition.
- *ParallelScenario* is a vector that defines possible active instances of scenarios. Here, in a vector  $\{a_1, a_2, \dots, a_m\}$ ,  $a_i$  denotes number of possible active instances of the  $i$ th scenario  $S_i$ , and  $m$  denotes the total number of scenarios. For example, if  $S_1, S_2, S_3$  and  $S_4$  are four scenarios and their vector is  $\{0, 0, 3, \infty\}$ , the first ‘0’ and the second ‘0’ denote that  $S_1$  and  $S_2$  cannot have active instance; ‘3’ denotes that  $S_3$  can have 3 active instances; and  $\infty$  denotes that  $S_4$  can have infinite active instances.

**Definition 3.2** *Message* :=  $\{SourceComponent, TargetComponent, VectorClock, Type\}$  represents a message, where:

- *SourceComponent* is a component that sends a message.
- *TargetComponent* is a component that receives a message.
- *VectorClock* is a vector clock that defines timestamps of messages.
- *Type* is the type of a message. We define two types of messages, *i.e.*, a request or a response.

In Figure 3, the left part shows the elements that are added to define scenarios, and the right part shows an example scenario that is defined in Breeze/ADL. In this example, the two components are *Client* and *Server*, and they are defined inside the *componentlist* label. The messages between the two components are defined inside the *messagelist* label, and the definition of a message includes its attributes such as *type*, *source*, and *target*.

### IV. COMBINATIONAL RULE FOR REDUCING THE AMBIGUITY

#### A. Combination Rule

*Combinational Rule* leads to improve the result and reduce ambiguity in scenario-based software architecture analysis. The rule is described as below:

**Definition 4.1.** A *Combinational Rule* (CR) is to combine different requirements modules ( $x_i$ ) based on variety weights ( $k_i$ ) (*i.e.*, probability of the selected modules) for inferring

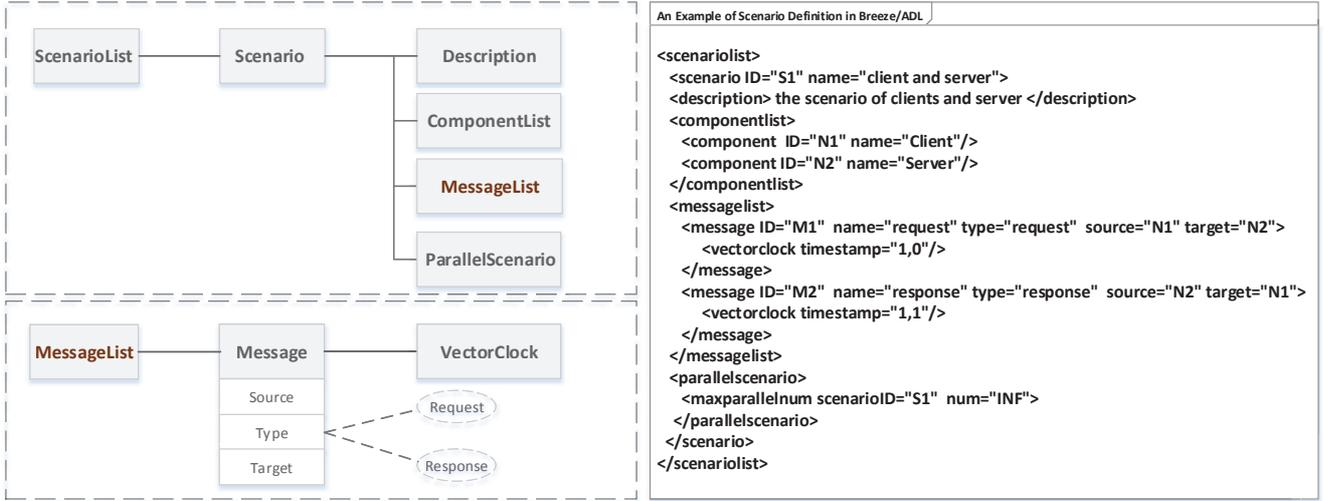


Fig. 3. The added elements for Breeze/ADL and an example

related requirements modules ( $y$ ) where  $i$  is the index of the module. The formal definition of  $CR$  is as follows:

$$k_0x_0 + k_1x_1 + \dots + k_nx_n \rightarrow y \quad (1)$$

where the  $n$  is the total number of the possible modules and the sum of the  $k_i$  should equal to 1, i.e.,

$$k_0 + k_1 + \dots + k_n = 1 \quad (2)$$

By using the above rule, we define  $x_0$  as Breeze/ADL model, and introduce the Vector Clock Algorithm (see next section) as  $x_1$ . Thus the equation for our approach can be written as:

$$k_0(\text{Breeze/ADLModel}) + k_1(\text{VectorClockAlgorithm}) \rightarrow (\text{Breeze/ADLModel})' \quad (3)$$

### B. Vector Clock Algorithm

This section presents the vector clock algorithm that reduces ambiguities. After we present the details of the algorithm, we next illustrate the algorithm with the sample ambiguous sequence diagram as shown in Section I.

The vector clock algorithm is a kind of logical time which is widely used in distributed systems to define the partial order of events [13]. A partial order set [22] consists of a set and a binary relation. In such a set, a pair of elements  $a \rightarrow b$  denotes that  $a$  precedes  $b$ , and the relation is called a partial order. In a partial order set, the relation between two elements ( $a$  and  $b$ ) falls into three categories,  $a \rightarrow b$ ,  $b \rightarrow a$ , or  $a \parallel b$ , where  $a \parallel b$  denotes that there is no order relation between them.

Algorithm 1 shows the detail of the vector clock algorithm. For a software architecture that contains  $n$  components, we use  $Arc = \{C_1, C_2 \dots C_n\}$  to denote the architecture, where  $C_i$  denotes the  $i$ th component of  $Arc$ . In the algorithm, a component is associated with an  $n$ -dimensional, non-negative vector  $vt_i[1..n]$ , where  $vt_i[i]$  records the timestamps of messages that are sent by  $C_i$ . The  $n$ -dimension vector of each component is initialized as a zero vector, and is updated according to the following rule. If component  $C_i$  is about to

send out a message  $M$ , our algorithm updates every dimension of its clock and chooses the message whose dimension is the biggest one happens before  $M$ . Here, our algorithm increases  $vt_i[i]$  by 1 before  $C_i$  sends the message. We assume that a component cannot send out more than one message at a time, so we separate messages by adding decimals. As the algorithm concisely define the partial order of message sending or receiving in scenarios, the ambiguity example in Section I will be eliminated.

In our approach, the timestamps have the basic property of isomorphism. We find that the timestamps of  $vt_i$  and  $vt_j$  follow the three relations:

- $vt_i \leq vt_j \Leftrightarrow \forall x \in [1, n] : \text{floor}(vt_i[x]) \leq \text{floor}(vt_j[x])$
- $vt_i < vt_j \Leftrightarrow vt_i \leq vt_j \text{ and } \exists x \in [1, n] : \text{floor}(vt_i[x]) < \text{floor}(vt_j[x])$
- $vt_i \parallel vt_j \Leftrightarrow \text{not}(vt_i < vt_j) \text{ and } \text{not}(vt_i > vt_j)$

Here,  $\text{floor}(x)$  rounds  $x$  downward. For example, if  $x = 1.6$ ,  $\text{floor}(x)$  returns 1.

As the relation  $\rightarrow$  defines partial orders, timestamps of messages follow the two properties:

$$m_1 \rightarrow m_2 \Leftrightarrow vt_i^{m_1} < vt_j^{m_2} \quad (4)$$

$$m_1 \parallel m_2 \Leftrightarrow vt_i^{m_1} \parallel vt_j^{m_2} \quad (5)$$

Here,  $m_1$  is a message sent by component  $C_i$ , and  $m_2$  is a message sent by component  $C_j$ . Their timestamps are  $vt_i^{m_1}$  and  $vt_j^{m_2}$ , respectively. For these two equivalence relations, if Equation 4 holds, Equation 5 also holds. As a result, we need to proof only Equation 4. Furthermore, according to Algorithm 1, if  $m_1 \rightarrow m_2$ ,  $vt_i^{m_1} < vt_j^{m_2}$ . As a result, we need to proof only  $vt_i^{m_1} < vt_j^{m_2} \Rightarrow m_1 \rightarrow m_2$ , and the proof is as follows:

- Assuming  $vt_i^{m_1} < vt_j^{m_2}$  and  $m_1 \parallel m_2$ .
- Suppose  $vt_i^{m_1} = x$ . The only way component  $C_j$  can obtain a value for the  $i$ th entry of its vector, is that at least  $x$  is through a chain of messages originating from  $C_i$  (at message  $m_1$  or later).
- Such a chain implies that  $m_1$  and  $m_2$  are not concurrent. However, we find a contradiction, so the assumption does

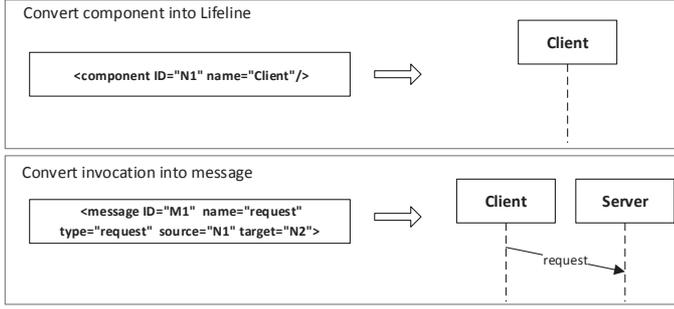


Fig. 4. Component and Message Translation

not hold. For example, if  $vt_i^{m_1} < vt_j^{m_2}$ ,  $m_1 \rightarrow m_2$  or  $m_2 \rightarrow m_1$ . At the same time, if  $m_2 \rightarrow m_1$ ,  $vt_i^{m_2} < vt_j^{m_1}$ . Here is a contradiction. Therefore,  $vt_i^{m_1} < vt_j^{m_2} \Rightarrow m_1 \rightarrow m_2$ .

- In summary, Equation 4 holds.

In summary, there exists an isomorphism between the set of partial ordered messages and the timestamps that attach to them. As a result, timestamps are useful to determine the relations between two messages.

We illustrate how to use our algorithm for detecting the ambiguity in the sequence diagram in Section I. For simplicity, we assume that this scenario consists of only the three components (*i.e.*, *A*, *B*, and *C*), and we assume that *B* and *C* do not send any messages before  $m_1$  and  $m_2$  are sent. Under this assumption, the timestamps of  $m_1$  and  $m_2$  are  $\langle 0, 1, 0 \rangle$  and  $\langle 0, 0, 1 \rangle$ , respectively. If *A* needs both  $m_1$  and  $m_2$  to deliver  $m_3$ , the resulting timestamp of  $m_3$  will be  $\langle 1, 1, 1 \rangle$  to  $m_3$ . On the other hand, if *A* needs only  $m_2$  (or  $m_3$ ), the resulting timestamp of  $m_3$  will be  $\langle 1, 1, 0 \rangle$  (or  $\langle 1, 0, 1 \rangle$ ). As every set of timestamps corresponds to a unique sequence, our approach detects an ambiguity, if more than one set of timestamps is produced during analysis.

## V. TRANSFORMATIONAL RULE FOR MODEL TRANSFORMATION

### A. Transformational Rule

*Transformational Rule* leads to change the thinking pattern of users and provides more practical way to implement it in the industry area. The rule is inspired by Machine Learning algorithm. To be specific, we borrow some ideas of Decision Tree [23] which helps us to determine the transformation directions. The rule is defined as follows:

**Definition 5.1** A *Transformational Rule* (TR) is to retrieve the module ( $x'$ ) which has the maximum entropy ( $E$ ) based on the current requirements module ( $x_i$ ) according to the users' preference ratio ( $k_i$ ) (*i.e.*, weight for the module feature). The formal definition of TR is as follows:

$$\max(E_1 * k_1, E_2 * k_2, \dots, E_n * k_n) \rightarrow x' \quad (6)$$

### Algorithm 1 The Vector Clock Algorithm

- 1: initialize the local clock for each component as a zero vector.
- 2: **if** component  $C_i$  is about sending a message  $M$  **then**
- 3:    $\text{dims} \leftarrow$  the total number of dimensions of  $vt_i$
- 4:   **for**  $j = 0$  to  $\text{dims}$  **do**
- 5:      $\text{tol} \leftarrow$  all of the MessageOccurrenceSpecifications happen before sending  $M$
- 6:     **for** each MessageOccurrenceSpecifications  $M_p$  in  $\text{tol}$  **do**
- 7:       **if**  $vt_i^{M_p}[j] > vt_i[j]$  **then**
- 8:          $vt_i[j] \leftarrow vt_i^{M_p}[j]$
- 9:       **end if**
- 10:     **end for**
- 11:   **end for**
- 12:    $vt_i[i] = vt_i[i] + 1$
- 13:   **while** there exist another message  $M_1$  on component  $C_1$  whose  $vt_i^{M_1}[i]$  equals  $vt_i[i]$  **do**
- 14:      $vt_i[i] \leftarrow vt_i[i] + 0.1$
- 15:   **end while**
- 16:   timestamp  $M$  with clock  $vt_i$  and send it
- 17: **end if**

where the  $n$  is the total number of the possible modules,  $E_i$  is entropy of module  $x_i$  and the sum of the  $k_i$  should equal to 1, *i.e.*,

$$k_0 + k_1 + \dots + k_n = 1 \quad (7)$$

Here we consider some popular models which are used to model architecture in the high level in different areas. The candidate models are selected as Petri net, UML model, AADL model and LQN model. The above equation can be written as:

$$\begin{aligned} &\max(E_{\text{Petri}} * k_1, E_{\text{UML}} * k_2, E_{\text{AADL}} * k_3, E_{\text{LQN}} * k_n) \\ &\rightarrow \text{GeneralModel} \end{aligned} \quad (8)$$

According to the domain expert experience, the UML is a most popular modeling language in industry area and the corresponding entropy is the maximum.

### B. Model Transformation

To make our approach more general, we provide a set of mapping rules between Breeze/ADL and UML. The benefit of mapping is that we can combine the advantages of both ADL and UML. The combination not only supports formally modeling scenarios at architecture-level, but also allows translating scenarios from Breeze/ADL to UML sequence diagrams that are supported by industrial tools, *e.g.*, Rational Rose, Enterprise Architect (EA), and PowerDesigner.

As EA is a professional and famous tool in industry, our approach translates scenarios from Breeze/ADL to EA style XML files. In this section, we define the mapping relations in the concrete syntax (Section V-B1) and the abstract syntax (Section V-C), respectively.

1) *Mapping Rules for Concrete Syntax*: To illustrate the mapping relations, we first present the concrete syntax of Sequence Diagram (CSSD).

**Definition 5.2** CSSD:= $\{Lifeline, Message, MessageOccurrenceSpecification, ExecutionSpecification\}$  is the concrete syntax of sequence diagrams, where:

- *Lifeline* is a set of participants of an interaction and a participant is defined as a component in Breeze/ADL.
- *Message* is an invocation/response among components.
- *MessageOccurrenceSpecification* specifies the occurrence of the events, e.g., sending and receiving of messages, or invoking and receiving of operation calls. As defined in UML<sup>3</sup>, it is a type of messages.
- *ExecutionSpecification* is a specification of the execution or an action within a Lifeline.

As in Breeze/ADL, components in *componentlist* correspond to lifelines in the sequence diagram, Figure 4 illustrates the mapping rule. The invocations in Breeze/ADL are mapped to messages between lifelines in sequence diagrams. In sequence diagrams, a message has two attributes such as a type (source or target) and a label. In particular, sequence diagrams use solid arrows to denote requests, and dot arrows to denote responses. Our approach maps these attributes to the corresponding attributes in Breeze/ADL.

Each message in Breeze/ADL corresponds to two events, i.e. a request and a response, in sequence diagrams. Our approach translates these events into *MessageOccurrenceSpecification* elements in sequence diagrams. As a result, each lifeline may have many *MessageOccurrenceSpecification* elements. In sequence diagrams, the interval between the earliest and latest *MessageOccurrenceSpecification* elements on the lifeline is specified by the *ExecutionSpecification* element. For example, the sequence diagram in Figure 1 includes two *MessageOccurrenceSpecification* elements on *Client*, i.e., *sending request* and *receiving response*. The timestamp of *sending request*  $\langle 1, 0 \rangle$  is smaller than that of *receiving Response*  $\langle 1, 1 \rangle$ , which means that *sending request* happens before *receiving response*. Figure 5 shows an example translation from Breeze/ADL to sequence diagrams. The event order is encoded in the *ExecutionSpecification* element of the lifeline.

### C. Mapping Rules for Abstract Syntax

The abstract syntax of sequence diagrams is defined in the meta-modeling language of UML. In particular, several fragments are related to sequence diagrams, e.g., the *BasicInteractions* package and the *Fragments* package. Breeze/ADL focuses on the elements in the *BasicInteractions* package, and it covers lifelines, messages, message ends, interactions, and general orders of sequence diagrams. Figure 6 shows the relations among these elements. In particular, a *GeneralOrdering* element denotes a binary relation among *OccurrenceSpecification* elements, and the binary relation defines that an *OccurrenceSpecification* element must occur before the other in a valid trace. This mechanism allows defining partial orders

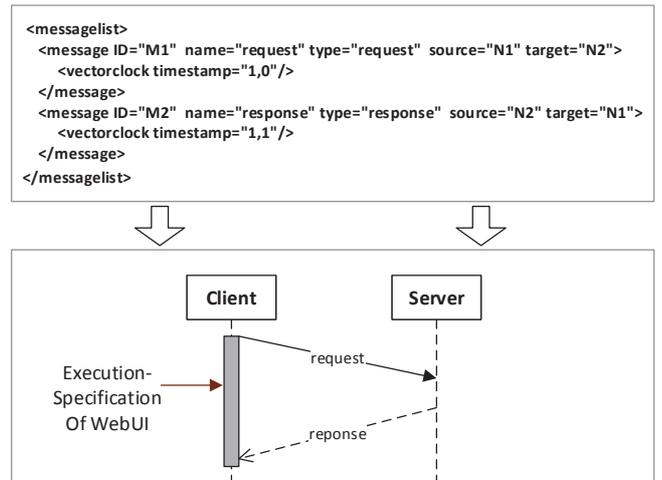


Fig. 5. Translation of ExecutionSpecification

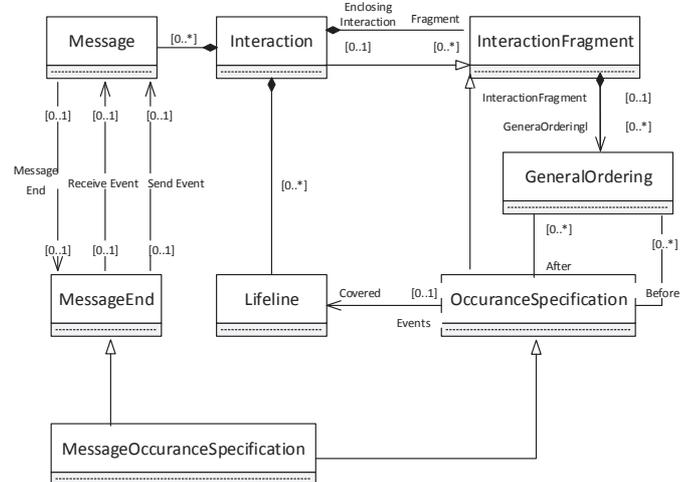


Fig. 6. Part of the abstract syntax of the BasicInteractions package

of *OccurrenceSpecifications*. When using the sequence diagram to depict scenarios, the partial order depicted by Vector Clock between messages is mapped to the *GeneralOrdering*. When transforming Breeze/ADL into sequence diagrams, our approach infers partial orders between messages by comparing their timestamps, which is based on the isomorphism property of the vector clock algorithm.

## VI. CASE STUDY

We implemented our approach as a plugin of Breeze, and with the support of the tool, in this section, we illustrate our approach using a poetry to music system. the latest version of Breeze is available at Github:

<https://github.com/BreezeCSA/Breeze>

### A. The Requirements

The implementation of the poetry to music system should follow the MVC framework<sup>4</sup>. As defined in the framework, the requests from customers are delivered to control components, and components further invoke function modules to

<sup>3</sup><http://www.omg.org/technology/documents/formal/uml.htm>

<sup>4</sup><http://www.martinfowler.com/eaDev/uiArchs.html>

```

Breeze\ADL of Modelling Poetry2Music
<arch xmi:id="arc1" name="Poetry2Music">
  <node xsi:type="Breeze:Component" xmi:id="n1" name="Control">
    <port xmi:id="p11" direction="inout"/>
    <port xmi:id="p12" direction="inout"/>
    <port xmi:id="p13" direction="inout"/>
    <port xmi:id="p14" direction="inout"/>
  </node>
  <node xsi:type="Breeze:Component" xmi:id="p15" name="WebUI">
    <port xmi:id="p16" direction="inout"/>
    <port xmi:id="p17" direction="in"/>
    .....
  </node>
  .....
  <ed e:id="e1" source="p12" target="p21">
    .....
  </arch>

```

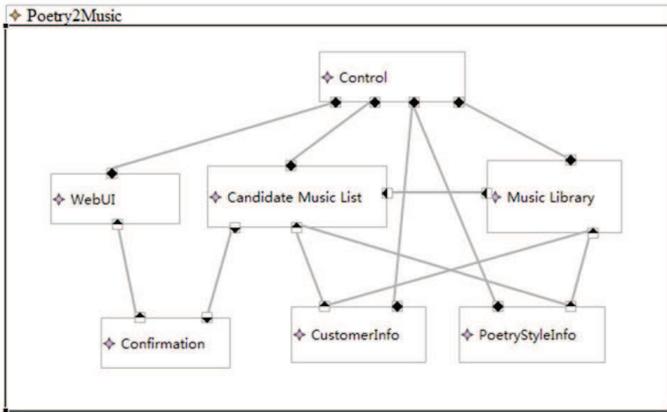


Fig. 7. The software architecture of poetry to music system

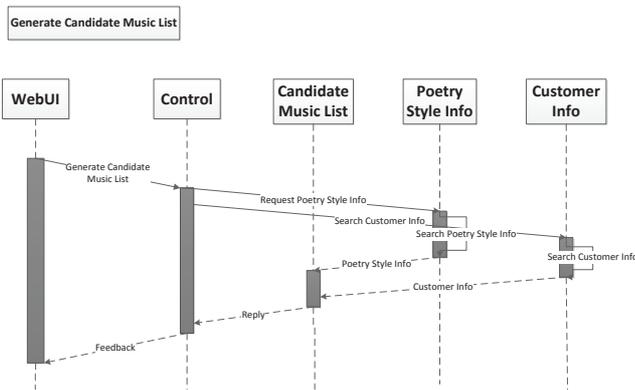


Fig. 8. The scenario of generating candidate music list ( $S_3$ ) illustrated in a sequence diagram

accomplish functionalities that are requested by customers. For simplicity, we focus on the functionalities of querying poetry style and generating related music list. To accomplish the two tasks, the system should at least implement the functionalities in the following scenarios:

- 1) *Logging in* ( $S_1$ ). Customers sign in the poetry to music system with their usernames and passwords.
- 2) *Querying Poetry Style Information* ( $S_2$ ). Customers query poetry with keywords, and find their correspond-

ing style.

- 3) *Generating candidate music list* ( $S_3$ ). According to poetry style, the system retrieval music library and generate candidate music list.
- 4) *Confirmation* ( $S_4$ ). The system arrange the music list and return results to customers.

We focus on only modeling  $S_3$ , due to space limit. Following the MVC framework, when a customer enters keywords, the *WebUI* component sends a request, *Generate Candidate Music List*, to the *Control* component. To accomplish this request, the *Control* component retrieves the poetry style and customer information from the *PoetryStyleInfo* component and the *CustomerInfo* component, respectively. After that, the *Control* component returns the information to the *WebUI* component, and the latter component presents the feedback to the customer.

### B. Modeling with UML

According the requirements as described in Section VI-A, with UML, the best choice for a designer it to draw a sequence diagram to describe  $S_3$ . Figure 8 shows such an example, and it presents many details of  $S_3$  (e.g., components and their messages). However, when developers implement the system according to the sequence diagram, they may easily become confused. For example, a developer may wonder whether there exists a strict order between *RequestPoetryStyleInfo* and *RequestCustomerInfo*, and whether both *PoetryStyleInfo* and *CustomerInfo* are required or only one of them is required before *Reply*.

### C. Modeling with Breeze

As introduced in Section II, Breeze supports two modeling languages such as Breeze/ADL and its graph format. Figure 7 shows the built architecture of the poetry to music system, in both formats. In particular, the upper part of Figure 7 shows the *WebUI* and *Control* components in Breeze/ADL. Due to space limit, we do not present the definitions of the other components. The Breeze/ADL file defines the ports of the two components and the interactions among these ports. According to directions of their messages, ports have three types such as *in*, *out* and *inout*. An interaction is defined as an edge between two ports. For example,  $\langle edge\ ID = "e1" source = "p12" target = "p21" \rangle$  denotes an  $e_1$  interaction from the  $p_{12}$  port to the  $p_{21}$  port. The lower part of Figure 7 shows the architecture of the whole poetry to music in the graph format of Breeze/ADL. In the graph, the small rectangles adhering to components represent ports, and lines represent interactions between ports.

### D. Reducing Ambiguities with our Extension

Our extension first encodes the components and their messages of the architecture in Section VI-C. Figure 9 shows the encoded architecture in Breeze/ADL. In the encoded Breeze/ADL file, the components in this scenario are defined in the *componentlist* tag, and the messages among the components are defined in the *messagelist* tag. The *ParallelScenario*

```

Scenario Generate Candidate Music List Definition in Breeze/ADL
<scenario>
  <scenario name="Login" id="S1">
  <componentlist>
  <component id="N1" name="WebUI"/>
  <component id="N2" name="Control"/>
  <component id="N3" name="Candidate Music List"/>
  <component id="N4" name="PoetryStyleInfo"/>
  <component id="N5" name="CustomerInfo"/>
  </componentlist>
  <message>
  <message id="M1" name="Input Keywords" type="request" source="WebUI" target="Control"/>
  <vectorlock timestamp="1,0,0,0,0"/>
  </message>
  <message>
  <message id="M2" name="Request Poetry Style Info" type="request" source="Control" target="PoetryStyleInfo"/>
  <vectorlock timestamp="1,1,0,0,0"/>
  </message>
  <message>
  <message id="M3" name="Request Customer Info" type="request" source="Control" target="CustomerInfo"/>
  <vectorlock timestamp="1,1,1,0,0"/>
  </message>
  <message>
  <message id="M4" name="Search Poetry Style Info" type="request" source="PoetryStyleInfo" target="PoetryStyleInfo"/>
  <vectorlock timestamp="1,1,0,1,0"/>
  </message>
  <message>
  <message id="M5" name="Search Customer Info" type="request" source="CustomerInfo" target="CustomerInfo"/>
  <vectorlock timestamp="1,1,0,0,1"/>
  </message>
  <message>
  <message id="M6" name="Poetry Style Info" type="reply" source="PoetryStyleInfo" target="Candidate Music List"/>
  <vectorlock timestamp="1,1,0,2,0"/>
  </message>
  <message>
  <message id="M7" name="Customer Info" type="reply" source="CustomerInfo" target="Candidate Music List"/>
  <vectorlock timestamp="1,1,1,0,2"/>
  </message>
  <message>
  <message id="M8" name="Reply" type="reply" source="Candidate Music List" target="Control"/>
  <vectorlock timestamp="1,1,1,1,2,2"/>
  </message>
  <message>
  <message id="M9" name="Feedback" type="reply" source="Control" target="WebUI"/>
  <vectorlock timestamp="1,2,1,1,2,2"/>
  </message>
  </message>
  </message>
  <parallelsenario>
  <maxparallelism scenarioID="S1" num="INF"/>
  <maxparallelism scenarioID="S2" num="INF"/>
  <maxparallelism scenarioID="S3" num="INF"/>
  <maxparallelism scenarioID="S4" num="0"/>
  </parallelsenario>
  </scenario>
  .....
</arch>

```

Fig. 9. The scenario of generating candidate music list ( $S_3$ ) illustrated in Breeze/ADL

vectors of scenarios are defined in the *parallelsenario* tag. The *parallelsenario* of  $S_3$  is  $\langle \infty, \infty, \infty, 0 \rangle$ , which means that the execution of  $S_3$  has no impact on  $S_1$ ,  $S_2$ , or itself, and  $S_3$  cannot run in parallel with  $S_4$ , since a user has to *input keywords* before returning the results to customers.

Each component is associated with a vector. As  $S_3$  contains 5 components, a timestamp in this scenario is a 5-dimension vector. According to Algorithm 1, the vectors are set to zero vectors initially. When the *WebUI* component sends the message *Generate Candidate Music List*, Breeze updates the timestamp of the component by executing  $vt_1[1] \leftarrow vt_1[1] + 1$ , and attaches the new timestamp,  $\langle 1, 0, 0, 0, 0 \rangle$ , to the message

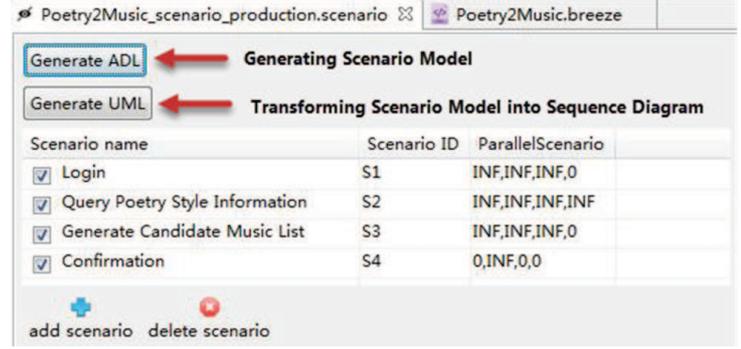


Fig. 10. Adding Scenarios

*Generate Candidate Music List*. When other messages such as *RequestPoetryStyleInfo*, *SearchPoetryStyleInfo*, *SearchCustomerInfo*, *PoetryStyleInfo*, *CustomerInfo* and *Feedback* are sent, Breeze updates their vectors in a similar way.

When the *Control* component sends the *RequestCustomerInfo* message, Breeze executes  $vt_2[2] \leftarrow vt_2[2] + 0.1$ , and the adaption avoids its timestamp from equaling to the timestamp of *RequestPoetryStyleInfo*. As *Reply* is the last message that involves the *PoetryStyleInfo* component and the *CustomerInfo* component, the timestamps of the two components become the largest, after sending the message.

After the timestamps are calculated, Breeze presents the partial order between these messages. In this scenario, there is no partial order between messages *RequestPoetryStyleInfo* and *RequestCustomerInfo*, since  $\text{floor}(vt_i^{\text{RequestPoetryStyleInfo}}) = \text{floor}(vt_i^{\text{RequestCustomerInfo}})$ . It is also very convenient for developers to obtain the information from these timestamps that both message *PoetryStyleInfo* and *CustomerInfo* are needed to deliver message *Reply*. As each set of timestamps corresponds to a unique conditions and sequences of events, a designer is able to determine which sequence best fits the requirements by choosing the correct set of timestamps.

Breeze allows translating the scenario from Breeze/ADL to a UML sequence diagram. Based on the mapping rules, each component is translated to a lifeline. In the sequence diagram, Breeze fills in *ExecutionSpecification* elements based on the timestamps of the *MessageOccurrenceSpecification* elements.

In this example, the designer decides that the following timestamps best fit the desirable functionality:

- *RequestPoetryStyleInfo* ( $\langle 1, 1, 0, 0, 0 \rangle$ )
- *RequestCustomerInfo* ( $\langle 1, 1.1, 0, 0, 0 \rangle$ )
- *Reply* ( $\langle 1, 1.1, 1, 2, 2 \rangle$ )
- *Feedback* ( $\langle 1, 2.1, 1, 1, 2 \rangle$ )

Based on the timestamps, Breeze determines the order of the four messages, and encodes the event order in the *ExecutionSpecification* element of the lifeline.

### E. Tool support

This section illustrates how to use Breeze to model and to translate scenarios step by step.

- 1) Step 1: Adding scenarios. As introduced in Section VI, after modeling the software architecture of the poetry to music system (Figure 7), we add the corresponding

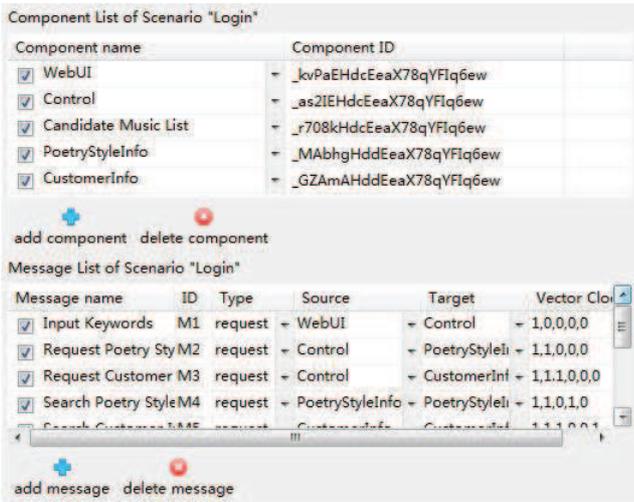


Fig. 11. Adding Components and Messages

scenarios such as *Login*, *Query Poetry Style Information*, *Generate Candidate Music List*, and *Confirmation* (Figure 10).

For each scenario, the user of Breeze needs to specify its name, ID, and scenarios that can run in parallel with the scenario.

- 2) Step 2: Initializing scenarios. The user of Breeze needs to initialize each scenario by adding all involved components and messages. Figure 11 shows the component list and the message list of the  $S_3$  scenario. The columns in the message list correspond to the attributes that are defined in Breeze/ADL.
- 3) Step 3: Generating scenario model. Breeze generates scenarios in Breeze/ADL, when its user clicks the corresponding button in Figure 10.
- 4) Step 4: Translating scenarios. Breeze translates scenarios from Breeze to sequence diagrams, when its user clicks the *Generate UML* button in Figure 10. Breeze generates sequence diagrams in the format of Enterprise Architect (EA). EA displays translated sequence diagrams in its editor, and elements of scenarios in its project browser. Figure 12 shows the translated  $S_3$  scenario.

## VII. CONCLUSION AND FUTURE WORK

In this paper, we reduce ambiguity in scenario-based software architecture analysis by proposing a creative scenario modeling and analysing approach, based on our extended Breeze/ADL, that facilitates designers to define scenarios in an unambiguity way. The modeling results are useful to support subsequent scenario-based analysis at architecture level. In addition, to combine the advantages of both Breeze/ADL and UML, we propose a set of rules for model transformation. These rules allow us to translate scenarios from Breeze/ADL to sequence diagrams automatically.

There are still some issues that should be addressed in our next step work. As Breeze does not support all the elements in sequence diagrams, it cannot eliminate all ambiguities in sequence diagrams, but we plan to introduce exploratory rule

to add more related elements in future work. In addition, we plan to extend Breeze, so it can translate existing sequence diagrams into Breeze/ADL for scenario-based software architecture analysis.

## REFERENCES

- [1] R. Pandey, Architectural Description Languages (ADLs) vs UML: A Review, *ACM SIGSOFT Software Engineering Notes*, 35(3), 1–5, 2010.
- [2] B. Tekinerdogan, H. Sozer, and M. Aksit, Software Architecture Reliability Analysis Using Failure Scenarios, *Journal of Systems and Software*, 81(4), 558–575, 2008.
- [3] R. Kazman, G. Abowd, L. Bass, and P. Clements, Scenario-based Analysis of Software Architecture, *Software*, 13(6), 47–55, 1996.
- [4] P. Bose, Scenario-driven Analysis of Component-based Software Architecture Models, In Proc. WICSA1, 1999.
- [5] S. Yacoub, B. Cukic, and H. H. Ammar, A Scenario-based Reliability Analysis Approach for Component-based Software, *Reliability*, IEEE Transactions on, 53(4), 465–480, 2004.
- [6] L. Cheung, L. Golubchik, and N. Medvidovic, Sharp: A Scalable Approach to Architecture-level Reliability Prediction of Concurrent Systems, In Proc. 32th ICSE Workshop, pages 1–8, ACM, 2010.
- [7] C. Sibertin-Blanc, N. Hameurlain, and O. Tahir, Ambiguity and Structural Properties of Basic Sequence Diagrams, *Innovations in Systems and Software Engineering*, 4(3), 275–284, 2008.
- [8] C. Li, L. Huang, L. Chen, and C. Yu, Breeze/adl: Graph Grammar Support for An XML-based Software Architecture Description Language, In Proc. 37th COMPSAC, pages 800–805, IEEE, 2013.
- [9] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, Specifying Distributed Software Architectures, In Proc. 5th ESEC, pages 137–153. Springer, 1995.
- [10] R. Allen, R. Douence, and D. Garlan, Specifying and Analyzing Dynamic Software Architectures. In Proc. 2th FASE, pages 21–37. Springer, 1998.
- [11] F. Oquendo, Y-adl: An Architecture Description Language Based On the Higher-order Typed Y-calculus for Specifying Dynamic and Mobile Software Architectures, *ACM SIGSOFT Software Engineering Notes*, 29(3), 1–14, 2004.
- [12] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide, Pearson Education India, 1999.
- [13] M. Raynal and M. Singhal, Logical Time: Capturing Causality in Distributed Systems, *Computer*, 29(2), 49–56, 1996.
- [14] N. Lassing, D. Rijsenbrij, and H. van Vliet, On Software Architecture Analysis of Flexibility, Complexity of Changes: Size isnt Everything. In Proc. 2th NOSA Workshop, volume 99, pages 1103–1581, 1999.
- [15] C. H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman. An Approach to Software Architecture Analysis for Evolution and Reusability, In Proc. 7th CASCON, page 15, IBM Press, 1997.
- [16] G. Rodrigues, D. Rosenblum, and S. Uchitel, Using Scenarios to Predict the Reliability of Concurrent Component-based Software Systems, In Proc. 14th FASE, pages 111–126, Springer, 2005.
- [17] L. G. Williams and C. U. Smith, Pasa sm: a Method for the Performance Assessment of Software Architectures, In Proc. 3th WOSP, pages 179–189, ACM, 2002.
- [18] M. A. Boden, The Creative Mind: Myths and Mechanisms (2nd ed), Routledge, London, 2004.
- [19] M. A. Boden, Computer Models of Creativity, *AI Magazine*, 30(3), 23–34, 2009.
- [20] Z. Micskei and H. Waeselynck, UML 2.0 Sequence Diagrams Semantics, Universite de Toulouse, Tech. Rep. 8389, 2008.
- [21] C. Li, L. Huang, and L. Chen, Breeze Graph Grammar: a Graph Grammar Approach for Modeling the Software Architecture of Big Data-Oriented Software Systems, *Software: Practice and Experience*, 2014.
- [22] wikipedia. Partially ordered set. [http://en.wikipedia.org/wiki/Partially\\_ordered\\_set](http://en.wikipedia.org/wiki/Partially_ordered_set) Formal definition.
- [23] Quinlan, J. R.: Induction of Decision Trees. *Machine Learning*. 1(1), 81–106 (1986)

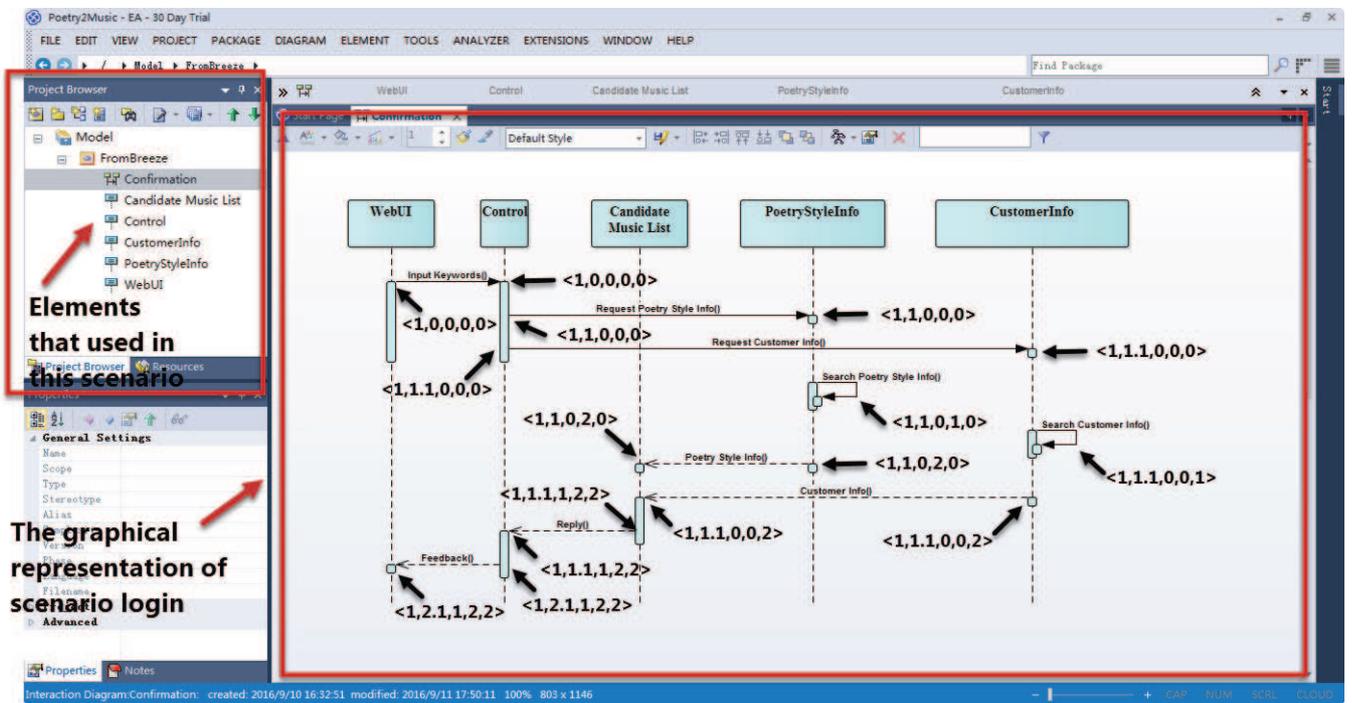


Fig. 12. Scenario Generate Candidate Music List in Enterprise Architect