On System Sequence Descriptions

Bert de Brock
University of Groningen, Faculty of Economics and Business
PO Box 800, 9700 AV Groningen, The Netherlands
E.O.de.Brock@rug.nl

Abstract

Context: Use cases (UCs) are widely used to specify the functionality of a SW system. A UC is usually worked out in a Main Success Scenario and several other (Alternative) Scenarios. To make the overall structure of the UC clear and to prepare for the software implementing the UC, these scenarios must be integrated into one structure per UC.
Question/problem: How to integrate the different scenarios of a UC into one structure?
Our solution: We propose so-called (textual) *System Sequence Descriptions* (SSDs). We introduce a suitable (context-free) grammar for our SSDs. We can express all usual constructs with our SSDs, such as primary and secondary actors, basic steps, the internal responsibilities of the system, sequential composition, arbitrary order, loops/repetition, conditionals, alternatives, options, choices, definitions, and calls/'Includes'.

To support validation of the resulting SSDs (with integrated scenarios) and check it with the users (Requirements quality assessment), we give (inductive) translation rules to translate the SSDs to natural language (Natural Language *Generation* for RE).

Additionally, we give (inductive) rules to generate *graphical* SSDs (like the more familiar UML-diagrams) from our *textual* SSDs. This might support validation too.
Results: With this new artefact design we can easily integrate the different scenarios of a UC into one clear SSD and also check the result with the users. That integrated SSD clarifies the overall structure of the UC and forms a suitable basis for implementation. With a nontrivial example we illustrate that the proposed grammar is very practical and that the approach scales up easily.
Main contribution: The paper describes novel technical solutions for the application of NL-technologies to RE-relevant artefacts. The paper includes additional solutions to the RE-problem of validation.

Keywords: System Sequence Description, Use Case, User Story, Linguistic Structure, Grammar, Natural Language Generation, Diagram

1 Introduction

An NL-based development path for functional requirements (via 'stepwise clarification') was sketched in [Bro19], starting with an initial user wish (UW) and then going from a user story (US) via a use case (UC) and its system sequence description (SSD) to an information machine (IM) and finally to a realization in an information system (IS), e.g., by a *method* in an OO-system or a (stored) *procedure* in a relational system. In a simple picture (where 'A => B' means 'use A for producing B'):

UW => US => UC => SSD => IM => IS

A UC is typically worked out into a Main Success Scenario (MSS) and zero or more Alternative Scenarios (AS) [Coc01, Lar05]. To integrate the different scenarios of a UC into one structure, we introduce and use textual *System Sequence Descriptions* (SSDs). This integration is illustrated in Example 1. Where MSSs and ASs might typically be written by people from the user organization, SSDs might typically be written by a business/ requirements analyst. SSDs can form the bridge between the users' world and the developers' world.

To support validation of the resulting SSDs and check the result with the users (Requirements quality assessment), we also give (inductive) translation rules to translate a textual SSD (tSSD) to natural language (Natural Language *Generation* for RE, NLG4RE). Additionally, we give (inductive) rules to generate a *graphical* SSD (gSSD, like the more familiar *UML-diagram*) from a *textual* SSD. In a simple, additional picture:

$$\begin{tikzcd} \text{UC} = \text{MSS} + \text{AS}^* \arrow[rr, Rightarrow] & & \text{tSSD} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\ & \text{NL-text} & & \text{gSSD} \end{tikzcd}$$

No such approach is mentioned in the recent systematic literature review of use case specifications research [Tiw15], where 119 papers in this area were thoroughly examined.

The paper is organized as follows: Section 2 explains some basic concepts and Section 3 recalls the general linguistic structures for development paths from [Bro19]. In Section 4 we introduce a grammar for *textual* system sequence descriptions (SSDs). Section 5 looks at the nature of the atomic instructions in SSDs and their relation with the system under development (which is considered as an information machine). To support validation of SSDs and check it with the users, Section 6 contains rules to translate the SSDs to natural language. Section 7 contains (inductive) rules to generate *graphical* SSDs (UML-like sequence diagrams) from *textual* SSDs. Sections 4, 5, 6, and 7 clearly extend [Bro19].

2 Basic concepts

Informally, a **user wish** (UW) is a 'wish', expressed in NL, of a (future) user which the system should be able to fulfil, e.g., '*Register a student*' or '*Process a sale*'.

For us, a parameterized **user story** (US) is a user wish extended with the role of the intended user and the relevant parameters, e.g., the wish of an administrator to '*Register a student with a given name, address, and phone number*'. A user story can have an optional *benefit*-part. According to [Luc16] user stories are popular as a method for representing requirements, especially in agile development environments. We could write:

US = UW + user role + relevant parameters [+ benefit]

A **use case** (UC) is a text in NL (natural language) that describes the sequence of steps in a typical usage of the system [Jac11, UC20]. A UC corresponds roughly to an *elementary business process* [Lar05].

UWs, USs, and UCs are all expressed in the natural language of the user (say English or Dutch). On the other hand, a **system sequence description** (SSD) of a use case schematically depicts the interactions between the primary actor (user), the system (as a black box), and other actors (if any), including the messages between them [Lar05]: An SSD is a kind of stylised UC and makes the prospective inputs, state changes, and outputs of the system more explicit. SSDs are usually drawn as UML-diagrams, see [Lar05, SSD20], but we use *textual* SSDs.

We can formalise the informal requirements using the formal notion of an information machine (IM):

An **information machine** is a 5-tuple (I, O, S, G, T) consisting of

- a set I (of *inputs*), a set O (of *outputs*), and a set S (of *states*)
- a function G: S x I → O, mapping pairs of a state and an input to the corresponding output
- a function T: S x I → S, mapping pairs of a state and an input to the corresponding next state

An IM is equivalent to the notion of *data machine* in [Pie89] and to a (not necessarily finite) *Mealy machine* without a special start state [Mea55]. The working of an IM can be illustrated in a picture (with $i \in I$ and $s \in S$):

$$i \rightarrow \boxed{s \longmapsto T(i,s)} \rightarrow G(i,s)$$

In words: Upon an input i, an IM in state s produces output G(i,s) and changes its internal state from s to T(i,s).

3 Some relevant linguistic structures

[Bro19] presents some general linguistic structures for the development path of a functional requirement (FR), which often starts from a user wish or user story [Coh04]. A general linguistic structure for a user wish (UW) is

UW: <(action) verb> **a** <noun (phrase)>

Table 1 summarizes the related subsequent grammatical forms for the US, 1st step of the UC, 1st step of the SSD, input for the IM, and IS-method (where α denotes the action verb and β the noun phrase in the original user wish). Each pattern clearly helps in producing the next one:

Table 1: Summary of the relationship between the subsequent grammatical forms

UW	α **a** β
US	**As a** <role>**, I want to** α **a** β **with a given** <parameter list>
UC	First step: **The** <role> (**user**) **asks the system to** α **a** β **with a given** <par. list>
SSD	First step: **User → System:** $\alpha\beta$**(**<parameter list>**)** where User is a <role>
IM	Inputs: $\alpha\beta$**(**<parameter list>**)** for all possible value combinations of <par. list>
IS	Method/procedure $\alpha\beta$ with <parameter list> (plus maybe an output parameter), e.g., a *method* in an OO-system or a (stored) *procedure* in a relational system

We note that the suggested naming policy provides *bi-directional traceability* [GF94, Cle12]: from the original user wish to the final software code and back. It also enhances *transparency* during the development of an FR.

Note that the first step in the UC/SSD gives a clue for the *heading* of the IS-method/procedure. The next steps in the SSD give a clue for the *body* of that method/procedure. E.g., if the SSD contains a lot of calls to (or *Includes* of) other SSDs (see Section 4) then that method/procedure might use a lot of calls to sub-methods/sub-programs.

However, [Bro19] did not discuss how the steps within an SSD could look like. That is what we will do now.

4 A grammar for textual SSDs

We propose a grammar for textual SSDs. The terminals are written in bold. The nonterminal A stands for 'atomic instruction' (step), P for 'actor' (or 'participant'), M for 'message', S for 'instruction' (or SSD), C for 'condition', B for 'basic condition', N for 'instruction name', and D for 'definition':

A ::= P ➡ P**:** M   */* where 'X ➡ Y: M' means: 'X sends M to Y'; see Section 5*
P ::= **System** | …
S ::= A | S **;** S | **begin** S **end** | **if** C **then** S [**else** S] **end** | **while** C **do** S **end** | **repeat** S **until** C
  | S **,** S | **maybe** S **end** | **either** S **or** S **end** | **do** N   */* the first 3 introduce non-determinism*
C ::= B | **true** | **false** | **not** C | (C **and** C) | (C **or** C)
D ::= **define** N **as** S **end**

Informally, the construct 's1**,** s2' indicates that the order is irrelevant ('do s1 and s2 in any order'), '**maybe** s **end**' means 'do s or do nothing', and '**either** s1 **or** s2 **end**' means 'choose between doing s1 and doing s2'.

The construct '**do** N' is known as an *Include* or a *Call*. [Bro20] gives a formal semantics for all the constructs mentioned here.

In order to avoid ambiguity, we use the binding rule that '**,**' binds stronger than '**;**'. So we must read 'A**,** B**;** C' as '(A**,** B)**;** C', which means: do B and do A in arbitrary order, and then do C. To get the reading 'A**,** (B**;** C)' we can write 'A**, begin** B**;** C **end**'.

We note that the values (terminals) for the nonterminals B, P, M, and N are application dependent ('domain specific'), apart from **System** for P. They will appear during the development of the specific application.

The terminal **System** represents the system under consideration.

5 The nature of atomic instructions

For atomic instructions where at least one actor/participant is **System**, we can distinguish the following situations:

(1)	Actor ➡ **System:** i	Elucidates the input messages the system can expect	*/* input step*
(2)	**System** ➡ **System:** y	Elucidates the transitions (and/or checks) the system should make	*/* internal step*
(3)	**System** ➡ Actor**:** o	Elucidates the output messages the system should produce	*/* output step*

where Actor ≠ **System**. An atomic instruction in which **System** is not involved, is called an *external step*.
The pattern *input step* followed by an *internal step* followed by an *output step* is quite common.

The relation between the first three types of atomic instructions and the system, considered as an information machine and shown as a ‘black box’, can be depicted as follows:

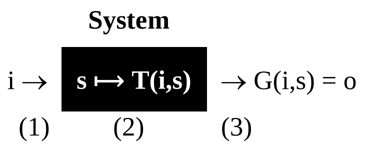


Figure 1: The relation between atomic instructions and the system

Further explanation of Figure 1:

(1) i is an input message for the information machine
(2) y describes a state change (from state s to state T(i,s))
(3) o is an output message of the information machine (which depends on the input message i and the state s).

Example 1: A textual SSD

As an illustration we made an excerpt from Larman’s well-known use case *Process Sale* [Lar05] such that almost all our constructs appear. The excerpt integrates the Main Success Scenario and several Alternative Scenarios. The underlined names below constitute clickable links to their definition.

```
define processSale as
  Cashier ➡ System: makeNewSale;                          /* input step
  System ➡ System: create Sale;                           /* internal step
  while customer has more items do
    do handleItem;
    System ➡ Cashier: description and running total       /* output step
  end;
  Cashier ➡ System: endSale;                              /* input step
  System ➡ System: compute and register total with taxes; /* internal step
  System ➡ Cashier: total with taxes;                     /* output step
  maybe do handleDiscount end;
  do handlePayment;
  System ➡ System: log completed sale,                    /* internal step
  System ➡ InvSys: sale and payment info;                 /* output to Inventory system
  if printer is out of paper then do handlePaperShortage end;
  System ➡ Cashier: receipt;                              /* output step
  maybe do handleGiftReceipt end
end

define handleItem as
  if item is normal
    then Cashier ➡ System: enterItem(itemID, quantity)    /* input step
    else  Cashier ➡ System: enterPricedItem(category, price) /* input step
  end;
  System ➡ System: log item sale                          /* internal step
end

define handlePayment as
  either Cashier ➡ System: makeCashPayment(amount)        /* input step
    or Customer ➡ System: makeCreditPayment(credit card, pin code) /* input step
  end;
  System ➡ System: register payment                       /* internal step
end

define handlePaperShortage as
  System ➡ Cashier: “Out of paper”;                       /* output step
  Cashier ➡ Cashier: replace paper;                       /* external step
  Cashier ➡ System: printReceipt                          /* input step
end
```

define handleDiscount **as** … **end**
define handleGiftReceipt **as** … **end**

For reasons of space, we did not work out the definitions of *handleDiscount* and *handleGiftReceipt*.

6 Generating natural language texts

A use case is described via an MSS plus several separate ASs. In the SSD these separate parts are integrated. To support validation of the resulting SSD and check it with the users, we translate the SSD to natural language (NL). The main purpose of the translation of the SSD back to NL is to validate the integration with the user organization.

Function F below inductively translates SSDs to English, by assigning to each SSD an expression in English, in terms of the direct constituents of that SSD (*compositionality principle* [SEP17]):

1.	F(Actor ➡ **System:** γ)	≝	**the** F(actor) **asks the System to** F(γ)	/* For Actor ≠ **System**
2.	F(**System** ➡ Actor**:** γ)	≝	**the System sends** F(γ) **to** F(actor)	/* For Actor ≠ **System**
3.	F(Actor ➡ Actor**:** γ)	≝	**the** F(actor) **does** F(γ)	/* If the same actor is mentioned twice
4.	F(e1**;** e2)	≝	F(e1)**.** <newline> F(e2)	/* Sequential order is indicated by a dot
5.	F(e1**,** e2)	≝	F(e1) **and** <newline> F(e2)	/* Arbitrary order is indicated by **and**
6.	F(**begin** e **end**)	≝	**begin** F(e) **end**	
7.	F(**if** c1 **then** e1 [**else** e2] **end**)	≝	**if** F(c1) **then** F(e1) [**else** F(e2)] **end**	
8.	F(**while** c **do** e **end**)	≝	**while** F(c) **do** F(e) **end**	
9.	F(**repeat** e **until** c)	≝	**repeat** F(e) **until** F(c)	
10.	F(**maybe** e **end**)	≝	**maybe** F(e) **end**	
11.	F(**either** e1 **or** e2 **end**)	≝	**either** F(e1) **or** F(e2) **end**	
12.	F(**do** n)	≝	**do** F(n)*****	/* The star indicates a follow-up:
13.	*** where** F(n) **means:** F(e) **end**			/* if n was introduced by '**define** n **as** e **end**'

Ad 1-2:	F(actor) often is the user but it could be an external system as well
Ad 3:	If the same actor is mentioned twice, the step indicates what that actor has to do. Often Actor = **System**
Ad 6-11:	These translations are straightforward
Ad 8:	Instead of **while** we could use, e.g., **as long as**
Ad 12:	The star is part of the translation
Ad 13:	The follow-up can be put after the complete translation of the main text

Section 5 explains the nature of the message γ in the basic steps. For an actor, message or instruction name x, when it was well-chosen, F(x) could simply be x itself (as in Example 2).

As a finishing touch, some translations should start with a capital letter: The occurrence of F(e2) after the dot in Rule 4, the occurrence of F(e) after '**means:**' in Rule 13, and the first sentence of the complete translation.

Essentially, it all boils down to replacing the basic steps by the sentences indicated, replacing '**;**' by '**.**', replacing '**,**' by '**and**', add a star after a Call (a.k.a. Include), and replacing '**define** n **as**' by '*** where** F(n) **means:**'.

The end result of translating the SSD back to natural language might be more structured and maybe even be clearer than the original description.

Example 2: The textual SSD for *Process Sale* translated to English

We apply our translation rules to the textual SSD from Example 1 (keeping the originally bold texts in bold). The underlined names below constitute clickable links to their description.

```
* where processSale means:
  The Cashier asks the System to makeNewSale.
  The System does create Sale.
  While customer has more items do
    do handleItem*.
    The System sends description and running total to Cashier
  end.
  The Cashier asks the System to endSale.
  The System does compute and register total with taxes.
  The System sends total with taxes to Cashier.
  Maybe do handleDiscount* end.
  Do handlePayment*.
  The System does log completed sale and
  the System sends sale and payment info to InvSys.
  If printer is out of paper then do handlePaperShortage* end.
  The System sends receipt to Cashier.
  Maybe do handleGiftReceipt* end
end

* where handleItem means:
    If item is normal
      then the Cashier asks the System to enterItem(itemID, quantity)
      else  the Cashier asks the System to enterPricedItem(category, price)
    end.
    The System does log item sale
end

* where handlePayment means:
    Either the Cashier asks the System to makeCashPayment(amount)
      or the Customer asks the System to makeCreditPayment(credit card, pin code)
    end.
    The System does register payment
end

* where handlePaperShortage means:
    The System sends "Out of paper" to Cashier.
    The Cashier does replace paper.
    The Cashier asks the System to printReceipt
end

* where handleDiscount     means: … end
* where handleGiftReceipt  means: … end
```

For reasons of space, we did not work out the descriptions of *handleDiscount* and *handleGiftReceipt*.

7 Generating graphical SSDs

In this section we give (inductive) translation rules from textual SSDs to graphical SSDs (sequence *diagrams*). For each textual SSD X we define its diagram *D*(X) inductively in terms of the diagrams of its direct constituents, indicated by yellow rectangles. (We note that inductive rules to generate graphical representations are rare.)

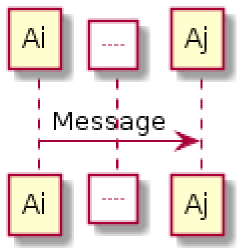
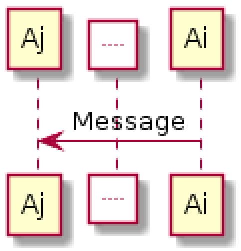
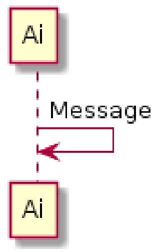
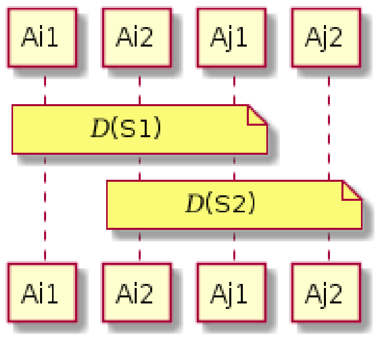
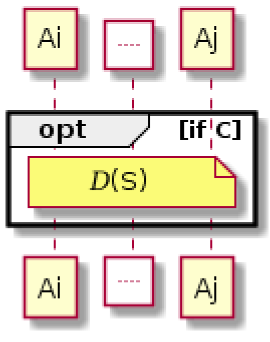
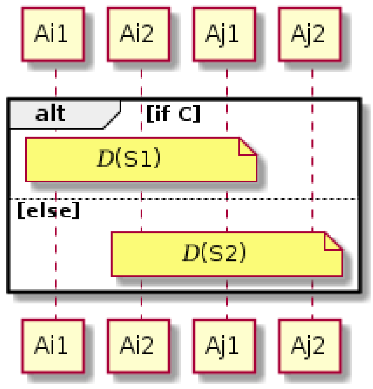
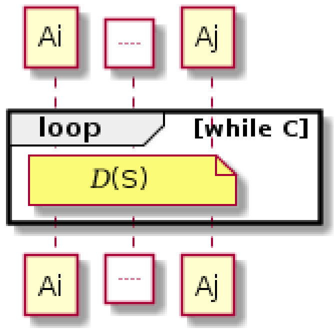
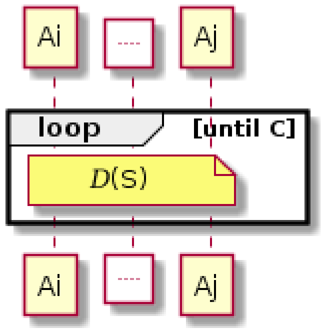
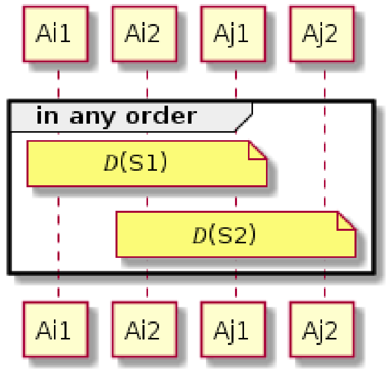
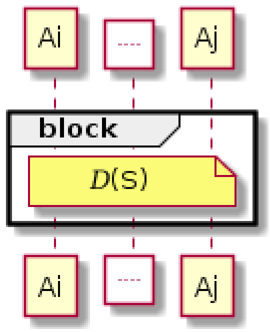
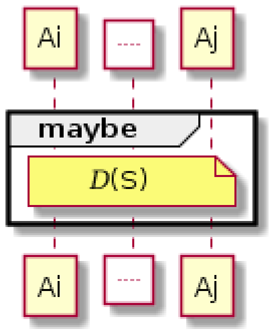
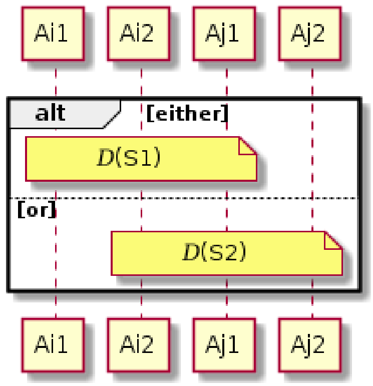
For each actor involved, including **System**, a graphical SSD contains a vertical dotted line below the actor. A message is displayed above a horizontal arrow from the line of the sending actor to the line of the receiving actor.

Below, A_i is the first mentioned (leftmost) actor of SSD S and A_j its last mentioned (rightmost) actor, A_{i1} is the leftmost actor of S1 and A_{j1} its rightmost actor, and A_{i2} is the leftmost actor of S2 and A_{j2} its rightmost actor.

We happily (mis)use the software tool PlantUML [Pla20], which turns textual representations into graphical representations, since its textual specification language closely resembles our textual specification language.

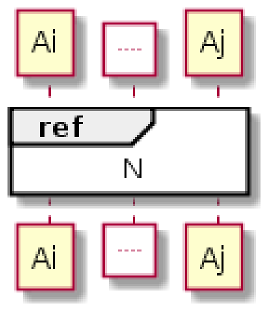
We start with 13 diagram structures for SSDs, followed by the diagram structure for a definition.

The diagram structures for SSDs

If actor A_i occurred earlier than A_j: D(A_i ➡ A_j: Message) ≝ 	If actor A_i occurred later than A_j: D(A_i ➡ A_j: Message) ≝ 	If A_i = A_j: D(A_i ➡ A_i: Message) ≝ 
D(S1; S2) ≝  Ai1 Ai2 Aj1 Aj2 D(S1) D(S2) Ai1 Ai2 Aj1 Aj2	D(**if** C **then** S **end**) ≝  Ai ---- Aj opt [if C] D(S) Ai ---- Aj	D(**if** C **then** S1 **else** S2 **end**) ≝  Ai1 Ai2 Aj1 Aj2 alt [if C] D(S1) [else] D(S2) Ai1 Ai2 Aj1 Aj2
D(**while** C **do** S **end**) ≝  Ai ---- Aj loop [while C] D(S) Ai ---- Aj	D(**repeat** S **until** C) ≝  Ai ---- Aj loop [until C] D(S) Ai ---- Aj	D(S1**,** S2) ≝  Ai1 Ai2 Aj1 Aj2 in any order D(S1) D(S2) Ai1 Ai2 Aj1 Aj2
D(**begin** S **end**) ≝ 	D(**maybe** S **end**) ≝ 	D(**either** S1 **or** S2 **end**) ≝  Ai1 Ai2 Aj1 Aj2 alt [either] D(S1) [or] D(S2) Ai1 Ai2 Aj1 Aj2

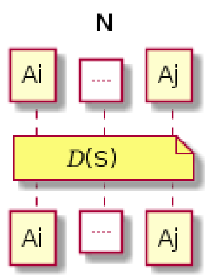
An Include/Call:

$D($ **do** N $) \overset{\text{def}}{=}$



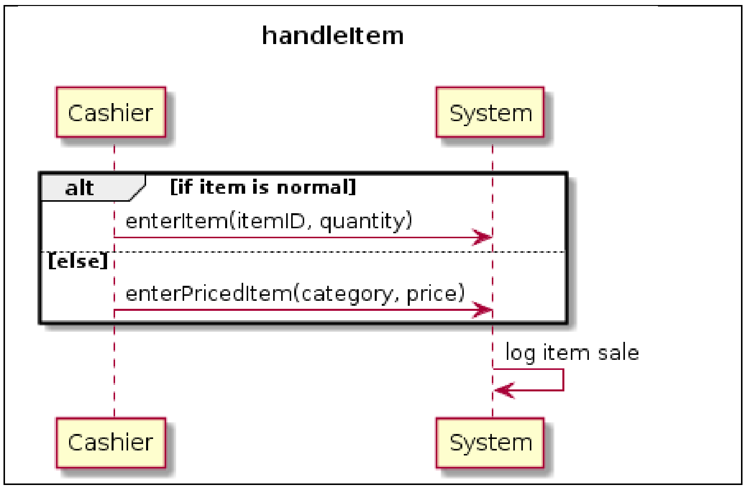
The diagram structure for a definition

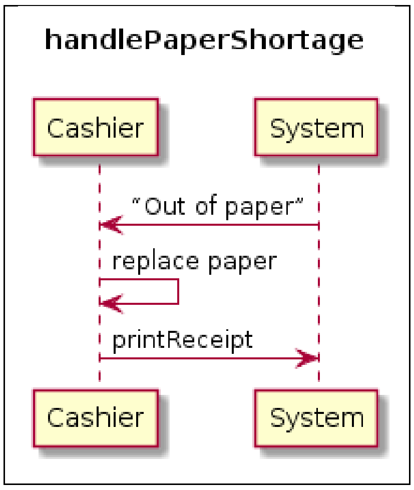
$D($ **define** N **as** S **end** $) \overset{\text{def}}{=}$

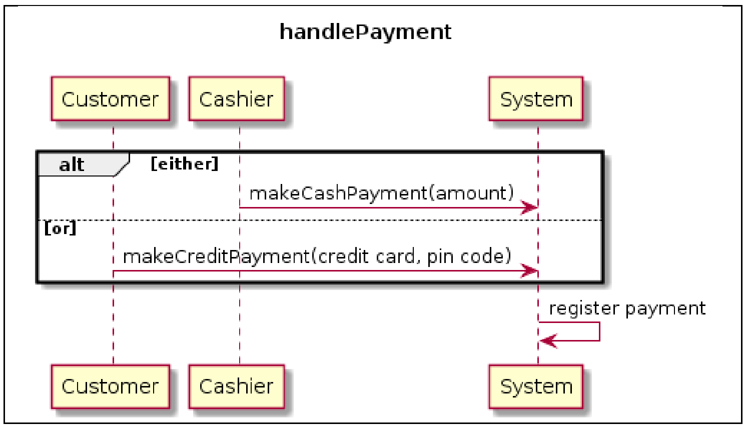


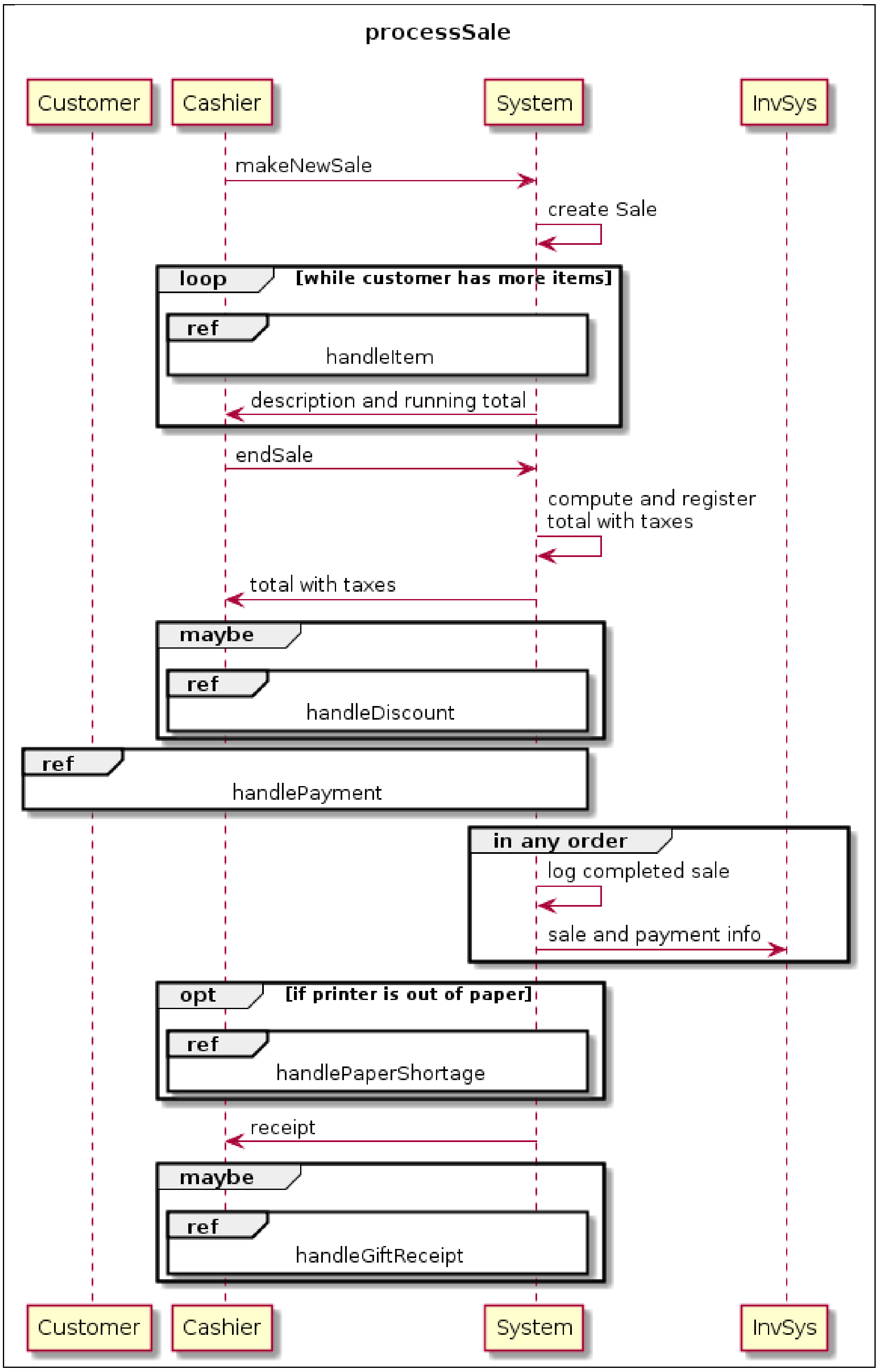
Example 3: Corresponding graphical SSD for *Process Sale*

Applying our translation rules to each of the 4 definitions worked out in Example 1, we get the next 4 diagrams (where for reasons of space, we start with the small ones):








processSale
Customer
Cashier
System
InvSys
makeNewSale
create Sale
loop
[while customer has more items]
ref
handleItem
description and running total
endSale
compute and register total with taxes
total with taxes
maybe
ref
handleDiscount
ref
handlePayment
in any order
log completed sale
sale and payment info
opt
[if printer is out of paper]
ref
handlePaperShortage
receipt
maybe
ref
handleGiftReceipt
Customer
Cashier
System
InvSys

Results and conclusion

The paper defines a formal grammar for textual SSDs. In the SSD, the Main Success Scenario plus the separate Alternative Scenarios of a Use Case are integrated. A clear advantage of textual SSDs is their alignment towards (textual) computer programs. To support validation of the integrated SSD and check it with the user organization, the paper also provides inductive rules to translate SSDs to natural language and to graphical representations. (Inductive rules to generate graphical representations are quite rare.)

The paper describes novel technical solutions for the application of NLG technologies to RE-relevant artefacts and contributes to its theory. The paper includes additional solutions to the RE-problem of validation.

References

[Bro19] E.O. de Brock: An NL-based Foundation for Increased Traceability, Transparency, and Speed in Continuous Development of Information Systems. In: Proc. of NLP4RE (2019).

[Coc01] A. Cockburn: Writing Effective Use Cases. Addison Wesley (2001).

[Lar05] C. Larman: Applying UML and patterns. Pearson Education (2005).

[Luc16] G. Lucassen, F. Dalpiaz, J. van der Werf, S. Brinkkemper: The Use and Effectiveness of User Stories in Practice. In: Proc. of REFSQ, 205–222 (2016).

[Jac11] I. Jacobson et al: Use Case 2.0: The Guide to Succeeding with Use Cases. Ivar Jacobson International (2011).

[UC20] https://en.wikipedia.org/wiki/Use_case

[SSD20] https://en.wikipedia.org/wiki/System_sequence_diagram

[Pie89] F.T.A.M. Pieper: Data machines and interfaces. PhD thesis, TU Eindhoven (1989).

[Mea55] G.H. Mealy: A Method for Synthesizing Sequential Circuits. Bell System Technical Journal, 1045–1079 (1955).

[Coh04] M. Cohn: User Stories Applied: For Agile Software Development. Addison Wesley (2004).

[GF94] O.C.Z. Gotel, C.W. Finkelstein: An analysis of the requirements traceability problem. RE journal, 94-101 (1994).

[Cle12] J. Cleland-Huang et al: Software and Systems Traceability. Springer (2012).

[Bro20] E.O. de Brock: The semantics of actions and instructions, under review (2020).

[SEP17] Stanford Encyclopedia of Philosophy (in particular Compositionality), Stanford.

[Pla20] PlantUML: https://plantuml.com/

[Tiw15] S. Tiwari and A. Gupta, A systematic literature review of use case specifications research, Information and Software Technology, pp.128-158, 2015

All links were last accessed on 2020/02/28