Automatically Deriving UML Sequence Diagrams from Use Cases

Tao Yue\textsuperscript{1,2}, Lionel C. Briand\textsuperscript{2}, and Yvan Labiche\textsuperscript{1}

\textsuperscript{1} Carleton University, Software Quality Engineering Lab, 1125 Colonel By Drive
Ottawa, ON K1S 5B6, Canada
labiche@sce.carleton.ca

\textsuperscript{2} Simula Research Laboratory & University of Oslo,
P.O. Box 134, Lysaker, Norway
\{tao, briand\}@simula.no

Abstract. Use cases are commonly used to structure and document requirements during requirement elicitation while sequence diagrams are often used during the analysis phase to document use cases as objects’ interactions. Since creating such sequence diagrams is mostly manual, automated support would provide significant, practical help. Additionally, traceability could be easily established through automated transformation, which could then be used for instance to relate requirements to design. In this paper, we propose an approach and a tool to automatically generate sequence diagrams from use cases while establishing traceability links. We validate our approach with six case studies, where we compare sequence diagrams generated by our tool to the ones devised by experts and trained 4th year undergraduate students. Results show that sequence diagrams automatically generated by our tool are highly consistent with the ones devised by experts and are also very complete. Results also show that the automatically generated diagrams are far more complete than the ones manually created by students. These encouraging results suggest that our approach and tool would indeed provide significant, practical help to engineers creating (initial) sequence diagrams from use case descriptions.

Keywords: Use Case; Use Case Modeling; UML; Sequence Diagram; Transformation; Traceability; Automation; Natural Language Processing.

1 Introduction

Use case modeling, through use case diagrams and use case textual specifications, is commonly applied to structure and document requirements from a user’s point of view during requirements elicitation \cite{1}. In a typical UML-based software development process, an analysis model is then created and is typically composed of class and sequence diagrams that describe structure and interactions, respectively. Sequence diagrams then document use cases as objects’ interactions \cite{1}. Realizing use cases by means of sequence diagrams ensures that the behavior of a system is described in a precise and visual manner. However, this activity heavily relies on heuristics \cite{2} and providing effective automated support is therefore important.

Additionally, automated transformation would enable automated traceability from requirements to sequence diagrams. Traceability is important during software development since it allows engineers to understand the connections between various artifacts of a software system. Traceability is also mandated by numerous standards (e.g., IEEE Std. 830-1998 \cite{3}) to support, for example, safety verification \cite{4}.

We conducted a systematic literature review \cite{5} on transformations of textual requirements into analysis models, represented as class, sequence and activity diagrams. Results show that existing transformations are either not fully automated or not necessarily practical and complete, and that most of them do not support traceability.

Our use case modeling approach RUCM \cite{6} relies on a use case template and a set of restriction rules for textual Use Case Specifications (UCSs) to reduce the imprecision and incompleteness inherent to UCSs. We have conducted a controlled experiment to evaluate RUCM and results indicate that RUCM, though it enforces a template and restriction rules, has enough expressive power, is easy to use, and helps improve the understandability of use cases and the quality of manually derived analysis models \cite{6}.

The current work is part of the aToucan approach and tool \cite{7}, which aims to transform a Use Case Model (UCMod) produced with RUCM into an initial UML analysis model that includes class \cite{7}.
activity [8] and sequence diagrams. aToucan involves three steps. First, requirements engineers manually define use cases by following RUCM. Second, aToucan reads these textual UCSs to identify Part-Of-Speech (POS) and grammatical relation dependencies of sentences, and then records that information into an instance of the metamodel UCMeta (our intermediate model) (Section 3.2). The third step is to transform the instance of UCMeta into an analysis model as an instance of the UML 2.0 metamodel. During these transformations, aToucan establishes traceability links between the UCMod and the generated UML diagrams. The objective of this work is to devise a method and tool to automatically generate UML sequence diagrams, that are consistent with generated class [7] and activity [8] diagrams, while establishing traceability links between requirements and sequence diagram elements.

Four case study systems have been used to evaluate sequence diagrams generated by aToucan against the ones devised by experts. Results indicate that, for all the six use cases we used, the sequence diagrams generated by aToucan are highly consistent with the ones devised by experts, and are also very complete. We also compared the sequence diagrams manually created by 30 fourth-year undergraduate students with the ones automatically generated by aToucan for two of those case study systems. Results show that, for the five use cases we used, diagrams generated by aToucan are far more complete than the ones manually generated by students.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 discusses RUCM and UCMeta. Section 4 discusses our transformation approach. Tool support is briefly discussed in Section 5. Evaluation is discussed in Section 6. Section 7 concludes the paper.

2 Related Work

We conducted a systematic literature review [5] on transformations of textual requirements into analysis models, represented as class, sequence, and activity diagrams. A carefully designed paper selection procedure in scientific journals and conferences from 1996 to 2008 and Software Engineering textbooks identified 20 primary studies (16 approaches). The method proposed here is based on the results of this review, with a focus on automatically deriving sequence diagrams from UCMods. Diaz et al. [9] propose an approach to derive interaction diagrams from use cases. Use cases are transformed into a use case linguistic model, which is then transformed into extended interaction diagrams by applying a set of transformation rules. Though the authors mention a tool has been developed for validation purposes, no automation or tool supporting their method is discussed. The approach does not address traceability. Case studies (not described in the paper) were performed to validate the transformations. The validation process is to compare fragments of the interaction diagrams obtained from the tool with the ones manually constructed by experts. Results show that 65% of the sequence diagram fragments generated by the tool are identical (i.e., modeling the same interactions with the same instances and the same messages) to the experts’ sequence diagram fragments, 28% of the automatically generated fragments are equivalent (modeling same interactions with different instances and messages), and 7% of these fragments are different (modeling different interactions). It is however unclear how sequence diagram fragments were defined. Furthermore, no evaluation measurement framework is provided in the paper, or in available documentation (e.g., technical report), to allow comparisons and replications.

A series of methods is proposed in [10] to precisely capture requirements and then manually transform requirements into a conceptual model composed of object models (e.g., class diagrams), dynamic models (i.e., state machines and sequence diagrams), and functional diagrams. Traceability links can be established between classes and use cases. They specify how the data (classes) are manipulated in the use cases: create, read, update, delete. The approach does not purport to provide a solution for transforming requirements into analysis models. Instead, it proposes a set of techniques for users to precisely specify requirements and conceptual models, and also proposes a process to guide the users in deriving the conceptual models from the requirements. No transformation method is reported in the paper.

Śmiaalek et al. [11] propose a set of three semantically equivalent, complementary use case scenario representations: restricted language representation, activity and interaction representation; which can be automatically transformed into one another. All these three representations are formalized using the MOF metamodel [12]. Several mapping rules among the three different representations are provided.
Since the paper does not attempt to provide a solution for the transformation of requirements into analysis models, though the proposed approach could be adapted for that purpose, no transformation is discussed (besides several mapping rules).

Feijs [13] establishes a correspondence between a use case written in natural language, without any restriction, and a Message Sequence Chart (MSC) [14]. A large part of the proposed transformation is manual (e.g., classifying sentences in use cases, identifying classes, events in sentences) An experimental tool was developed; however only the user interface for editing the intermediate model of the tool is discussed in the paper. Therefore we don’t know whether the MSC generation is automated. A case study with one use case is discussed to illustrate the approach, and the result shows that it can generate some MSCs that are similar to the kind of MSCs “obtained in practices”, as stated by the authors. No empirical evaluation is performed. The approach does not address traceability.

In summary, none of the existing approaches is able to fully and automatically generate sequence diagrams from a use case model while establishing traceability links, which is exactly what we are proposing in the next section. Furthermore, none of the existing approaches proposes a carefully justified evaluation framework (see Section 6) to evaluate automatically derived sequence diagrams (except for [9]).

3 Background

We first provide an overview of our use case modeling approach RUCM (Section 3.1) and the intermediate model (UCMeta) of our transformations (Section 3.2) [6].

3.1 RUCM

RUCM encompasses a use case template and 26 restriction rules [6]. Rules are classified into restrictions on the use of Natural Language (NL), and rules enforcing the use of keywords for specifying control structures. The goal of RUCM is to be easy to use while reducing ambiguity and improving understanding, and to facilitate automated analysis [6]. Below we discuss the features of RUCM that are particularly helpful to generate sequence diagrams. An example of UCS documented

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Validate PIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brief Description</td>
<td>The system validates ATM customer PIN number.</td>
</tr>
<tr>
<td>Precondition</td>
<td>The system is idle. The system is displaying a Welcome message.</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Card Reader</td>
</tr>
<tr>
<td>Dependency</td>
<td>None</td>
</tr>
</tbody>
</table>

**Basic flow steps**

1) Card Reader sends the ATM card information of ATM Customer to the system.
2) If the system recognizes the ATM card, THEN the system reads the ATM card number.
3) The system prompts ATM Customer for PIN number.
4) ATM Customer enters PIN number to the system.
5) The system VALIDATES THAT the expiration date of the ATM card is valid.
6) The system VALIDATES THAT the PIN number entered by ATM customer matches the ATM card PIN number maintained by the system.
7) The system obtains the ATM customer accounts accessible with the ATM card.
8) The system displays ATM customer accounts.
9) The system prompts ATM customer for transaction type: Withdrawal, Query, or Transfer. Postcondition: ATM customer PIN number has not been validated.
10) The system ejects the ATM card. ENDIF Postcondition: ATM customer PIN number has not been validated.

**Specific Alt. Flow (RFS Basic flow 2)**

ELSE 1) The system ejects the ATM card. 2) ABORT. Postcondition: ATM customer PIN number has not been validated.

**Specific Alt. Flow (RFS Basic flow 7)**

IF ATM customer enters the incorrect PIN number three times, THEN 1) The system confiscates the ATM card; 2) ABORT, ELSE RESUME BACK Basic flow 3. Postcondition: ATM customer PIN number has not been validated.

**Bounded Alt. Flow (RFS Basic flow 5-6)**

1) The system confiscates the ATM card; 2) ABORT. Postcondition: ATM customer PIN number has not been validated.

**Global Alt. Flow**

IF ATM customer enters Cancel THEN 1) The system cancels the transaction MEANWHILE the system ejects the ATM card. 2) ABORT. ENDIF Postcondition: ATM customer PIN number has not been validated. The system is idle. The system is displaying a Welcome message.
with RUCM is presented in Table 1 (excerpt).

A use case description has one basic flow and can have one or more alternative flows (first column in Table 1). An alternative flow always depends on a condition occurring in a specific step in a flow of reference, referred to as reference flow, which is either the basic flow or an alternative flow itself. We classify alternative flows into three types: A specific alternative flow refers to a specific step in the reference flow; A bounded alternative flow refers to more than one step in the reference flow—consecutive steps or not; A global alternative flow (called general alternative flow in [16]) refers to any step in the reference flow. Use case Validate PIN has one specific alternative flow referring to step 2 of the basic flow, one bounded alternative flow referring to step 5 and step 6 of the basic flow, and one global alternative flow (Table 1).

The different types of alternative flows specify precisely the interactions between the reference flow and its alternative flows. For specific and bounded alternative flows, a RFS (Reference Flow Step) section specifies one or more (reference flow) step numbers. Whether and where the flow merges back into the reference flow or terminates the use case is specified as the last step of the alternative flow.

RUCM defines a set of keywords to specify conditional logic sentences (IF-THEN-ELSE-ELSEIF-ENDIF), concurrency sentences (MEANWHILE), condition checking sentences (VALIDATES THAT), and iteration sentences (DO-UNTIL). These keywords limit chances of ambiguities in UCSs. They also greatly facilitate the automated generation of sequence diagrams. There exists a direct mapping between these situations and some UML sequence diagram elements. For example, concurrency sentences with keyword MEANWHILE can be accurately transformed into a “par” combined fragment, each operand of which contains the messages corresponding to a parallel sentence connected by keyword MEANWHILE. Keywords ABORT and RESUME STEP are used to describe an exceptional exit action and where an alternative flow merges back in its reference flow, respectively. An alternative flow ends either with ABORT or RESUME STEP, which means that the last step of the alternative flow should clearly specify whether the flow returns back to the reference flow and where (using keywords RESUME STEP followed by a returning step number) or terminates (using keyword ABORT).

3.2 UCMeta

UCMeta is the intermediate model in aToucan [7], used to bridge the gap between a textual UCMod and a UML analysis model (class, sequence and activity diagrams). As a result, we have two transformations: from the textual UCMod to the intermediate model, and from the intermediate model to the analysis model. Metamodel UCMeta also complies with the restrictions and use case template of RUCM. The current version of UCMeta is composed of 108 metaclasses and is expected to evolve over time. The detailed description of UCMeta is given in [7].

UCMeta is hierarchical and contains five packages: UML::UseCases, UCSTemplate, SentencePatterns, SentenceSemantics, and SentenceStructure. UML::UseCases is a package of UML 2 superstructure [17], which defines the key concepts used for modeling use cases such as actors and use cases. Package UCSTemplate models the concepts of the use case template of RUCM: those concepts model the structure that one can observe in Table 1.

SentencePatterns is a package describing different types of sentence patterns, which uniquely specify the grammatical structure of simple sentences, e.g., SVDO (subject-verb-direct object) (e.g., Table 1, basic flow, step 9). We classify sentence patterns into eight categories: SV (subject-verb), SVC (subject-verb-complement), SVCC (subject-verb-complement-complement), SVDO (subject-verb-direct object), SVDOC (subject-verb-direct object-complement), SVIODOC (subject-verb-indirect object-direct object-complement), SLVSubjectComplt (subject-linking verb-subject complement). These are a refinement of the five basic English sentence patterns proposed in Linguistics [18] and English grammar books [19].

SentenceSemantics is a package modeling the classification of sentences from the aspect of their semantic functions in a UCMod. Each sentence in a UCS can either be a ConditionSentence or an ActionSentence. An action sentence must describe one of the following five transactions, of which the first four are reused from [20]: 1) Initiation: the primary actor sends a request and data to the system. 2) Validation: the system validates a request and data. 3) InternalTransaction: the system alters its internal state (e.g., recording or modifying something). 4) Response2PrimaryActor: the system replies
to the primary actor with a result. 5) Response2SecondaryActor: the system sends a request to a secondary actor.

Package SentenceStructure models NL concepts in sentences such as subject or Noun Phrase (NP). Package UCSTemplate is mostly related to the sequence diagram generation and therefore it is the only package discussed below due to space limitation. Package UCSTemplate not only models the concepts of the use case template but also specifies three kinds of sentences: SimpleSentence, ComplexSentence, and SpecialSentence. In linguistics, a SimpleSentence has one independent clause and no dependent clauses [18]: one Subject and one Predicate. UCMeta has four types of ComplexSentence: ConditionCheckSentence, ConditionalSentence, IterativeSentence, and ParallelSentence, which correspond to four keywords that are specified in RUCM (Section 3.1) to model conditions, iterations, concurrency, and validations in UCS sentences. UCMeta also has four types of special sentences to specify how flows in a use case or between use cases relate to one another. They correspond to keywords RESUME STEP, ABORT, INCLUDE USE CASE, EXTENDED BY USE CASE, and RFS (Reference Flow Step).

4 Approach

Recall that our objective is to automatically transform a textual UCMod expressed using RUCM into UML sequence diagrams while establishing traceability links. Our approach relies on a prior identification of classes from UCMod [7] that, broadly speaking, implements Abbott’s heuristics [21]. We present transformation rules, algorithms, and traceability in Section 4.1, Section 4.2, and Section 4.3, respectively. Some of the transformations are illustrated in Figure 1, using the example of Table 1.

4.1 Transformation Rules

The transformation from an instance of UCMeta to sequence diagrams involves 19 rules, summarized in Table 2. (The interested reader is referred to Appendix A for rule specifications in the Kermeta language [22].) Note that these transformations rely on the identification of classes and operations of a class diagram. Generating class diagrams is however described in [7]. The indentation of rules in Table 2 indicates some rules (composite rules) are composed of, and therefore invoke other rules (atomic rules), e.g., composite rule 1 invokes atomic rules 1.1-1.6. Rules 1.1-1.6 process three types of sentences: SimpleSentence, ComplexSentence, and SpecialSentence.

Rules 1.1-1.6 process seven of the eight different types of simple sentences: SV, SVC, … (Section 3.2). SVSubjComplt (subject-link verb-subject complement) sentences (e.g., “The system is idle”) are condition sentences and are therefore not included in this rule set. These sentences appear in pre- and post-conditions, or in complex sentences (e.g., condition check sentences and conditional sentences handled by rules 2.1 and 2.2, respectively).

Rule 1.1 generates a self-message on the system lifeline for sentences with pattern SV. These sentences all have type InternalTransaction and their subject is therefore always “the system”. We generate a lifeline in each sequence diagram to represent “the system”, which we refer to as “the system lifeline” in the rest of this paper. Sentences with patterns SVC, SVCC, SVDO, SVDOC, SVIODO, and SVIODOC, handled by rules 1.2-1.6, might have four different transaction types: Initiation, InternalTransaction, ResponseToPrimaryActor, and ResponseToSecondaryActor (Section 3.2). Each rule generates different UML sequence diagrams elements according to the transaction type. For example, if the sentence pattern of a simple sentence is SVDO and its transaction type is Initiation, then rule 1.4 generates two sequential messages: one from the actor lifeline to the actor boundary lifeline and one from the boundary lifeline to the system lifeline. The rationale is that the primary actor sends a request and data to the system in an Initiation sentence (Section 3.2) and therefore the subject of the sentence is always the primary actor of the use case and its direct object must be the system. The condensed notation “actor lifeline (subject) → actor boundary lifeline → the system lifeline (direct object)” in Table 2 represents these two sequential messages. If the transaction type of the sentence is InternalTransaction then only one message from the system lifeline to the lifeline representing the class identified from the object of the sentence [7] is generated. Notice that if such a lifeline does not exist, it is created (the classifier is the class derived from the object of the sentence) and a create message from the system lifeline to this newly created lifeline is generated. For example,
the InternalTransaction sentence (i.e., basic flow, step 2, THEN action, in Table 1) has object “the ATM card number”, which is a noun phrase with noun “number” as its head. This noun is identified as a class [7]. According to the rule, a lifeline representing entity class Number (i.e., <<Entity>>:Number) is generated and a message (i.e., 2: read the ATM card number) from the system lifeline (i.e., <<Control>>:Validate PIN Control) to the newly created lifeline is created. If the transaction type of the sentence is ResponseToPrimaryActor or ResponseToSecondaryActor, then the subject of the sentence is always “the system” and the object of the sentence is either the primary actor or a secondary actor. Therefore, two sequential messages are generated from the system lifeline to the actor boundary class lifeline and then from it to the actor lifeline.

Table 2. Summary of transformation rules

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Create an Interaction (UML metaclass) for each use case.</td>
</tr>
<tr>
<td>1</td>
<td>Invoke rules 1.1-1.6 to process seven (of eight) types of simple sentence.</td>
</tr>
<tr>
<td>1.1</td>
<td>InternalTransaction—generate a self-message at the system lifeline.</td>
</tr>
<tr>
<td>1.2</td>
<td>InternalTransaction—generate two sequential messages: actor lifeline (subject) → actor boundary class lifeline → system lifeline (complement).</td>
</tr>
<tr>
<td>1.3</td>
<td>InternalTransaction—generate two sequential messages: actor lifeline (subject) → actor boundary class lifeline → system lifeline (one of the complements).</td>
</tr>
<tr>
<td>1.4</td>
<td>InternalTransaction—generate two sequential messages: actor lifeline (subject) → actor boundary class lifeline → system lifeline (direct object).</td>
</tr>
<tr>
<td>1.5</td>
<td>InternalTransaction—generate two sequential messages: system lifeline (subject) → boundary class lifeline of the actor (the object) → the actor lifeline.</td>
</tr>
<tr>
<td>1.6</td>
<td>InternalTransaction—generate two sequential messages: system lifeline (subject) → boundary class lifeline of the actor (the indirect object) → the actor lifeline.</td>
</tr>
<tr>
<td>2</td>
<td>Invoke rules 2.1-2.4 to process complex sentence types.</td>
</tr>
<tr>
<td>2.1</td>
<td>ConditionCheckSentence—if the sentence contains an actor to execute the action, generate two sequential messages: the system lifeline (the subject) → the boundary class lifeline of the actor → the actor lifeline.</td>
</tr>
<tr>
<td>2.2</td>
<td>ConditionCheckSentence—if the sentence contains an actor to execute the action, generate an “alt” combined fragment for the alternative flow branching from the sentence and do the following: 1) generate an interaction, 2) generate an interaction used in the combined fragment referring to the newly generated interaction, and 3) invoke rules 1-3 to generate messages for the sentences contained in each alternative flow.</td>
</tr>
<tr>
<td>2.3</td>
<td>ParallelSentence—generate a “par” combined fragment and invoke rules 1-3 to generate messages for the sentences connected by the keyword MEANWHILE.</td>
</tr>
<tr>
<td>2.4</td>
<td>IterativeSentence—generate a “loop” combined fragment and invoke rules 1-3 to generate messages for the sentences in the loop.</td>
</tr>
<tr>
<td>3</td>
<td>Invoke rules 3.1-3.4 to process each special sentence.</td>
</tr>
<tr>
<td>3.1</td>
<td>IncludeSentence—generate an interaction use referring to the interaction corresponding to the included use case and place the interaction use where the special sentence is invoked.</td>
</tr>
<tr>
<td>3.2</td>
<td>ExtendSentence—generate an interaction use referring to the interaction corresponding to the extending use case and place the interaction use where the special sentence is invoked.</td>
</tr>
<tr>
<td>3.3</td>
<td>AbortSentence—generate a destroy message pointing to the system lifeline.</td>
</tr>
<tr>
<td>3.4</td>
<td>ResumeBackSentence—generate a gate message referring to resume-back-step (message).</td>
</tr>
<tr>
<td>4</td>
<td>Process global alternative flows. The rule also invokes rules 1-3 to process the steps contained in the global alternative flows.</td>
</tr>
</tbody>
</table>

Rule 2 invokes rules 2.1-2.4 to process four different types of complex sentences. If the action of the sentence of a condition check sentence (containing keyword VALIDATES THAT) is initiated by the system, then a self message is generated at the system lifeline, otherwise a message is generated from the system lifeline to the actor lifeline (rule 2.1). An instance of the “opt” CombinedFragment is
generated for the alternative flow branching from the sentence. This combined fragment contains an instance of InteractionUse which refers to the instance of Interaction generated to contain the alternative flow actions. Rule 2.2 processes conditional sentences containing keyword IF-THEN-ELSE-ELSEIF-END. An “alt” CombinedFragment is generated and conditions of the sentence (i.e., IF and/or ELSEIF conditions) map to the guards of each operand and the THEN and ELSE actions are mapped to sequence diagram elements in the corresponding operands. For example, basic flow, step 2 in Table 1 is transformed into the “alt” combined fragments with condition [IF the system recognizes the ATM card]. Rule 2.3 transforms a parallel sentence into an instance of the “par” CombinedFragment. Each sentence connected by keyword MEANWHILE maps to an operand of the “par” combined fragment. Rule 2.4 transforms an iterative sentence into an instance of the “loop” combined fragment.

Atomic rules 3.1-3.4 process special sentences containing keywords INCLUDE USE CASE, EXTENDED BY USE CASE, ABORT, and RESUME STEP. For example, step 2 of the specific alternative flow (Table 1) is transformed into a destroy message (message 3 of the ELSE operand of the “alt” combined fragment with condition [IF The system recognizes the ATM card]).

Rule 4 processes global alternative flows, which refer to any step in the reference flow (Section 3.1). Such a behavior is very hard to model in sequence diagrams: one would need a “opt” combined fragment for each message in the diagram, which would clutter the diagram. Activity diagrams are a better solution to model this [8]. Our current solution is to generate an “alt” combined fragment. One of its operands covers all the messages of the reference flow and the rest covers the messages generated for the alternative flow steps. For example, an “alt” combined fragment is generated for the global alternative flow (Table 1). Operand [IF NOT: ATM customer enters Cancel] of the combined fragment covers all the elements in the basic flow and operand ELSE contains interaction use Global Alt. Flow, which refers to another interaction covering all the elements corresponding to the steps of the global alternative flow. Also notice that the other two types of alternative flows (i.e., specific and bounded alternative flows) are taken care of by the rules transforming ConditionCheckSentence and ConditionalSentence complex sentences.

---

4.2 Algorithm

Rules are structured in a hierarchy, as illustrated by the numbering scheme in Table 2, which indicates that some rules have to be executed one after the other while others can be executed independently. This facilitates the modification, addition, and deletion of rules and also simplifies the algorithm to apply them. The invocation of each rule is determined by its precondition.

Figure 2 presents the high-level algorithm for transforming each use case of a use case model (an instance of UCMeta) into sequence diagrams contained in the corresponding analysis model (an instance of the UML 2.0 metamodel). The algorithm is formalized using pseudocode where some variables are typed as model elements in UCMeta (prefixed with ucm::) and the UML Interaction metamodel (prefixed with uml::), which are highlighted in courier new font.

The Transform function has two input parameters: the use case model being transformed into sequence diagrams (instance of metaclass UseCaseModel of UCMeta: ucModel) and the UML analysis model to contain sequence diagrams to be generated (instance of metaclass Model of the UML 2 metamodel). The function outputs the refined UML analysis model (umlModel) containing the generated sequence diagrams for each use case of the use case model.

5.3 Traceability

We establish two sets of traceability links during the transformation from a textual UCMod to sequence diagrams: from the UCMod to the instance of UCMeta; from the UCMeta instance to the automatically generated sequence diagrams. If necessary, direct traceability links from the textual UCMod to the sequence diagrams can be derived from these two sets.

Traceability links from UCMod to UCMeta link the fields of the use case template used to document textual UCSs to instances of the corresponding metaclasses in UCMeta. For example, field Brief
Description of the use case template is linked to an instance of metaclass UCMeta. A sentence in the brief description is then linked to an instance of metaclass Sentence of UCMeta.

Regarding the second set of traceability links, UCMeta metaclass instances are linked to corresponding model elements in the UML Interaction metamodel based on our transformation rules. For example, we establish a traceability link between a complex sentence (e.g., Basic flow step 2: Conditional Sentence of the UCMeta instance) and its corresponding model elements in the sequence diagram (e.g., the “alt” combined fragment [IF The system recognizes the ATM card], message 1 and 2 of the first operand of the combined fragment) generated during the transformation when rule 2.2 is invoked.

5 Automation

Our approach has been implemented as part of aToucan [7]. aToucan aims to automatically transform requirements given as a UCM in RUCM into a UML analysis model including a class diagram, and a set of sequence and activity diagrams. aToucan relies on a number of existing technologies. It is built as an Eclipse plug-in, using the Eclipse development platform. UCMeta is implemented as an Ecore model, using Eclipse EMF [23], which generates code as Eclipse plug-ins. The Stanford (Java) Parser [24] is used as a NL parser. It generates a syntactic parse tree for a sentence and the sentence’s grammatical dependencies (e.g., subject, direct object). The generation of the UML analysis model relies on Kermeta [22], a metamodeling language also built on top of the Eclipse platform and EMF. The target UML analysis model is instantiated using the Eclipse UML2 project, which is an EMF-Based implementation of the UML 2 standard.

The architecture of aToucan is easy to extend and can accommodate certain types of changes. Transformation rules for generating different types of diagrams are structured into different packages to facilitate their modifications and extensions. Thanks to the generation of an Eclipse UML2 analysis model, generated UML models can be imported and visualized by many open source and commercial tools. Similarly, though UCSs are currently provided as text files, a specific package to import UCSs allows integration with open source and commercial requirement management tools. More details on the design of aToucan can be found in [7].

We adapted the traceability model proposed in the traceability component (fr.irisa.triskell.traceability.model) of Kermeta [22] to establish traceability links. Details of the traceability model is discussed in [7].

6 Evaluation

We used six different software system descriptions to evaluate our approach. In the rest of this section, they are referred to as ATM (Automated Teller Machine), Elevator, ARENA (tournament management system), FRIEND (First Responder Interactive Emergency Navigational Database), CPD (Car Par Dealer system), and VS (Video Store system). These systems come from different sources. ATM and Elevator are called Banking System and Elevator System in [15], respectively, and we have their UCMods, the corresponding sequence diagrams for ATM and collaboration diagrams for Elevator. We can consider these artifacts to be created by experts (e.g., renowned textbook authors) and therefore we can use them as a reference for evaluation purposes. Since UML 1.4 collaboration diagrams are equivalent to sequence diagrams, we mapped the ones for Elevator [15] into their equivalent sequence diagrams, and used these as a reference for our evaluation purpose. ARENA and FRIENDS come from [25], where use cases Announce Tournament and Report Emergency, and corresponding analysis sequence diagrams are described. The UCMods of CPD and VS were created by Masters students and the sequence diagrams were automatically generated by aToucan. These two systems are used to compare generated sequence diagrams with those manually created by trained 4th year undergraduate students in laboratory conditions. Since the UCSs of these systems come from different sources, they were re-written by applying RUCM. Table 3 summarizes some characteristics of each case study system: use cases, the number of messages, lifelines, interaction use (IU), and combined fragments (CF) of each reference sequence diagram.
Recall that aToucan generates traceability links between a textual use case model and automatically generated sequence diagrams. We did not evaluate this aspect since the models we have for the four systems do not have this information.

In the rest of the section, we first discuss the design of the case studies (Section 6.1). In Section 6.2, the evaluation measurement is defined and justified, followed by the discussion of evaluation analysis and results (Section 6.3). The sequence diagrams automatically generated by aToucan for all these six case studies are provided in Appendix B.

### Table 3 Characteristics of each case study system - sequence diagrams

<table>
<thead>
<tr>
<th>Case studies</th>
<th>Use cases</th>
<th># of messages</th>
<th># of lifelines</th>
<th># of IU</th>
<th># of CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATM</td>
<td>Validate PIN</td>
<td>15</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Elevator</td>
<td>Withdraw Funds</td>
<td>27</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Select Destination</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ARENA</td>
<td>Request Elevator</td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FRIEND</td>
<td>Announce Tournament</td>
<td>30</td>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPD</td>
<td>Report Emergency</td>
<td>20</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VS</td>
<td>Create Customer Order</td>
<td>13</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Order Part</td>
<td>14</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Insufficient Stock</td>
<td>16</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Rent Video</td>
<td>27</td>
<td>11</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Check Database</td>
<td>33</td>
<td>12</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

### 6.1 Design of study

We aim to answer two research questions: How do sequence diagrams created by aToucan compare to 1) sequence diagrams created created by an expert (ATM, Elevator, ARENA, and FRIEND), and 2) sequence diagrams created by trained 4th year undergraduate students (CPD and VS)? To answer the first question, we compare sequence diagrams generated with aToucan with sequence diagrams manually devised by experts in terms of the evaluation measurement in Section 6.2. These diagrams were used as a basis for the assessment as they are considered correct and complete, and therefore referred to as reference diagrams. To answer the second question, we compare the sequence diagrams manually created by 4th year undergraduate students in course labs with the ones automatically generated by aToucan (reference diagrams).

### 6.2 Evaluation measurement

Five measures evaluate the quality of a sequence diagram (SD): SD Completeness, SD Consistency, SD Redundancy, BCE Consistency, and SDCD Consistency. They are defined (Table 5) based on simpler measures (Table 4). Data are collected from the reference sequence diagrams to compute the first four measures in Table 4; while data are collected from the sequence diagrams being evaluated for the last 14 measures.

### Table 4 Collected sequence diagram data

<table>
<thead>
<tr>
<th>#</th>
<th>Measure</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( N_{m_{ref}} )</td>
<td># of messages in the reference sequence diagram</td>
</tr>
<tr>
<td>2</td>
<td>( N_{l_{ref}} )</td>
<td># of lifelines in the reference sequence diagram</td>
</tr>
<tr>
<td>3</td>
<td>( N_{IU_{ref}} )</td>
<td># of interaction uses in the reference sequence diagram</td>
</tr>
<tr>
<td>4</td>
<td>( N_{CF_{ref}} )</td>
<td># of combined fragments in the reference sequence diagram</td>
</tr>
<tr>
<td>5</td>
<td>( N_{m_{test}} )</td>
<td># of messages in a tested sequence diagram</td>
</tr>
<tr>
<td>6</td>
<td>( N_{l_{test}} )</td>
<td># of lifelines in a tested sequence diagram</td>
</tr>
<tr>
<td>7</td>
<td>( N_{IU_{test}} )</td>
<td># of interaction uses in a tested sequence diagram</td>
</tr>
<tr>
<td>8</td>
<td>( N_{CF_{test}} )</td>
<td># of combined fragments in a tested sequence diagram</td>
</tr>
<tr>
<td>9</td>
<td>( N_{msg_{test}} )</td>
<td># of missing messages in a tested sequence diagram</td>
</tr>
<tr>
<td>10</td>
<td>( N_{CF_{test}} )</td>
<td># of missing combined fragments in a tested sequence diagram</td>
</tr>
<tr>
<td>11</td>
<td>( N_{IU_{test}} )</td>
<td># of missing interaction uses in a tested sequence diagram</td>
</tr>
<tr>
<td>12</td>
<td>( N_{inc_{test}} )</td>
<td># of occurrences of incorrect messages (e.g., incorrect message name, incorrect lifelines, wrong direction, and/or incorrect message types) in a tested sequence diagram</td>
</tr>
<tr>
<td>13</td>
<td>( N_{incr_{test}} )</td>
<td># of occurrences of incorrectly-applied interaction uses in a tested sequence diagram</td>
</tr>
<tr>
<td>14</td>
<td>( N_{CF_{test}} )</td>
<td># of occurrences of incorrectly-applied combined fragments</td>
</tr>
</tbody>
</table>

1. Interaction operand is not provided or a wrong operand is used.
2. Condition is not provided or a wrong condition is used.
3. Lifelines are not correctly covered.
4. Wrong messages are covered.
SDCD Completeness. \( R_{SDCD} \) evaluates whether a student’s sequence diagram conforms to the Boundary-Control-Entity (BCE) design principle of analysis sequence diagrams [2]. Any violation of the following three rules is considered to be a violation of the BCE principle and therefore contributes to measure \( N_{vBCE} \) (Table 4): 1) A message is sent from a Boundary object to an Entity object. 2) A message is sent from an Entity object to a Control object. 3) A message is sent from an Entity object to a Control object.

As shown in Table 5, the completeness of a sequence diagram (\( R_{SDcompl} \)) is computed as the average of the Message Completeness (\( R_{comMsg} \)), IU (InteractionUse) Completeness (\( R_{corIU} \)), and CF (CombinedFragment) Completeness (\( R_{corCF} \)). The consistency of a sequence diagram (\( R_{SDcorrect} \)) is determined by the consistency of messages (\( R_{comMsg} \)), interaction uses (\( R_{corIU} \)), combined fragments (\( R_{corCF} \)), and message ordering (\( R_{corSeq} \)) since these are considered to be important aspects in a sequence diagram. The redundancy of a sequence diagram (\( R_{SDr} \)) is computed as the ratio of redundant messages (\( N_{reMsg-t} \)) over all the messages of a student’s sequence diagram (\( N_{msg-t} \)). Measure \( BCE \) Consistency (\( R_{BCE} \)) evaluates whether a student’s sequence diagram conforms to the Boundary-Control-Entity (BCE) design principle of analysis sequence diagrams [2]. Any violation of the following three rules is considered to be a violation of the BCE principle and therefore contributes to measure \( N_{vBCE} \) (Table 4): 1) A message is sent from a Boundary object to an Entity object. 2) A message is sent from an Entity object to a Control object. 3) A message is sent from an Entity object to a Boundary object.

6.3 Evaluation results and analysis

Results from comparing the sequence diagrams generated by \( aToucan \) with experts’ solutions are provided in Table 6, where columns list our evaluation criteria: Message Completeness, Message Completeness, and Message Redundancy. Notice that the sequence diagrams created by experts (in textbooks) do not fully conform to UML 2 sequence diagram (i.e., no interaction uses and combined fragments were derived). These diagrams also do not comply with the BCE design principle (Section 6.2). For these four case studies, we therefore cannot assess the diagrams generated by \( aToucan \) against experts’ solutions for the following measures: IU Completeness, IU Completeness, CF Completeness, CF Completeness, and BCE Consistency. As a result not all measures defined in Table 5 can be used for comparison. Besides, textbook sequence diagrams are not consistent with their corresponding class diagrams and, therefore, \( aToucan \) cannot be evaluated against the expert-derived sequence diagrams in terms of SCD Consistency. \( aToucan \) generated sequence diagrams have the following characteristics:
1) they fully comply with the BCE principle (100% BCE Consistency) and 2) they are fully consistent with their corresponding class diagrams (100% SDCD Consistency).

As shown in the last row of Table 6, on average, when compared with experts, aToucan achieved very high message consistency (91%) and completeness (97%), 100% message ordering consistency, and near 0% SD redundancy. aToucan achieved relatively low message consistency for two use cases of the Elevator system: 75% and 83% for use cases Select Destination and Request Elevator, respectively. A careful investigation showed that the collaboration diagrams in the source textbook do not fully conform to their UCMods. The collaboration diagrams are very coarse-grained and some steps described in the UCMods are not realized in the collaboration diagrams. However, the sequence diagrams automatically generated by aToucan fully correspond to the UCMods and are therefore not fully consistent with the textbook collaboration diagrams.

Table 6 Evaluation results of aToucan against experts

<table>
<thead>
<tr>
<th>Systems</th>
<th>Use cases</th>
<th>Message Consistency</th>
<th>Message Completeness</th>
<th>Message Ordering Consistency</th>
<th>SD Redundancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARENA</td>
<td>Announce Tournament</td>
<td>0.99</td>
<td>0.87</td>
<td>1.00</td>
<td>0.03</td>
</tr>
<tr>
<td>FRIEND</td>
<td>Elevator</td>
<td>0.99</td>
<td>0.93</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ATM</td>
<td>Report Emergency</td>
<td>0.75</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ATM</td>
<td>Select Destination</td>
<td>0.83</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ATM</td>
<td>Request Elevator</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ATM</td>
<td>Validate PIN</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ATM</td>
<td>Withdraw Funds</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>0.91</td>
<td>0.97</td>
<td>1.00</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 7 presents a summary of the comparison of aToucan-generated sequence diagrams with the ones from trained 4th year undergraduate students for two systems: CPD and VS. During a three-hour experiment, the students were asked to derive sequence diagrams for the CPD and VS system use cases of Table 3. Some students were not able to derive all required sequence diagrams and we decided to discard their observations in order to obtain comparable data, i.e., data based on the same sets of sequence diagrams. Eventually we obtained samples of 17 and 13 observations for CPD and VS, respectively. Table 7 shows the mean values of these two samples with respect to different criteria. For example, the messages of the sequence diagrams manually derived by students for the systems CPD and VS are on average 78% and 73% consistent with the messages of the sequence diagrams automatically generated by aToucan, respectively. Recall (from Section 6.2) that the measure SD Consistency is computed as the consistency average for Message, IU, CF, and Message Ordering. From Table 7, we can observe that the students only achieved 52% and 45% Message Completeness for CPD and VS, respectively. In other words, on average, nearly half of the messages of aToucan-generated sequence diagrams were not created by students. Due to the clear specifications of sequential steps in flows of events in the use cases, students achieved 99% Message Ordering Consistency for both systems. For a similar reason, the sequence diagrams manually derived by students contain zero message redundancy. Results in Table 7 clearly suggest that trained students are unlikely to perform as well as aToucan when deriving sequence diagrams from use case specifications.

Table 7 Evaluation results of 4th students against aToucan

<table>
<thead>
<tr>
<th>Systems</th>
<th>Consistency</th>
<th>Completeness</th>
<th>Redundancy</th>
<th>BCE Consistency</th>
<th>SDCD Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPD</td>
<td>0.78</td>
<td>0.11</td>
<td>0.62</td>
<td>0.99</td>
<td>0.85</td>
</tr>
<tr>
<td>VS</td>
<td>0.73</td>
<td>1</td>
<td>0.39</td>
<td>0.99</td>
<td>0.95</td>
</tr>
</tbody>
</table>

7 Conclusion

As an important step of model-driven development, realizing use cases by sequence diagrams ensures that the dynamic behavior of a system is described in a precise and visual manner. However, this activity heavily relies on heuristics [2] and manually deriving sequence diagrams is an error-prone process as suggested by experiments: trained 4th year undergraduate students only created half of the required messages and around 75% of created messages were consistent with the reference diagrams.

A use case modeling approach (RUCM) was proposed in [6] and was used in this paper to automatically generate sequence diagrams from textual use case descriptions. This work is part of the aToucan approach [7], which aims to transform a use case model produced with RUCM into an initial
UML analysis model that includes class [7], activity [8] and sequence diagrams. Generating class diagrams [7] is a prerequisite step for generating sequence diagrams. Automatically generated sequence diagrams are consistent with class and activity diagrams generated from the same use case model. A set of systematically derived transformation, the corresponding algorithms and traceability support are proposed in this paper.

aToucan is fully automated and our prototype tool is built on the top of a set of advanced natural language and model engineering technologies such as the Stanford (Java) Parser [24], Eclipse EMF [23], and Kermeta [22].

Sequence diagram generation has been assessed, from different aspects, on six case studies, four of which have been used to evaluate sequence diagrams generated by aToucan against the ones devised by experts. Results indicate that, sequence diagrams generated by our approach are highly consistent with the ones manually devised by experts, and are also very complete. Sequence diagrams generated by aToucan are also far more complete than the ones manually generated by 30 fourth-year, trained undergraduate students. Our future work includes evaluating aToucan with more and larger case studies.

8 Acknowledgement

The work of Lionel Briand and Tao Yue was supported by a grant from Det Norske Veritas, Norway, in the context of the ModelME! project.

9 References

24. The Stanford Natural Language Processing Group. The Stanford Parser version 1.6
Appendix A

Transformation rules are formalized in Kermeta language [22].

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Rule type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Create an Interaction (UML metaclass) for each use case.</td>
<td>Composite</td>
</tr>
<tr>
<td></td>
<td><strong>Transformation</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>operation transform_0(uc : ucm::UseCase) : Void</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pre</td>
<td></td>
</tr>
<tr>
<td></td>
<td>uc != void and trace_helper.findTargetInteraction(uc) != void</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>var message : traceability::Message</td>
<td></td>
</tr>
<tr>
<td></td>
<td>message := trace_helper.createMessage(&quot;0&quot;, &quot;uc -&gt; interaction&quot;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>var interaction : uml::Interaction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>var basicFlow : BasicFlow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>var altFlows : Sequence&lt;AlternativeFlow&gt; init Sequence&lt;AlternativeFlow&gt;.new</td>
<td></td>
</tr>
<tr>
<td></td>
<td>var bf_steps : OrderedSet&lt;Sentence&gt; init OrderedSet&lt;Sentence&gt;.new</td>
<td></td>
</tr>
<tr>
<td></td>
<td>interaction := trace_helper.findTargetInteraction(uc)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>basicFlow := uc.specification.flows.select{f</td>
<td>f.isInstanceOf(BasicFlow)}.one.asType(BasicFlow)</td>
</tr>
<tr>
<td></td>
<td>uc.specification.flows.select{f</td>
<td>f.isInstanceOf(AlternativeFlow)}.each</td>
</tr>
<tr>
<td></td>
<td>{alt</td>
<td>altFlows.add(alt.asType(AlternativeFlow))}</td>
</tr>
<tr>
<td></td>
<td>bf_steps := basicFlow.steps</td>
<td></td>
</tr>
<tr>
<td></td>
<td>var globalOperand : uml::InteractionOperand</td>
<td></td>
</tr>
<tr>
<td></td>
<td>altFlows.each{alt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if alt.isInstanceOf(GlobalAlternative) then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transform_4(globalOperand, alt, interaction, uc, void)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bf_steps.each{s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if s.isInstanceOf(SimpleSentence) then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transform_1(globalOperand, s.asType(SimpleSentence), interaction, uc, void)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if s.isInstanceOf(ComplexSentence) then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transform_2(globalOperand, s.asType(ComplexSentence), interaction, uc, void)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if s.isInstanceOf(SpecialSentence) then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transform_3(globalOperand, s.asType(SpecialSentence), interaction, uc, void)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Rule type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Invoke rules 1.1-1.6 to process seven (of eight) types of simple sentence.</td>
<td>Composite</td>
</tr>
<tr>
<td></td>
<td><strong>Transformation</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>operation transform_1(globalOperand : uml::InteractionOperand, ss :</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SimpleSentence, interaction : uml::Interaction, uc : ucm::UseCase,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>enclosingOperand : InteractionOperand) : Void</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pre</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ss != void</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>var message : traceability::Message</td>
<td></td>
</tr>
<tr>
<td></td>
<td>message := trace_helper.createMessage(&quot;1&quot;, &quot;SimpleSentence -&gt; Message&quot;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if ss.pattern != void then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if ss.pattern.isInstanceOf(SV) then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transform_1_1(globalOperand, ss, interaction, uc, enclosingOperand)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
### ID
1.1

**Description**
InternalTransaction—generate a self-message at the system lifeline.

**Rule type**
Atomic

**Transformation**

```
if ss.pattern.isInstanceOf(SVC) then
    transform_1_2(globalOperand, ss, interaction, uc, enclosingOperand)
end
if ss.pattern.isInstanceOf(SVCC) then
    transform_1_3(globalOperand, ss, interaction, uc, enclosingOperand)
end
if ss.pattern.isInstanceOf(SVDO) then
    transform_1_4(globalOperand, ss, interaction, uc, enclosingOperand)
end
if ss.pattern.isInstanceOf(SVDOC) then
    transform_1_5(globalOperand, ss, interaction, uc, enclosingOperand)
end
if ss.pattern.isInstanceOf(SVIDO) or ss.pattern.isInstanceOf(SVIDOCC) then
    transform_1_6(globalOperand, ss, interaction, uc, enclosingOperand)
end
end
```

<table>
<thead>
<tr>
<th>ID</th>
<th>1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>InternalTransaction—generate a self-message at the system lifeline.</td>
</tr>
<tr>
<td><strong>Rule type</strong></td>
<td>Atomic</td>
</tr>
</tbody>
</table>
| **Transformation** | ```
operation transform_1_1(globalOperand : uml::InteractionOperand, sen : SimpleSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void
pre precondition is do
    sen !="void" and sen.pattern.isInstanceOf(SV)
end
is do
    var message : traceability::Message
    message := trace_helper.createMessage("1_1", "SV -> Message")
    var umlMessage : uml::Message
    var subject : uml::Class
    var subject_np : NounPhrase
    var opr : uml::Operation
    if sen.subject != "void" then
        if sen.subject.form != "void" then
            subject_np := sen.subject.form.asType(NounPhrase)
        end
        if subject_np != "void" then
            if not subject_np.sysPosition.equals(UseCaseNL::NPSystemPositionType::NoneSys) then
                subject := trace_helper.findTargetClass(uc, "void")
            else
                subject := trace_helper.findTargetClass(subject_np, "void")
            end
            if subject != "void" then
                subject := trace_helper.findTargetClass(subject_np.head, "void")
            end
            opr := subject.ownedOperation.select{o| o.name.toLowerCase().equals(sen.predicate.description.toLowerCase)}.one
            if opr != "void" then
                opr := target_model_helper.generateOperation(subject, sen.predicate.description)
            end
            if opr != "void" then
                var lifeline : uml::Lifeline
                lifeline := interaction.lifeline.select{l|l.represents.type.equals(subject)}.one
                if lifeline != "void" then
                    lifeline := generateLifeline(interaction, subject)
                end
                umlMessage := generateMessage(globalOperand, opr.name, interaction, lifeline, "void", opr, enclosingOperand)
            end
end
``` |
<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td><strong>Initiation</strong>—generate two sequential messages: actor lifeline (subject) $\rightarrow$ actor boundary class lifeline $\rightarrow$ system lifeline (complement). <strong>InternalTransaction</strong>—generate a self-message for system lifeline. <strong>ResponseToPrimaryActor/ResponseToSecondaryActor</strong>—generate two sequential messages: system lifeline (subject) $\rightarrow$ boundary class lifeline of actor (complement) $\rightarrow$ the actor.</td>
</tr>
</tbody>
</table>

| Rule type | Atomic |

<table>
<thead>
<tr>
<th>Transformation</th>
<th>operation transform_1_2(globalOperand : uml::InteractionOperand, sen : SimpleSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre</td>
<td>precondition is do</td>
</tr>
<tr>
<td></td>
<td>sen != void and sen.pattern.isInstanceOf(SVC) and sen.transactionType != void end</td>
</tr>
<tr>
<td>is do</td>
<td>var message : traceability::Message</td>
</tr>
<tr>
<td></td>
<td>message := trace_helper.createMessage(&quot;1_2&quot;, &quot;SVC =&gt; Message&quot;)</td>
</tr>
<tr>
<td></td>
<td>var umlMessages : Set<a href="">uml::Message</a> init Set<a href="">uml::Message</a>.new</td>
</tr>
<tr>
<td>if</td>
<td>sen.transactionType.isInstanceOf(Initiation) then</td>
</tr>
<tr>
<td></td>
<td>var subject : uml::Class</td>
</tr>
<tr>
<td></td>
<td>var actor : ucm::Actor</td>
</tr>
<tr>
<td></td>
<td>var theSystem : uml::Class</td>
</tr>
<tr>
<td></td>
<td>if sen.subject.isInstanceOf(ActorSubj) then</td>
</tr>
<tr>
<td></td>
<td>actor := sen.subject.asType(ActorSubj).actor</td>
</tr>
<tr>
<td></td>
<td>subject := trace_helper.findTargetClass(actor, false)</td>
</tr>
<tr>
<td></td>
<td>theSystem := trace_helper.findTargetClass(uc, void)</td>
</tr>
<tr>
<td></td>
<td>if subject != void and theSystem != void then</td>
</tr>
<tr>
<td></td>
<td>umlMessages.addAll(generateMsgsActorBoundarySystem(globalOperand, interaction, sen.predicate.description, actor, subject, theSystem, enclosingOperand))</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>if</td>
<td>sen.transactionType.isInstanceOf(InternalTransaction) then</td>
</tr>
<tr>
<td></td>
<td>var subject : uml::Class</td>
</tr>
<tr>
<td></td>
<td>var opr : uml::Operation</td>
</tr>
<tr>
<td></td>
<td>subject := trace_helper.findTargetClass(uc, void)</td>
</tr>
<tr>
<td></td>
<td>if subject != void then</td>
</tr>
<tr>
<td></td>
<td>opr := subject.ownedOperation.select{o</td>
</tr>
<tr>
<td></td>
<td>o.name.toLowerCase.equals(sen.predicate.description.toLowerCase)}.one</td>
</tr>
<tr>
<td></td>
<td>if opr == void then opr := target_model_helper.generateOperation(subject, sen.predicate.description) end</td>
</tr>
<tr>
<td></td>
<td>if opr != void then</td>
</tr>
<tr>
<td></td>
<td>var lifeline : uml::Lifeline</td>
</tr>
<tr>
<td></td>
<td>lifeline := interaction.lifeline.select{l</td>
</tr>
<tr>
<td></td>
<td>equals(subject)}.one</td>
</tr>
<tr>
<td></td>
<td>if lifeline == void then lifeline := generateLifeline(interaction, subject)</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>umlMessages.add(generateMessage(globalOperand, opr.name, interaction, lifeline, void, opr, enclosingOperand))</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>if</td>
<td>sen.transactionType.isInstanceOf(ResponseToPrimaryActor) or</td>
</tr>
<tr>
<td></td>
<td>sen.transactionType.isInstanceOf(ResponseToSecondaryActor) then</td>
</tr>
<tr>
<td></td>
<td>var subject : uml::Class</td>
</tr>
<tr>
<td></td>
<td>var actor : ucm::Actor</td>
</tr>
<tr>
<td></td>
<td>var ppObj : uml::Class</td>
</tr>
<tr>
<td></td>
<td>var opr : uml::Operation</td>
</tr>
<tr>
<td></td>
<td>subject := trace_helper.findTargetClass(uc, void)</td>
</tr>
</tbody>
</table>
```java
if sen.predicate.complements.select{o|o.isInstanceOf(ObjPPComplt)}.one.
   asType(ObjPPComplt).toActor != void then
   actor := sen.predicate.complements.select{o|o.isInstanceOf(ObjPPComplt)}.
   one.asType(ObjPPComplt).toActor
   ppObj := trace_helper.findTargetClass(actor, false)
end
if subject != void and ppObj != void then
   var opr_name : String
   opr_name := sen.predicate.complements.select{c|c.isInstanceOf
   (ObjInfinitiveComplt)}.one.description.substring(3, sen.
   predicate.complements.select{c|c.isInstanceOf(ObjInfinitiveComplt)}.
   one.description.size())
   opr := ppObj.ownedOperation.select{o| o.name.toLowerCase.
       equals(opr_name.toLowerCase)}.one
   if opr == void then opr := target_model_helper.generateOperation
       (ppObj, opr_name) end
   if opr != void then
      umlMessages.addAll(generateMsgsSystemBoundaryActor(globalOperand,
               interaction, opr, actor, ppObj, subject, enclosingOperand))
   end
end
if umlMessages.size >0 then
   umlMessages.each{msg|trace_helper.createTLink(sen, msg, message)}
end
end
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Rule type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3</td>
<td>Initiation—generate two sequential messages: actor lifeline (subject) → actor boundary class lifeline → system lifeline (one of the complements). InternalTransaction—generate a self-message for system lifeline. ResponseToPrimaryActor/ResponseToSecondaryActor—generate two sequential messages: system lifeline (subject) → boundary class lifeline of actor (one of the two object complements) → the actor.</td>
<td>Atomic</td>
</tr>
<tr>
<td>Operation transform_1_3(globalOperand : uml::InteractionOperand, sen : SimpleSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pre</td>
<td>precondition is do</td>
<td></td>
</tr>
<tr>
<td>sen != void and sen.pattern.isInstanceOf(SVCC) and sen.transactionType != void</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is do</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var message : traceability::Message</td>
<td></td>
<td></td>
</tr>
<tr>
<td>message := trace_helper.createMessage(&quot;1_3&quot;, &quot;SVCC -&gt; Message&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var umlMessages : Set<a href="">uml::Message</a> init Set<a href="">uml::Message</a>::new</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if sen.transactionType.isInstanceOf(Initiation) then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var subject : uml::Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var actor : ucm::Actor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var theSystem : uml::Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if sen.subject.isInstanceOf(ActorSubj) then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>actor := sen.subject.asType(ActorSubj).actor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subject := trace_helper.findTargetClass(actor, false)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>theSystem := trace_helper.findTargetClass(uc, void)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if subject != void and theSystem != void then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>umlMessages.addAll(generateMsgsActorBoundarySystem(globalOperand,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interaction, sen.predicate.description, actor, subject, theSystem,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>enclosingOperand))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if sen.transactionType.isInstanceOf(InternalTransaction) then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var subject : uml::Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var opr : uml::Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subject := trace_helper.findTargetClass(uc, void)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if subject != void then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opr := subject.ownedOperation.select(o</td>
<td>o.name.</td>
<td></td>
</tr>
</tbody>
</table>
toLowerCase.equals(sen.predicate.description.toLowerCase()).one

if opr == void then opr := target_model_helper.generateOperation(subject, sen.predicate.description) end
if opr != void then
    var lifeline : uml::Lifeline
    lifeline := interaction.lifeline.select{l|l.represents.type.equals(subject)}.one
    if lifeline == void then lifeline := generateLifeline(interaction, subject) end
    umlMessages.add(generateMessage(globalOperand, opr.name, interaction, lifeline, void, opr, enclosingOperand))
end
end

if sen.transactionType.isInstanceOf(ResponseToPrimaryActor) or sen.transactionType.isInstanceOf(ResponseToSecondaryActor) then
    var subject : uml::Class
    var actor : ucm::Actor
    var ppObj : uml::Class
    var opr : uml::Operation
    subject := trace_helper.findTargetClass(uc, void)
    if sen.predicate.complements.select{o|o.isInstanceOf(ObjPPComplt)}.one.asType(ObjPPComplt).toActor != void then
        actor := sen.predicate.complements.select{o|o.isInstanceOf(ObjPPComplt)}.one.asType(ObjPPComplt).toActor
        ppObj := trace_helper.findTargetClass(actor, false)
    end
    if subject != void and ppObj != void then
        var opr_name : String
        opr_name := sen.predicate.complements.select{o|o.isInstanceOf(ObjInfinitiveComplt)}.one.description.substring(3, sen.predicate.complements.select{o|o.isInstanceOf(ObjInfinitiveComplt)}.one.description.size())
        opr := ppObj.ownedOperation.select{o|o.name.toLowerCase.equals(opr_name.toLowerCase)}.one
        if opr == void then opr := target_model_helper.generateOperation(ppObj, opr_name) end
        if opr != void then
            umlMessages.addAll(generateMsgsSystemBoundaryActor(globalOperand, interaction, opr, actor, ppObj, subject, enclosingOperand))
        end
    end
end

if umlMessages.size > 0 then
    umlMessages.each{msg|trace_helper.createTLink(sen, msg, message)}
end

<table>
<thead>
<tr>
<th>ID</th>
<th>1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Initiation—generate two sequential messages: actor lifeline (subject) (\rightarrow) actor boundary class lifeline (\rightarrow) system lifeline (direct object). InternalTransaction—generate a message from the system lifeline to the object class lifeline. ResponseToPrimaryActor/ResponseToSecondaryActor—generate two sequential messages: the system lifeline (the subject) (\rightarrow) the boundary class lifeline of the actor (the object) (\rightarrow) the actor lifeline.</td>
</tr>
<tr>
<td>Rule type</td>
<td>Atomic</td>
</tr>
<tr>
<td>Transformation</td>
<td>operation transform_1_4(globalOperand : uml::InteractionOperand, sen : SimpleSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void</td>
</tr>
<tr>
<td>precondition is do</td>
<td>sen != void and sen.pattern.isInstanceOf(SVDO) and sen.transactionType != void end</td>
</tr>
<tr>
<td>is do</td>
<td>var message : traceability::Message message := trace_helper.createMessage(&quot;1 4&quot;, &quot;SVDO -&gt; Message&quot;)</td>
</tr>
</tbody>
</table>
var umlMessages : Set<uml::Message> init Set<uml::Message>.new
if sen.transactionType.isInstanceOf(Initiation) then
  var subject : uml::Class
  var object : uml::Class
  var actor : ucm::Actor
  var opr1 : uml::Operation
  var opr2 : uml::Operation
  if sen.subject.isInstanceOf(ActorSubj) then
    actor := sen.subject.asType(ActorSubj).actor
    subject := trace_helper.findTargetClass(actor, false)
  end
  if sen.predicate.objects.select{o|o.isInstanceOf(DirectObject)}.one.isInstanceOf(DirectSysObj) then
    object := trace_helper.findTargetClass(uc, void)
  end
  if subject != void and object != void then
    umlMessages.addAll(generateMsgsActorBoundarySystem(globalOperand, 
      interaction, sen.predicate.description, actor, subject, object, 
      enclosingOperand))
  end
end
if sen.transactionType.isInstanceOf(InternalTransaction) then
  var subject : uml::Class
  var object : uml::Class
  var opr : uml::Operation
  subject := trace_helper.findTargetClass(uc, void)
  if trace_helper.findTargetClass(sen.predicate.objects.one, void) != void then
    object := trace_helper.findTargetClass(sen.predicate.objects.
      select{o|o.isInstanceOf(DirectObject)}.one, void)
  else if trace_helper.findTargetClass(sen.predicate.objects.
    select{o|o.isInstanceOf(DirectObject)}.one.form.asType(NounPhrase).head, 
      void) != void then
    object := trace_helper.findTargetClass(sen.predicate.objects.
      select{o|o.isInstanceOf(DirectObject)}.one.form.
        asType(NounPhrase).head, void)
  else if sen.predicate.objects.select{o|o.isInstanceOf(DirectObject)}.one.form.asType(NounPhrase).preHeadString.
    select{phs|phs.isInstanceOf(Noun)}.size>0 then
    object := trace_helper.findTargetClass(sen.predicate.objects.
      select{o|o.isInstanceOf(DirectObject)}.one.form.asType(NounPhrase).
        preHeadString.select{phs|phs.isInstanceOf(Noun)}.one, void)
  end
end
if subject != void and object != void then
  opr := subject.ownedOperation.select{o|
    o.name.toLowerCase.equals(sen.predicate.description.toLowerCase)}.one
  if opr == void then opr := target_model_helper.generateOperation(subject, 
    sen.predicate.description) end
var lifelineL : uml::Lifeline
lifelineL := interaction.lifeline.select{l|l.represents.type.
  equals(subject)}.one
if lifelineL == void then lifelineL := generateLifeline(interaction, 
  subject) end
if sen.predicate.predicator.isInstanceOf(CreateActionVerb) then
  generateCreateMessage(globalOperand, interaction, lifelineL, object, opr, 
    enclosingOperand)
else
  var lifelineR : uml::Lifeline
  lifelineR := interaction.lifeline.select{l|l.represents.type.
    equals(object)}.one
  var opr1 : uml::Operation
  opr1 := object.ownedOperation.select{o|
    o.name.toLowerCase.equals(sen.predicate.description.toLowerCase)}.one
  if lifelineR == void then
    generateCreateMessage(globalOperand, interaction, lifelineL, object, 
      opr, enclosingOperand)
  if opr1 == void then opr1 := target_model_helper.generateOperation
(object, sen.predicate.description) end
lifelineR := interaction.lifeline.select{l|l.represents.type.
  equals(object)}.one
if lifelineR != void then generateMessage(globalOperand, opr.name,
  interaction, lifelineL, lifelineR, opr1, enclosingOperand) end
else
  if opr1 == void then opr1 := target_model_helper.generateOperation
    (object, sen.predicate.description) end
  umlMessages.add(generateMessage(globalOperand, opr.name, interaction,
    lifelineL, lifelineR, opr1, enclosingOperand))
end
end
end
end

if sen.transactionType.isInstanceOf(ResponseToPrimaryActor)
or
  sen.transactionType.isInstanceOf(ResponseToSecondaryActor) then
  var subject : uml::Class
  var object : uml::Class
  var opr : uml::Operation
  var actor : ucm::Actor
  subject := trace_helper.findTargetClass(uc, void)
  if sen.predicate.objects.one.asType(DirectObject).form.
    isInstanceOf(NounPhrase) then
    if sen.predicate.objects.one.asType(DirectObject).form.
      asType(NounPhrase).containedActor != void then
      actor := sen.predicate.objects.one.asType(DirectObject).form.
        asType(NounPhrase).containedActor
      object := trace_helper.findTargetClass(actor, false)
    end
  end
  if subject != void and object != void then
    opr := object.ownedOperation.select{o| o.name.toLowerCase.equals
      (sen.predicate.description.toLowerCase).one
    if opr == void then opr := target_model_helper.generateOperation(object,
      sen.predicate.description) end
    if opr != void then
      umlMessages.addAll(generateMsgsSystemBoundaryActor(globalOperand,
        interaction, opr, object, subject, enclosingOperand))
    end
  end
end
if umlMessages.size >0 then
  umlMessages.each{msg|trace_helper.createTLink(sen, msg, message)}
end
end
end

<table>
<thead>
<tr>
<th>ID</th>
<th>1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Initiation—generate two sequential messages: actor lifeline (subject) (\rightarrow) actor boundary class lifeline (\rightarrow) the system (direct object) lifeline. InternalTransaction—generate a message from the system lifeline to the object class lifeline. ResponseToPrimaryActor/ResponseToSecondaryActor—generate two sequential messages: the system lifeline (the subject) (\rightarrow) the boundary class lifeline of the actor (the object or the object complement) (\rightarrow) the actor lifeline.</td>
</tr>
<tr>
<td>Rule type</td>
<td>Atomic</td>
</tr>
</tbody>
</table>
| Transformation | Operation transform_1_5(globalOperand : uml::InteractionOperand, sen : SimpleSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void pre
  precondition is do
    sen != void and sen.pattern.isInstanceOf(SVDOC) and sen.transactionType != void
end
  is do
    var message : traceability::Message
    message := trace_helper.createMessage("1_5", "SVDOC -> Message")
    var umlMessages : Set<uml::Message> init Set<uml::Message>.new
    if sen.transactionType.isInstanceOf(Initiation) then
var subject : uml::Class
var object : uml::Class
var actor : ucm::Actor
var objComplt : ObjectComplt
var ppObjClass : uml::Class

if sen.subject.isInstanceOf(ActorSubj) then
    actor := sen.subject.asType(ActorSubj).actor
    subject := trace_helper.findTargetClass(actor, false)
end

if sen.predicate.objects.one.isInstanceOf(DirectSysObj) then
    object := trace_helper.findTargetClass(uc, void)
end

if sen.predicate != void then
    if sen.predicate.complements.size > 0 then
        if sen.predicate.complements.select{c|c.isInstanceOf(ObjPPComplt)}.size > 0 then
            sen.predicate.complements.select{c|c.isInstanceOf(ObjPPComplt)}.each{complt|
                if complt.asType(ObjPPComplt).containingSystem then
                    ppObjClass := trace_helper.findTargetClass(uc, void)
                end
            }
        end
        if sen.predicate.complements.select{c|c.isInstanceOf(ObjInfinitiveComplt)}.size > 0 then
            objComplt := sen.predicate.complements.select{c|c.isInstanceOf(ObjInfinitiveComplt)}.one.asType(ObjInfinitiveComplt)
        end
    end
end

if subject != void and object != void then
    generateMsgsActorBoundarySystem(globalOperand, interaction, sen.predicate.description, actor, subject, object, enclosingOperand)
end

if subject != void and ppObjClass != void then
    umlMessages.addAll(generateMsgsActorBoundarySystem(globalOperand, interaction, sen.predicate.description, actor, subject, ppObjClass, enclosingOperand))
end

if sen.transactionType.isInstanceOf(InternalTransaction) then
    var subject : uml::Class
    var object : uml::Class
    var opr : uml::Operation

    subject := trace_helper.findTargetClass(uc, void)
    if trace_helper.findTargetClass(sen.predicate.objects.one, void) != void then
        object := trace_helper.findTargetClass(sen.predicate.objects.
            select{o|o.isInstanceOf(DirectObject)}.one, void)
    else if trace_helper.findTargetClass(sen.predicate.objects.select
        {o|o.isInstanceOf(DirectObject)}.one.form.asType(NounPhrase).head,
        void) != void then
        object := trace_helper.findTargetClass(sen.predicate.objects.
            select{o|o.isInstanceOf(DirectObject)}.one.form.asType(NounPhrase).head, void)
    else if sen.predicate.objects.select{o|o.isInstanceOf(DirectObject)}.
        one.form.asType(NounPhrase).preHeadString.select{phs|phs.
            isInstanceOf(Noun)}.size > 0 then
        object := trace_helper.findTargetClass(sen.predicate.objects.
            select{o|o.isInstanceOf(DirectObject)}.one.form.asType(NounPhrase).
            preHeadString.select{phs|phs.isInstanceOf(Noun)}.one, void)
    end
end

if subject != void and object != void then
    opr := subject.ownedOperation.select{o|o.name.
        toLowerCase.equals(sen.predicate.description.toLowerCase)}.one
if opr == void then opr := target_model_helper.generateOperation(subject, sen.predicate.description) end

var lifelineL : uml::Lifeline
lifelineL := interaction.lifeline.select{l|l.represents.type. equals(subject)}.one
if lifelineL == void then lifelineL := generateLifeline(interaction, subject) end
if sen.predicate.predicate.form.isInstanceOf(CreateActionVerb) then
umlMessages.add(generateCreateMessage(globalOperand, interaction, lifelineL, object, opr, enclosingOperand))
else
var lifelineR : uml::Lifeline
lifelineR := interaction.lifeline.select{l|l.represents.type. equals(object)}.one
if lifelineR == void then lifelineR := generateLifeline(interaction, object) end
var opr1 : uml::Operation
opr1 := object.ownedOperation.select{o| o.name.toLowerCase.equals(sen.predicate.description.toLowerCase)}.one
if opr1 == void then opr1 := target_model_helper.generateOperation(object, sen.predicate.description) end
umlMessages.add(generateMessage(globalOperand, opr1.name, interaction, lifelineL, lifelineR, opr1, enclosingOperand))
end

end

end if

if sen.transactionType.isInstanceOf(ResponseToPrimaryActor) or sen.transactionType.isInstanceOf(ResponseToSecondaryActor) then
var subject : uml::Class
var object : uml::Class
var actor : ucm::Actor
var opr : uml::Operation

subject := trace_helper.findTargetClass(uc, void)
if sen.predicate.objects.select{o|o.isInstanceOf(DirectObject)}.one.form.asType(NounPhrase).actorPosition != NPActorPositionType.NoneActor then
actor := sen.predicate.objects.select{o|o.isInstanceOf(DirectObject)}.one.form.asType(NounPhrase).containedActor
else
if sen.predicate.complements.size > 0 then
if sen.predicate.complements.select{c|c.isInstanceOf(ObjPPComplt)}.size > 0 then
if sen.predicate.complements.select{c|c.isInstanceOf(ObjPPComplt)}.one.asType(ObjPPComplt).toActor != void then
actor := sen.predicate.complements.select{c|c.isInstanceOf(ObjPPComplt)}.one.asType(ObjPPComplt).toActor
end
end
end
if actor != void then
object := trace_helper.findTargetClass(actor, false)
end
if subject != void and object != void then
opr := object.ownedOperation.select{o| o.name. toLowerCase.equals(sen.predicate.description.toLowerCase)}.one
if opr == void then opr := target_model_helper.generateOperation(object, sen.predicate.description) end
if opr != void then
umlMessages.addAll(generateMsgsSystemBoundaryActor(globalOperand, interaction, opr, actor, object, subject, enclosingOperand))
end
end
end
if umlMessages.size > 0 then
    umlMessages.each{msg|trace_helper.createTLink(sen, msg, message)}
end
end

<table>
<thead>
<tr>
<th>ID</th>
<th>1.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td><strong>Initiation</strong>: generate two sequential messages: actor lifeline (subject) \rightarrow actor boundary class lifeline \rightarrow system lifeline (indirect object). <strong>InternalTransaction</strong>: generate a message from the system lifeline to the indirect object class lifeline. <strong>ResponseToPrimaryActor/ResponseTo-SecondaryActor</strong>: generate two sequential messages: system lifeline (the subject) \rightarrow boundary class lifeline of the actor (the indirect object) \rightarrow the actor lifeline.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule type</th>
<th>Atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformation</td>
<td>operation transform_1_6(globalOperand : uml::InteractionOperand, sen : SimpleSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void</td>
</tr>
<tr>
<td>precondition</td>
<td>is do</td>
</tr>
<tr>
<td>sen != void and (sen.pattern.isInstanceOf(SVIIDO) or sen.pattern.isInstanceOf(SVIODOC)) and sen.transactionType != void</td>
<td></td>
</tr>
<tr>
<td>end is do</td>
<td></td>
</tr>
<tr>
<td>var message : traceability::Message</td>
<td></td>
</tr>
<tr>
<td>message := trace_helper.createMessage(&quot;1_6&quot;, &quot;SVIIDO or SVIODOC-&gt; Message&quot;)</td>
<td></td>
</tr>
<tr>
<td>var umlMessages : Set<a href="">uml::Message</a> init Set<a href="">uml::Message</a>.new</td>
<td></td>
</tr>
<tr>
<td>if sen.transactionType.isInstanceOf(Initiation) then</td>
<td></td>
</tr>
<tr>
<td>var subject : uml::Class</td>
<td></td>
</tr>
<tr>
<td>var object : uml::Class</td>
<td></td>
</tr>
<tr>
<td>var actor : ucm::Actor</td>
<td></td>
</tr>
<tr>
<td>if sen.subject.isInstanceOf(ActorSubj) then</td>
<td></td>
</tr>
<tr>
<td>actor := sen.subject.asType(ActorSubj).actor</td>
<td></td>
</tr>
<tr>
<td>subject := trace_helper.findTargetClass(actor, false)</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>if sen.predicate.objects.select{o</td>
<td>o.isInstanceOf(IndirectObject)}.one.isInstanceOf(IndirectSysObj) then</td>
</tr>
<tr>
<td>object := trace_helper.findTargetClass(uc, void)</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>if subject != void and object != void then</td>
<td></td>
</tr>
<tr>
<td>umlMessages.addAll(generateMsgsActorBoundarySystem(globalOperand, interaction, sen.predicate.description, actor, subject, object, enclosingOperand))</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>if sen.transactionType.isInstanceOf(InternalTransaction) then</td>
<td></td>
</tr>
<tr>
<td>var subject : uml::Class</td>
<td></td>
</tr>
<tr>
<td>var object : uml::Class</td>
<td></td>
</tr>
<tr>
<td>var opr : uml::Operation</td>
<td></td>
</tr>
<tr>
<td>subject := trace_helper.findTargetClass(uc, void)</td>
<td></td>
</tr>
<tr>
<td>if trace_helper.findTargetClass(sen.predicate.objects.one, void) != void then</td>
<td></td>
</tr>
<tr>
<td>object := trace_helper.findTargetClass(sen.predicate.objects. select{o</td>
<td>o.isInstanceOf(IndirectObject)}.one, void)</td>
</tr>
<tr>
<td>else if trace_helper.findTargetClass(sen.predicate.objects.select {o</td>
<td>o.isInstanceOf(IndirectObject)}.one.form.asType(NounPhrase).head, void) != void then</td>
</tr>
<tr>
<td>object := trace_helper.findTargetClass(sen.predicate.objects. select{o</td>
<td>o.isInstanceOf(IndirectObject)}.one.form.asType(NounPhrase).head, void)</td>
</tr>
<tr>
<td>else if sen.predicate.objects.select{o</td>
<td>o.isInstanceOf(IndirectObject)}.one.form.asType(NounPhrase).preHeadString.size&gt;0 then</td>
</tr>
<tr>
<td>if sen.predicate.objects.select{o</td>
<td>o.isInstanceOf(IndirectObject)}.one.form.asType(NounPhrase).preHeadString. select{phs</td>
</tr>
<tr>
<td>object := trace_helper.findTargetClass(sen.predicate.objects.select {o</td>
<td>o.isInstanceOf(IndirectObject)}.one.form. asType(NounPhrase).preHeadString.select(phs</td>
</tr>
<tr>
<td>ID</td>
<td>Description</td>
</tr>
<tr>
<td>----</td>
<td>-------------</td>
</tr>
<tr>
<td>2</td>
<td>Invoke rules 2.1-2.4 to process complex sentence types.</td>
</tr>
<tr>
<td></td>
<td>pre</td>
</tr>
<tr>
<td></td>
<td>cs != <code>void</code></td>
</tr>
</tbody>
</table>
```plaintext
end

var message : traceability::Message
message := trace_helper.createMessage("2", "ComplexSentence -> Message")
if cs.isInstanceOf(ConditionCheckSentence) then
    transform_2_1(globalOperand, cs.asType(ConditionCheckSentence), interaction, uc, enclosingOperand)
end
if cs.isInstanceOf(ConditionalSentence) then
    transform_2_2_2(globalOperand, cs.asType(ConditionalSentence), interaction, uc, enclosingOperand)
end
if cs.isInstanceOf(ParallelSentence) then
    transform_2_2_3(globalOperand, cs.asType(ParallelSentence), interaction, uc, enclosingOperand)
end
if cs.isInstanceOf(IterativeSentence) then
    transform_2_2_4(globalOperand, cs.asType(IterativeSentence), interaction, uc, enclosingOperand)
end

ID 2.1

Description ConditionCheckSentence—If the sentence contains an actor to execute the action, generate two sequential messages: the system lifeline (the subject) Æ the boundary class lifeline of the actor Æ the actor lifeline. If the sentence executes the action by itself, generate a self-message at the system lifeline. Generate a "opt" combined fragment for the alternative flow branching from the sentence and do the following: 1) generate an interaction, 2) generate an interaction use contained in the combined fragment referring to the newly generated interaction, and 3) invoke rules 1-3 to generate messages for the sentences contained in each alternative flow.

Rule type Composite

Transformation operation transform_2_1(globalOperand : uml::InteractionOperand, cs : ConditionCheckSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void
pre precondition is do
    cs != void
end
is do
    var message : traceability::Message
    message := trace_helper.createMessage("2_1", "ConditionCheckSentence -> Messages + CombinedFragments")
    var umlMessages : Set<uml::Message> init Set<uml::Message>.new
    var basicFlow : BasicFlow
    var altFlows : Sequence<AlternativeFlow> init Sequence<AlternativeFlow>.new
    basicFlow := uc.specification.flows.select{f|f.isInstanceOf(BasicFlow)}.one.asType(BasicFlow)
    uc.specification.flows.select{f|f.isInstanceOf(AlternativeFlow)}.each{alt|altFlows.add(alt.asType(AlternativeFlow))}
    var alts : Sequence<AlternativeFlow> init Sequence<AlternativeFlow>.new
    altFlows.each{alt| if alt.bfs.includes(cs) then alts.add(alt) end}
    var subject : uml::Class
    var opr : uml::Operation
    subject := trace_helper.findTargetClass(uc, void)
    if subject != void then
        opr := subject.ownedOperation.select{o| o.name.equals("VALIDATES THAT ").select{o|o.ownedParameter.exists{p| p.name.equals(cs.condition.description.substring(0, cs.condition.description.size))}).one
        if opr == void then
            opr := target_model_helper.generateOperation(subject, "VALIDATES THAT ")
            target_model_helper.generateParameter(opr, cs.condition.description.substring(0, cs.condition.description.size))
        end
        if opr != void then
            if cs.toActor == void then
                end
```
```java
var lifeline : uml::Lifeline
lifeline := interaction.lifeline.select{l|l.represents.type.equals(subject)}.one
if lifeline == void then lifeline := generateLifeline(interaction, subject)
    umlMessages.add(generateMessage(globalOperand, opr.name + "(" + cs.condition.description + ")", interaction, lifeline, void, opr, enclosingOperand))
else
    var lifelineL : uml::Lifeline
    lifelineL := interaction.lifeline.select{l|l.represents.type.equals(subject)}.one
    if lifelineL == void then
        lifelineL := generateLifeline(interaction, subject)
    end
    var lifelineR : uml::Lifeline
    var actor_class : uml::Class
    actor_class := trace_helper.findTargetClass(cs.toActor, false)
    lifelineR := interaction.lifeline.select{l|l.represents.type.equals(actor_class)}.one
    if lifelineR == void then lifelineR := generateActorLifeline(interaction, cs.toActor)
        umlMessages.addAll(generateMsgsSystemBoundaryActor(globalOperand, interaction, opr, cs.toActor, actor_class, subject, enclosingOperand))
    end
end
if alts.size >0 then
    var optCF: uml::CombinedFragment
    init uml::CombinedFragment.new
    var operandELSE : uml::InteractionOperand
    init uml::InteractionOperand.new
    var guardELSE : uml::InteractionConstraint
    init uml::InteractionConstraint.new
    var guardContextELSE : uml::OpaqueExpression
    init uml::OpaqueExpression.new
    optCF.interactionOperator := uml::InteractionOperatorKind.opt
    optCF.operand.add(operandELSE)
    guardContextELSE.body.add("IF NOT(" + cs.condition.description + ")")
    operandELSE.guard := guardELSE
    guardELSE.specification := guardContextELSE
    if enclosingOperand != void then
        enclosingOperand.fragment.add(optCF)
    else if globalOperand != void then
        globalOperand.fragment.add(optCF)
else
    interaction.fragment.add(optCF)
end
var alt_interaction : uml::Interaction
if interaction.container().asType(uml::Collaboration).ownedBehavior.select(b|b.isInstanceOf(Interaction)).select(i|i.name.contains(alts.one.name)).size ==0 then
    var alt_inter : uml::Interaction
    init uml::Interaction.new
    alt_inter.name := alts.one.name
    interaction.container().asType(uml::Collaboration).ownedBehavior.add(alt_inter)
if alts.size >0 then
    if alts.one.steps != void then
        alts.one.steps.each{s|
            if s.isInstanceOf(SimpleSentence) then
                transform_C2_1(void, s.asType(SimpleSentence), alt_inter, uc, void)
            end
        if s.isInstanceOf(ComplexSentence) then
                transform_C2_2(void, s.asType(ComplexSentence), alt_inter, uc, void)
            end
        if s.isInstanceOf(SpecialSentence) then
                transform_C2_3(void, s.asType(SpecialSentence), alt_inter, uc, void)
        end
    }
```
```plaintext
alt_interaction := alt_inter
else
    alt_interaction := interaction.container().asType.uml::Collaboration).
                     ownedBehavior.select{b|b.isInstanceOf(Interaction)}.select
                     {i|i.name.contains(alts.one.name)}.one.asType(Interaction)
end

var interactionUse_ELSE : uml::InteractionUse init uml::InteractionUse.new
interactionUse_ELSE.refersto := alt_interaction
interactionUse_ELSE.name := alt_interaction.name + " : (" + "NOT " +
 cs.condition.description + ")"
operandELSE.fragment.add(interactionUse_ELSE)
alt_interaction.lifeline.each{alt_l|
    var l : uml::Lifeline
    l := interaction.lifeline.select{l|l.name.equals(alt_l.name)}.one
    if l == void then
        if alt_l.represent.type.isInstanceOf(Class) then
            l := generateLifeline(interaction, alt_l.represent.type.asType(Class))
        end
        if alt_l.represent.type.isInstanceOf(uml::Actor) then
            l := copyActorLifeline(interaction, alt_l.represent.type.asType(uml::Actor))
        end
    end
    optCF.covered.add(l) interactionUse_ELSE.covered.add(l)
}

if alt_interaction != void then
    trace_helper.createTLink(cs, alt_interaction, message)
end
if interactionUse_ELSE != void then
    trace_helper.createTLink(cs, interactionUse_ELSE, message)
end
if umlMessages.size >0 then
    umlMessages.each{msg|trace_helper.createTLink(cs, msg, message)}
end
```

<table>
<thead>
<tr>
<th>ID</th>
<th>2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>ConditionalSentence—generate an “alt” combined fragment and invoke rules 1-3 to generate messages for the sentences contained in THEN actions and ELSE actions. Conditions of the sentence (e.g., IF condition) map to the conditions of the “alt” combined fragment. If a partial of the sentence (e.g., ELSEIF-THEN) contained in an alternative flow, then a set of messages is generated for the sentences of the alternative flow and contained in the combined fragment.</td>
</tr>
<tr>
<td>Rule type</td>
<td>Composite</td>
</tr>
<tr>
<td>Transformation</td>
<td>operation transform_2_2_2(globalOperand : uml::InteractionOperand, cs : ConditionalSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void</td>
</tr>
<tr>
<td>pre</td>
<td>precondition is do</td>
</tr>
<tr>
<td></td>
<td>cs != void</td>
</tr>
<tr>
<td>end</td>
<td>is do</td>
</tr>
<tr>
<td></td>
<td>var message : traceability::Message</td>
</tr>
<tr>
<td></td>
<td>message := trace_helper.createMessage(&quot;2_2&quot;, &quot;ConditionalSentence -&gt; alt CombinedFragment&quot;)</td>
</tr>
<tr>
<td>var</td>
<td>basicFlow : BasicFlow</td>
</tr>
<tr>
<td>var</td>
<td>altFlows : Sequence&lt;AlternativeFlow&gt; init Sequence&lt;AlternativeFlow&gt;.new</td>
</tr>
<tr>
<td></td>
<td>basicFlow := uc.specification.flows.select{f</td>
</tr>
<tr>
<td></td>
<td>uc.specification.flows.select{f</td>
</tr>
<tr>
<td>var</td>
<td>alts : Sequence&lt;AlternativeFlow&gt; init Sequence&lt;AlternativeFlow&gt;.new</td>
</tr>
<tr>
<td></td>
<td>altFlows.each{alt</td>
</tr>
<tr>
<td></td>
<td>if alt.bfs.includes(cs) then</td>
</tr>
<tr>
<td></td>
<td>alts.add(alt)</td>
</tr>
</tbody>
</table>
```java
var IFcondition : Sentence
var THENactions : OrderedSet<Sentence> init OrderedSet<Sentence>.new
var ELSEIFcondition : Sentence
var ELSEIFactions : OrderedSet<Sentence> init OrderedSet<Sentence>.new
var ELSEIFactions : OrderedSet<Sentence> init OrderedSet<Sentence>.new
IFcondition := cs.IFcondition
THENactions := cs.THENactions
ELSEIFactions := cs.ELSEIFactions
ELSEIFactions := cs.ELSEIFactions
var altCF: uml::CombinedFragment init uml::CombinedFragment.new
altCF.interactionOperator := uml::InteractionOperatorKind.alt
var operand : uml::InteractionOperand init uml::InteractionOperand.new
if IFcondition != void and THENactions.size >0 then
  var guard : uml::InteractionConstraint init uml::InteractionConstraint.new
  var guardContext : uml::OpaqueExpression init uml::OpaqueExpression.new
  altCF.operand.add(operand)
  guardContext.body.add("IF " + cs.IFcondition.description)
  operand.guard := guard
  guard.specification := guardContext
  THENactions.each(s|
    if s.isInstanceOf(SimpleSentence) then
      transform_1(globalOperand, s.asType(SimpleSentence), interaction, uc, operand)
    end
    if s.isInstanceOf(ComplexSentence) then
      transform_2(globalOperand, s.asType(ComplexSentence), interaction, uc, operand)
    end
    if s.isInstanceOf(SpecialSentence) then
      transform_3(globalOperand, s.asType(SpecialSentence), interaction, uc, operand)
    end
  )
end
if ELSEactions.size >0 then
  var operandELSE : uml::InteractionOperand init uml::InteractionOperand.new
  var guardELSE : uml::InteractionConstraint init uml::InteractionConstraint.new
  var guardContextELSE : uml::OpaqueExpression init uml::OpaqueExpression.new
  var guard : uml::InteractionConstraint init uml::InteractionConstraint.new
  altCF.operand.add(operandELSE)
  guardContextELSE.body.add("ELSE")
  operandELSE.guard := guard
  guard.specification := guardContextELSE
  ELSEactions.each(s|
    if s.isInstanceOf(SimpleSentence) then
      transform_1(globalOperand, s.asType(SimpleSentence), interaction, uc, operandELSE)
    end
    if s.isInstanceOf(ComplexSentence) then
      transform_2(globalOperand, s.asType(ComplexSentence), interaction, uc, operandELSE)
    end
    if s.isInstanceOf(SpecialSentence) then
      transform_3(globalOperand, s.asType(SpecialSentence), interaction, uc, operandELSE)
    end
  )
end
if ELSEIFcondition != void and ELSEIFactions.size >0 then
  var operandELSEIF : uml::InteractionOperand init uml::InteractionOperand.new
  var guardELSEIF : uml::InteractionConstraint init
```
```java
var guardContextELSEIF : uml::OpaqueExpression
        init
        uml::OpaqueExpression
var guard : uml::InteractionConstraint
        init
        uml::InteractionConstraint
        new
altCF.operand.add(operandELSEIF)
guardContextELSEIF.body.add("ELSEIF " + ELSEIFcondition.description)
operandELSEIF.guard := guard
guard.specification := guardContextELSEIF
ELSEIFactions.each{s|
    if s.isInstanceOf(SimpleSentence) then
        transform_1(globalOperand, s.asType(SimpleSentence), interaction, uc, operandELSEIF)
    end
    if s.isInstanceOf(ComplexSentence) then
        transform_2(globalOperand, s.asType(ComplexSentence), interaction, uc, operandELSEIF)
    end
    if s.isInstanceOf(SpecialSentence) then
        transform_3(globalOperand, s.asType(SpecialSentence), interaction, uc, operandELSEIF)
    end
}
}
if alts.size >0 then
    alts.each{alt|
        alt.steps.each{s|
            if s.isInstanceOf(ConditionalSentence) then
                if s.asType(ConditionalSentence).ELSEactions.size>0 then
                    var operandELSE : uml::InteractionOperand
                    init
                    uml::InteractionOperand
var guardELSE : uml::InteractionConstraint
                    init
                    uml::InteractionConstraint
var guardContextELSE : uml::OpaqueExpression
                    init
                    uml::OpaqueExpression
var guard : uml::InteractionConstraint
                    init
                    uml::InteractionConstraint
        altCF.operand.add(operandELSE)
guardContextELSE.body.add("ELSE")
operandELSE.guard := guard
guard.specification := guardContextELSE
s.asType(ConditionalSentence).ELSEactions.each{s|
        if s.isInstanceOf(SimpleSentence) then
            transform_1(globalOperand, s.asType(SimpleSentence), interaction, uc, operandELSE)
        end
        if s.isInstanceOf(ComplexSentence) then
            transform_2(globalOperand, s.asType(ComplexSentence), interaction, uc, operandELSE)
        end
        if s.isInstanceOf(SpecialSentence) then
            transform_3(globalOperand, s.asType(SpecialSentence), interaction, uc, operandELSE)
        end
    }
}
end
if s.asType(ConditionalSentence).ELSEIFcondition != void and
    s.asType(ConditionalSentence).ELSEIFactions.size >0 then
    var operandELSEIF : uml::InteractionOperand
    init
    uml::InteractionOperand
var guardELSEIF : uml::InteractionConstraint
    init
    uml::InteractionConstraint
var guardContextELSEIF : uml::OpaqueExpression
    init
    uml::OpaqueExpression
var guard : uml::InteractionConstraint
    init
    uml::InteractionConstraint
```
altCF.operand.add(operandELSEIF)
guardContextELSEIF.body.add("ELSEIF " +
  s.asType(ConditionalSentence).ELSEIFcondition.description)
operandELSEIF.guard := guard
guard.specification := guardContextELSEIF
s.asType(ConditionalSentence).ELSEIFactions.each{s|
  if s.isInstanceOf(SimpleSentence) then
    transform_1(globalOperand, s.asType(SimpleSentence),
      interaction, uc, operandELSEIF)
  end
  if s.isInstanceOf(ComplexSentence) then
    transform_2(globalOperand, s.asType(ComplexSentence),
      interaction, uc, operandELSEIF)
  end
  if s.isInstanceOf(SpecialSentence) then
    transform_3(globalOperand, s.asType(SpecialSentence),
      interaction, uc, operandELSEIF)
  end
}
}
end
if enclosingOperand != void then
  enclosingOperand.fragment.add(altCF)
else if globalOperand != void then
  globalOperand.fragment.add(altCF)
else
  interaction.fragment.add(altCF)
end
if altCF != void then trace_helper.createTLink(cs, altCF, message) end
end

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Rule type</th>
<th>Transformation</th>
</tr>
</thead>
</table>
| 2.3   | ParallelSentence—generate a “par” combined fragment and invoke rules 1-3 to generate messages for the sentences connected by the keyword MEANWHILE. | Composite   | operation transform_2_3(globalOperand : uml::InteractionOperand, cs : ParallelSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void  
|       |                                                                             |             | pre precondition is do                                |
|       |                                                                             |             | cs != void                                           |
|       |                                                                             |             | is do                                               |
|       |                                                                             |             | var message : traceability::Message                 |
|       |                                                                             |             | message := trace_helper.createMessage("2_3", "ParallelSentence -> par CombinedFragment") |
|       |                                                                             |             | var par : uml::CombinedFragment init uml::CombinedFragment.new |
|       |                                                                             |             | par.interactionOperator := uml::InteractionOperatorKind.par |
|       |                                                                             |             | var sens : Sequence<Sentence> init Sequence<Sentence>.new |
|       |                                                                             |             | sens.addAll(cs.parallelActions)                      |
|       |                                                                             |             | sens.each{s | |
|       |                                                                             |             | var operand : uml::InteractionOperand init uml::InteractionOperand.new |
|       |                                                                             |             | par.operand.add(operand)                             |
|       |                                                                             |             | if s.isInstanceOf(SimpleSentence) then               |
|       |                                                                             |             | transform_C2_1(globalOperand, s.asType(SimpleSentence), interaction, uc, operand) |
|       |                                                                             |             | end                                                  |
|       |                                                                             |             | if s.isInstanceOf(ComplexSentence) then               |
|       |                                                                             |             | transform_C2_2(globalOperand, s.asType(ComplexSentence), interaction, uc, operand) |
|       |                                                                             |             | end                                                  |
|       |                                                                             |             | if s.isInstanceOf(SpecialSentence) then               |

transform_C2_3(globalOperand, s.asType(SpecialSentence), interaction, uc, operand)
end
}
if enclosingOperand != void then
  enclosingOperand.fragment.add(par)
else if globalOperand != void then
  globalOperand.fragment.add(par)
else
  interaction.fragment.add(par)
end
if par != void then
  trace_helper.createTLink(cs, par, message)
end
end

ID | Description
---|---
2.4 | IterativeSentence—generate a “loop” combined fragment and invoke rules 1-3 to generate messages for the sentences in the loop.

Rule type: Composite

**Transformation**

<table>
<thead>
<tr>
<th>operation</th>
<th>transform_2_4(globalOperand : uml::InteractionOperand, cs : IterativeSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pre</strong></td>
<td>precondition is do</td>
</tr>
<tr>
<td></td>
<td>cs != void</td>
</tr>
<tr>
<td></td>
<td>is do</td>
</tr>
<tr>
<td></td>
<td>var message : traceability::Message</td>
</tr>
<tr>
<td></td>
<td>message := trace_helper.createMessage(&quot;2_4&quot;, &quot;IteractiveSentence -&gt; loop CombinedFragment&quot;)</td>
</tr>
<tr>
<td></td>
<td>var loopOpr: uml::CombinedFragment init uml::CombinedFragment.new</td>
</tr>
<tr>
<td></td>
<td>loopOpr.interactionOperator := uml::InteractionOperatorKind.loop</td>
</tr>
<tr>
<td></td>
<td>var actions : Sequence&lt;Sentence&gt; init Sequence&lt;Sentence&gt;.new</td>
</tr>
<tr>
<td></td>
<td>actions.addAll(cs.DOactions)</td>
</tr>
<tr>
<td></td>
<td>var condition : Sentence</td>
</tr>
<tr>
<td></td>
<td>condition := cs.WHILEcondition</td>
</tr>
<tr>
<td></td>
<td>var operand : uml::InteractionOperand init uml::InteractionOperand.new</td>
</tr>
<tr>
<td></td>
<td>var guard : uml::InteractionConstraint init uml::InteractionConstraint.new</td>
</tr>
<tr>
<td></td>
<td>var guardContext : uml::OpaqueExpression init uml::OpaqueExpression.new</td>
</tr>
<tr>
<td></td>
<td>loopOpr.operand.add(operand)</td>
</tr>
<tr>
<td></td>
<td>guardContext.body.add(&quot;UNTIL &quot; + condition.description)</td>
</tr>
<tr>
<td></td>
<td>operand.guard := guard</td>
</tr>
<tr>
<td></td>
<td>guard.specification := guardContext</td>
</tr>
<tr>
<td></td>
<td>actions.each{s</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| | }
| | if enclosingOperand != void then enclosingOperand.fragment.add(loopOpr) |
| | else if globalOperand != void then |
| | | globalOperand.fragment.add(loopOpr) |
| | else |
| | | interaction.fragment.add(loopOpr) |
| | else |
| | | if loopOpr != void then |
| | | trace_helper.createTLink(cs, loopOpr, message) |
| | end |
| | end |

ID | Description
---|---
3 | Invoke rules 3.1-3.4 to process each special sentence.
Rule type: Composite

**Operation**: transform_3(globalOperand : uml::InteractionOperand, ss : SpecialSentence, interaction : uml::Interaction, uc : ucm::UseCase, enclosingOperand : InteractionOperand) : Void

**Precondition**: is do

- ss != void

**Is do**

- var message : traceability::Message
  
  message := trace_helper.createMessage("3", "SpecialSentence")

- if ss.isInstanceOf(IncludeSentence) then
  
  transform_3_1(globalOperand, ss.asType(IncludeSentence), interaction, enclosingOperand)

- if ss.isInstanceOf(ExtendSentence) then
  
  transform_3_2(globalOperand, ss.asType(ExtendSentence), interaction, enclosingOperand)

- if ss.isInstanceOf(AbortSentence) then
  
  transform_3_3(globalOperand, ss.asType(AbortSentence), interaction, enclosingOperand, uc)

- if ss.isInstanceOf(ResumeStepSentence) then
  
  transform_3_4(globalOperand, ss.asType(ResumeStepSentence), interaction, enclosingOperand, uc)

---

**ID**: 3.1

**Description**: IncludeSentence – generate an interaction use referring to the interaction corresponding to the included use case and place the interaction use where the special sentence is invoked.

**Rule type**: Atomic

**Operation**: transform_3_1(globalOperand : uml::InteractionOperand, sen : IncludeSentence, interaction : uml::Interaction, enclosingOperand : InteractionOperand) : Void

**Precondition**: is do

- sen != void

**Is do**

- var message : traceability::Message
  
  message := trace_helper.createMessage("3_1", "IncludeSentence -> InteractionUse")

- var includedUseCase : ucm::UseCase
  
  includedUseCase := sen.includedUseCase

- var interactionUse : uml::InteractionUse
  
  init uml::InteractionUse.new

  interactionUse.refersTo := trace_helper.findTargetInteraction(includedUseCase)

  interactionUse.name := includedUseCase.name

- var includedInteraction : uml::Interaction
  
  includedInteraction := umlModel.packagedElement.select{p|p.isInstanceOf(Collaboration)}.select{c|c.name.equals(interactionUse.name)}.

  one.asType(uml::Collaboration).ownedBehavior.

  select{b|b.isInstanceOf(uml::Interaction)}.select{i|i.name.

  equals(interactionUse.name)}.one.asType(uml::Interaction)

  interactionUse.refersTo := includedInteraction

- if enclosingOperand != void then
  
  enclosingOperand.fragment.add(interactionUse)

- else if globalOperand != void then
  
  globalOperand.fragment.add(interactionUse)

- else
  
  interaction.fragment.add(interactionUse)

**End**

- includedInteraction.lifeline.each{included_l|
  
  var l : uml::Lifeline
  
  l := interaction.lifeline.select{l|l.name.equals(included_l.name)}.one
### ID 3.2

**Description**
ExtendSentence—generate an interaction use referring to the interaction corresponding to the extending use case and place the interaction use where the special sentence is invoked.

**Rule type** Atomic

**Transformation**

```plaintext
operation transform_C2_3_2(globalOperand : uml::InteractionOperand, sen : ExtendSentence, interaction : uml::Interaction, enclosingOperand : InteractionOperand) : Void
pre

precondition

is do

sen != void

end

is do

var message : traceability::Message
message := trace_helper.createMessage("3_2", "ExtendSentence -> InteractionUse")
var extendedByUseCase : ucm::UseCase
extendedByUseCase := sen.extendingUseCase
var interactionUse : uml::InteractionUse init uml::InteractionUse.new
interactionUse.refersTo := trace_helper.findTargetInteraction((extendedByUseCase)
interactionUse.name := extendedByUseCase.name
var includedInteraction : uml::Interaction
includedInteraction := umlModel.packagedElement.select{p|p.isInstanceOf(Collaboration)}. select{c|c.name.equals(interactionUse.name)}.one.asType(uml::Collaboration).ownedBehavior. select{b|b.isInstanceOf(uml::Interaction)}.select{i|i.name.equals(interactionUse.name)}.one.asType(uml::Interaction)
interactionUse.refersTo := includedInteraction
if enclosingOperand != void then
    enclosingOperand.fragment.add(interactionUse)
else if globalOperand != void then
    globalOperand.fragment.add(interactionUse)
else
    interaction.fragment.add(interactionUse)
end

end

end

includedInteraction.lifeline.each{included_l|
var l : uml::Lifeline
l := interaction.lifeline.select{l|l.name.equals(included_l.name)}.one
if l == void then
    if included_l.represents.type.isInstanceOf(Class) then
        l := generateLifeline(interaction,
            included_l.represents.type.asType(Class))
    end
    if included_l.represents.type.isInstanceOf.uml::Actor) then
        l := copyActorLifeline(interaction,
            included_l.represents.type.asType(uml::Actor))
    end
end
```
interactionUse.covered.add(l)
}
if interactionUse != void then trace_helper.createTLink(sen, interactionUse, message) end

ID 3.3
Description AbortSentence—generate a destroy message pointing to the system lifeline.
Rule type Atomic
Transformation
operation transform_C2_3_3(globalOperand : uml::InteractionOperand, sen : AbortSentence, interaction : uml::Interaction, enclosingOperand : InteractionOperand, uc : ucm::UseCase) : Void
pre precondition is do
sen != void
end
is do
var message : traceability::Message
message := trace_helper.createMessage("3_3", "AbortSentence -> DestroyMessage")
var umlMessage : uml::Message
var subject : uml::Class
var lifeline : uml::Lifeline
subject := trace_helper.findTargetClass(uc, void)
if subject != void then
lifeline := interaction.lifeline.select{l | l.represents.type.equals(subject)}.one
if lifeline != void then
umlMessage := generateDestroyMessage(globalOperand, interaction, lifeline, enclosingOperand)
end
if umlMessage != void then trace_helper.createTLink(sen, umlMessage, message)
end
end

ID 3.4
Description ResumeBackSentence—generate a gate message referring to resume-back-step (message).
Rule type Atomic
Transformation
operation transform_C2_3_4(globalOperand : uml::InteractionOperand, sen : ResumeStepSentence, interaction : uml::Interaction, enclosingOperand : InteractionOperand, uc : ucm::UseCase) : Void
pre precondition is do
sen != void
end
is do
var tmessage : traceability::Message
tmessage := trace_helper.createMessage("C2_3_2", "ResumeStepSentence")
var subject : uml::Class
var lifeline : uml::Lifeline
var message : uml::Message init uml::Message.new
var opr : uml::Operation
var name : String
if sen.description != void then
name := sen.description
else if sen.content != void then
name := sen.content
end
end
subject := trace_helper.findTargetClass(uc, void)
if subject != void then
opr := subject.ownedOperation.select{o | o.name.toLowerCase.equals(name.toLowerCase)}.one
if opr != void then opr := target_model_helper.generateOperation(subject, name)
end
lifeline := Interaction.lifeline.select{l|l.represents.type.equals(subject)}.one

if lifeline != void and opr != void then
    var sendOccrSpe : uml::MessageOccurrenceSpecification init
        uml::MessageOccurrenceSpecification.new
    var sendOperationEvent : uml::SendOperationEvent init
        uml::SendOperationEvent.new
    var gate : uml::Gate init uml::Gate.new
        sendOperationEvent.operation := opr
        sendOccrSpe.message := message
        sendOccrSpe.event := sendOperationEvent
        sendOccrSpe.covered.add(lifeline)
        gate.message := message
        gate.name := name
        message.name := name
        message.sendEvent := sendOccrSpe
        message.receiveEvent := gate
        message.messageSort := uml::MessageSort.synchronisation
        if enclosingOperand == void then
            if globalOperand != void then
                sendOccrSpe.enclosingOperand := globalOperand
                globalOperand.container().asType(CombinedFragment).covered.add(lifeline)
                globalOperand.fragment.add(sendOccrSpe)
            else
                interaction.fragment.add(sendOccrSpe)
            end
        else
            sendOccrSpe.enclosingOperand := enclosingOperand
            enclosingOperand.container().asType(CombinedFragment).covered.add(lifeline)
            enclosingOperand.fragment.add(sendOccrSpe)
            if globalOperand != void then
                globalOperand.container().asType(CombinedFragment).covered.add(lifeline)
            end
        end
    umlModel.packagedElement.add(sendOperationEvent)
    interaction.message.add(message)
    interaction.formalGate.add(gate)
    if message != void then
        trace_helper.createTLink(sen, message, tmessage)
    end
    if gate != void then
        trace_helper.createTLink(sen, gate, tmessage)
    end
end

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Process global alternative flows. The rule also invokes rules 1-3 to process the steps contained in the global alternative flows.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule type</th>
<th>Atomic</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Transformation</th>
<th>operation  transform_4(globalOperand : uml::Interaction, alt : GlobalAlternativeFlow, interaction : uml::Interaction, uc : ucm::UseCase) : Void</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre condition</td>
<td>is do</td>
</tr>
<tr>
<td></td>
<td>alt != void</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>is do</td>
</tr>
<tr>
<td></td>
<td>var alt_interaction : uml::Interaction init uml::Interaction.new</td>
</tr>
<tr>
<td></td>
<td>alt_interaction.name := alt.name</td>
</tr>
<tr>
<td></td>
<td>interaction.container().asType(uml::Collaboration).ownedBehavior.add(alteration)</td>
</tr>
<tr>
<td></td>
<td>var altCF : uml::CombinedFragment init uml::CombinedFragment.new</td>
</tr>
<tr>
<td></td>
<td>var operand : uml::InteractionOperand init uml::InteractionOperand.new</td>
</tr>
<tr>
<td></td>
<td>var guard : uml::InteractionConstraint init uml::InteractionConstraint.new</td>
</tr>
<tr>
<td></td>
<td>var guardContext : uml::OpaqueExpression init uml::OpaqueExpression.new</td>
</tr>
<tr>
<td></td>
<td>var operandELSE : uml::InteractionOperand init uml::InteractionOperand.new</td>
</tr>
<tr>
<td></td>
<td>var globalOperand := operand</td>
</tr>
<tr>
<td></td>
<td>var guardELSE : uml::InteractionConstraint init uml::InteractionConstraint.new</td>
</tr>
</tbody>
</table>
```java
var guardContextELSE : uml::OpaqueExpression init uml::OpaqueExpression.new
altCF.interactionOperator := uml::InteractionOperatorKind.alt
altCF.operand.add(globalOperand)
guardContext.body.add("IF NOT: " + alt.condition.description)
globalOperand.guard := guard
guard.specification := guardContext
altCF.operand.add(operandELSE)
guardContextELSE.body.add("ELSE")
operandELSE.guard := guardELSE
guardELSE.specification := guardContextELSE
interaction.fragment.add(altCF)
var interactionUse : uml::InteractionUse init uml::InteractionUse.new
interactionUse.refersTo := alt_interaction
interactionUse.name := alt.name
operandELSE.fragment.add(interactionUse)
alt.steps.each{s|
    if s.isInstanceOf(SimpleSentence) then
        transform_1(void, s.asType(SimpleSentence), alt_interaction, uc, void)
    end
    if s.isInstanceOf(ComplexSentence) then
        transform_2(void, s.asType(ComplexSentence), alt_interaction, uc, void)
    end
    if s.isInstanceOf(SpecialSentence) then
        transform_3(void, s.asType(SpecialSentence), alt_interaction, uc, void)
    end
}]
alt_interaction.lifeline.each{alt_l|
    var l : uml::Lifeline
    l := interaction.lifeline.select{l|l.name.equals(alt_l.name)}.one
    if l == void then
        if alt_l.represents.type.isInstanceOf(uml::Actor) then
            l := copyActorLifeline(interaction, alt_l.represents.type.asType(uml::Actor))
        else if alt_l.represents.type.isInstanceOf(uml::Class) then
            l := generateLifeline(interaction, alt_l.represents.type.asType(Class))
        end
    end
    altCF.covered.add(l)
    interactionUse.covered.add(l)
}]
if alt_interaction != void then trace_helper.createTLink
    (alt, alt_interaction, message) end
if altCF != void then trace_helper.createTLink(alt, altCF, message) end
if interactionUse != void then trace_helper.createTLink
    (alt, interactionUse, message) end
end
```
11 Appendix B

11.1 ATM

11.1.1 Validate PIN
Validate PIN (from textbook)
11.1.2 Withdraw Funds
Withdraw Funds (textbook)
Simula Research Laboratory, Technical Report 2010-04
Carleton University, Technical Report SCE-10-03

Alt 1

1. create
2. displays an apology message

1. create
2. ejects the ATM card

1. destroy

Alt 2

1. create
2. displays an apology message

1. create
2. ejects the ATM card

1. shuts down
2. destroy
11.1.3 Elevator
Select Destination (text book)
Alt 1

1. Ask the elevator to stay at the current floor.

2. Keep the elevator door open.

3. Destroy
11.1.4 ARENA
Announce Tournament (Textbook)
11.1.5 FRIEND
Report Emergency (textbook)
5. Communicates with the system to fill the form by selecting the emergency level, type, location, brief description of the situation, and possible responses to the emergency situation.

6. Communicates with the system to fill the form by selecting the emergency level, type, location, brief description of the situation, and possible responses to the emergency situation.

7. Creates a new incident in the database.

8. Notifies the dispatcher.

9. Communicates with the system to create an incident in the database.

10. Communicates with the system to create an incident in the database.

11. Acknowledges the system of the emergency report.

12. Acknowledges the system of the emergency report.

13. Communicates with the system to select a response.

14. Communicates with the system to select a response.

15. Communicates with the system to select a response.
11.1.6 CPD
Create Customer Order (aToucan)
Alt 1

1. informs Sales: part number unknown

2. provides an alternative part number to the system

3. provides an alternative part number to the system

4. RESUME STEP 1

Alt 2
Insufficient Stock (textbook)

```
if (isAccepted) {
    create(requiredQty)
    savePartOrder(po)
    poPartOrder:
    destroy
}
if (!isAccepted) {
    create
    savePendingOrder(pdo)
    pdoPendingOrder:
    destroy
}
CreateVendorOrder
```

Insufficient Stock (aToucan)
11.1.7 VS
Rent Video (aToucan)

Alt 1

1. create
   - <<Control>> Rent video Control
2. returns a failure message ()
3. destroy