Towards a Theory about Continuous Requirements Engineering for Information Systems

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. The problem we address in this paper is the question how to come from initial user wishes to a running system in a straightforward, transparent, modular, and agile manner. In particular, we want to develop a sound underlying theory that grounds engineering approaches that tackle this broad problem (the complete development path for functional requirements, from initial user wishes up to a running system). We develop a theory that crosses the boundaries of several (sub)disciplines, e.g., requirements engineering, machine theory, and (database) systems development.

Keywords: user story, use case, system sequence diagram, information machine, implementation, ANSI-SPARC three-level architecture, development path, incremental, continuous, requirements engineering, information system

Overview

Section 1 recalls the background notions **user story** (US), **use case** (UC) and **system sequence diagram** (SSD). Section 2 introduces the notion of an **information machine** (IM): An IM can receive an input and will then produce an output and might change its state. Section 3 depicts a transparent **development path** for functional requirements, from USs via UCs and SSDs to an IM. Section 4 discusses some aspects of **incremental requirements engineering** for information systems and then Section 5 treats **continuous requirements engineering** for information systems.

An information machine is a *blueprint*, and can have many different **implementations**, as Section 6 points out. That section also shows the relation with the fundamental **ANSI-SPARC three-level architecture**, but now extended from databases to information machines in general: USs, UCs and SSDs belong to the **external level**, an IM belongs to the **conceptual level**, and implementations of an IM belong to the **internal level**.

1 User Stories, Use Cases and System Sequence Diagrams

We first recall some background notions (but in the words as we want to look at them).

Informally speaking, a **user story** (US) is a 'wish' of a (future) user which the system should be able to fulfil (see [1] for instance), e.g., the wish '*Remove a student with a given student number*' of a university employee.

A **use case** (UC) is a text in natural language that describes the steps of a typical usage of the system (see [2] for instance). For the user story '*Remove a student with a given student number*' the use case could be as follows:

1. A university employee (the 'user') asks the system to remove a student with a given number
2. The system removes the student info (if the student was known to the system)
3. The system informs the employee that the system did it (or that the student number was unknown)

Initially, use cases can be produced by (future) users of the system, domain experts or (other) staff members, or officials from the organization for which the system has to be built. Initially written use cases might need to be improved (sharpened/enhanced/ detailed/completed) in order to clarify what the system should do exactly (and when). E.g., our sample use case could more clearly distinguish between the two cases whether or not the student was known to the system. A (business) analyst might help to produce sharpened versions of initially written use cases.

A **system sequence diagram** (SSD) of a use case is a 'diagram' that depicts the interaction between the primary actor (user), the system, and its supporting actors (if any), including the messages between them (see Chapter 10 of [3] for instance). An SSD is a kind of stylised UC that makes the prospective inputs, state changes, and outputs more explicit. Although SSDs can be drawn in much fancier ways (e.g., see [3-4]), we will only draw their bare essence. For example, for (the refined version of) the use case just given, the (stylized) SSD could look as follows:

- User → System: RemoveStudent(<number>);
- **if** the student number is known to the system
  then System → System: remove the student info;
  System → User: "Done"
  else System → User: "Unknown student number"

We distinguish 3 types of basic interaction steps (each present in the example above):
User → System: Elucidates the inputs the system can expect (*input step*)
System → User: Elucidates the outputs the system should produce (*output step*)
System → System: Elucidates the transitions the system should make (*transition step*)

2 Formal Modelling: Information Machines

For our next development step, we need the 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*)
- a set S (of *states*)
- a function G: S x I → O, mapping state-input pairs to the corresponding output
- a function T: S x I → S, mapping state-input pairs to the corresponding next state

The notation f: X → Y used above, indicates that f is a function with X as its domain and its range being a subset of Y. The notion of *information machine* is equivalent to

the notion of *data machine* in [5]. It is a – not necessarily finite – Mealy machine without a special start state (see [6]).

G is called the *output function* of the IM and T the *transition function* of the IM.

We could equivalently have chosen for one (combined) function: $F: I \rightarrow (S \rightarrow S \times O)$

In that case, each input leads to a function assigning a 'new' state and an output to an 'old' state.

If the system also communicates with supporting actors, say other systems, then we still distinguish three types of basic interaction steps, but with (primary) 'User' generalized to 'Actor', where an actor can be any other system. Expressed in terms of an information machine:

Actor → System: Elucidates the minimally needed set I of inputs of the IM
System → Actor: Elucidates the minimally needed set O of outputs and the output function G of the IM
System → System: Elucidates the minimally needed set S of states and the transition function T of the IM

For instance, our earlier '*Remove student*' example would lead to inputs of the form *RemoveStudent(<number>)* and to the outputs "Unknown student number" and "Done". Suppose that each state $s \in S$ contains as a component a student table STUD with an attribute NUMBER, then the condition that a student number *n* is known to the system translates to $n \in \{ t(\text{NUMBER}) \mid t \in s(\text{STUD}) \}$.

When $n \in \{ t(\text{NUMBER}) \mid t \in s(\text{STUD}) \}$ and the input i = RemoveStudent(*n*), then the output G(s, i) will be "Done" and the student table in the new state T(s, i) will be $\{ t \in s(\text{STUD}) \mid t(\text{NUMBER}) \neq n \}$. In the other cases output G(s, i) will be "Unknown student number" and the new state T(s, i) will be s, i.e. stay the same.

3 From User Stories via Use Cases and SSDs to an IM

Starting from the USs and the UCs via their corresponding SSDs we can define an IM. In a general scheme (where the arrows indicate what is input for what):

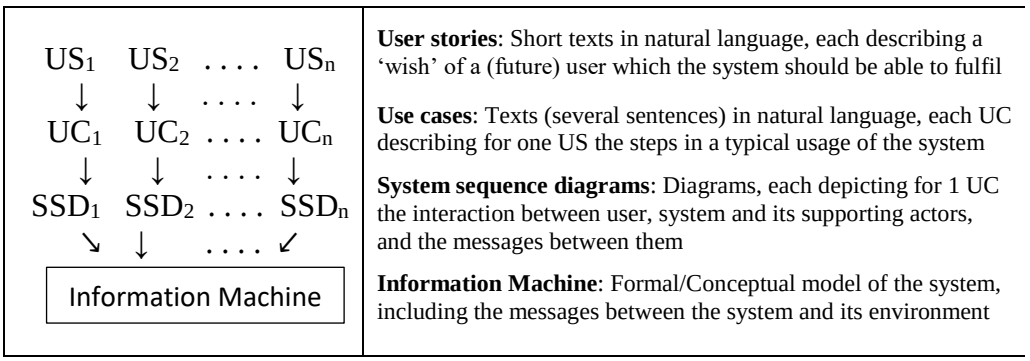

Fig. 1. The relation between user stories, use cases, system sequence diagrams and an IM

4 Incremental Requirements Engineering for Information Systems

Information machines in practice are really sophisticated, i.e., supporting a lot of use cases, resulting in very large input sets, output sets, state sets, and with complicated output functions and transition functions. Moreover, in practice such machines are often under continuous development ('under construction'), just as a city for instance.

Instead of defining and developing such a sophisticated machine in one go ('big bang'), including 'all' functionality that is needed – as might be suggested in Section 3 – since a few decades such machines are often defined and developed incrementally, i.e., starting with a simple, small version and extending/adapting it in several small steps into larger, more sophisticated versions.

Figure 2 indicates how an initial version of an IM might develop into newer versions. Note that, e.g. due to changing requirements, existing US/UC/SSD-triples might be adapted. (The "$^+$ " in US1$^+$ etc. indicate adaptions of earlier versions.) So, we see the addition of new US/UC/SSD-triples but also the adaption of earlier versions.

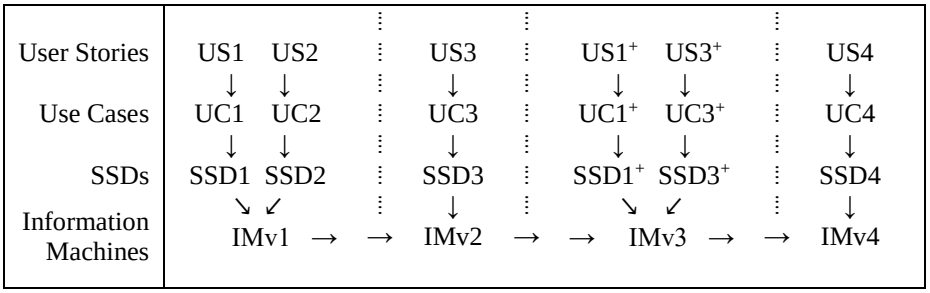

Fig. 2. Example of how an initial version of an IM might develop into newer versions

Figure 2 already indicates some structure in 'incremental requirements engineering': Via 1 or 2 USs, UCs and their corresponding SSDs (and the previous IM-version) we can define an initial (resp. next) version of the IM.

In Figure 3 each of the four basic functions known in the literature as **CRUD** (Create, Read, Update and Delete), a well-known acronym originating from [7], is illustrated by a user story.

Name	**Alternatively used names**
Create	Register, Add, Enter
Read	Retrieve, View, Show, Search
Update	Change, Modify, Edit, Alter
Delete	Remove, Destroy, Deactivate

Some sample user stories
Register a new student with a given name, address, (etc.)
Retrieve the info of all students with a grade average > 9
Change the name of a student with a given student number
Remove a student with a given student number

Fig. 3. The well-known CRUD-functions illustrated by user stories

We now treat incremental development of functional requirements for an information system more generally. Figure 2 can be generalized easily: Via a few USs, UCs and their corresponding SSDs (and the previous version of the IM) one can define an initial (resp. next) version of the IM:

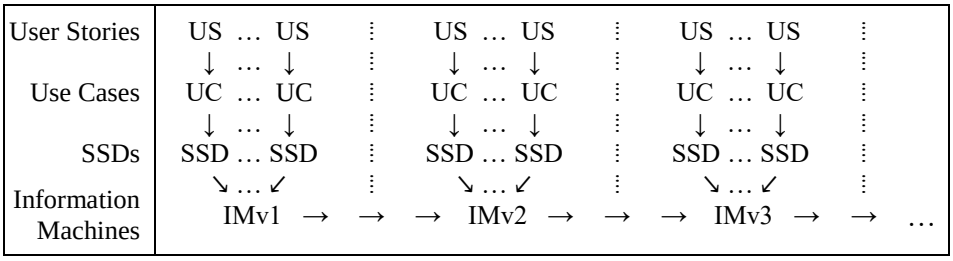

Fig. 4. How an IM can incrementally develop into newer versions

5 Continuous Requirements Engineering for Information Systems

Now we are heading towards *continuous* development of functional requirements. We note that such an incremental development of functional requirements can go on 'forever'. In a sense, such a development process is cyclic and can even be (almost) *continuous* during the lifecycle of an information system. We use the word *continuous* here when individual (or 'discrete') versions can hardly (or not) be distinguished anymore. Since we concentrate on development of functional requirements only, we are not talking about continuous *delivery* or continuous *deployment*, for example, but about continuous *requirements engineering*; see for instance [8] for those distinctions.

So, all in all, Figure 5 might be more appropriate:

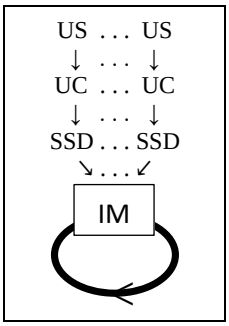

Fig. 5. How an IM can continuously develop into newer versions ('forever')

One cycle might contain only a few USs, UCs and their corresponding SSDs, or maybe even only *one* US, UC and SSD. Or, maybe even *less than* one full UC: In a more agile development process of functional requirements a simple 'core' scenario (or 'main success scenario') of a - maybe yet unclear - 'full' UC might be delivered first, followed by 'fuller' versions in subsequent cycles (see [3]). So, existing US/UC/SSD-triples might be adapted as well. Short cycles especially hold in case of *daily/nightly builds* (see [9]) and *continuous integration* (see [10]).

6 Realizations/Implementations of an Information Machine

Information machines can be considered as *blueprints*. An IM can have many different realizations/ implementations. For instance, an information machine can be realized by

a human servant (say a clerk), by an 'SQL servant' (i.e., a computer with SQL software), or by a 'Java servant' (i.e., a computer with Java software):

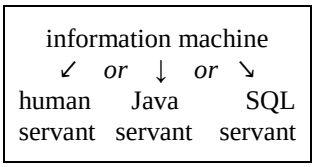

Fig. 6. Different kinds of realizations of an information machine

If we combine Figure 6 with the previous ones then we obtain the fundamental ANSI-SPARC three-level architecture, see e.g. [11], but now extended from databases to information machines in general:

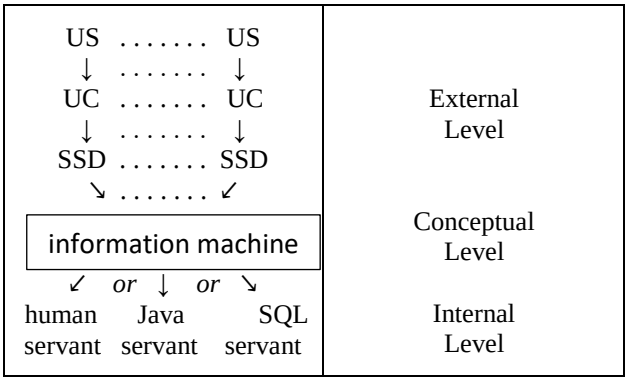

Fig. 7. Relation with the ANSI-SPARC three-level architecture

Relational SQL-based systems, for instance, are very suitable for incremental development, and since long they are used in this way. For a realization of user stories by means of an SQL-system for instance, we can use (stored) procedures. E.g., for our sample user story '*Remove a student with a given student number*' we can look back at the corresponding SSD in Section 1 and the details worked out in the last paragraph of the section on information machines (Section 2). In a naturally way this will lead to the SQL-procedure below. (Parameter names are preceded by an "@" in SQL.)

```
CREATE PROCEDURE  RemoveStudent @n INTEGER,
                  @output VARCHAR OUTPUT   AS
    IF @n IN (SELECT NUMBER FROM STUD)
    THEN  DELETE FROM STUD t WHERE t.NUMBER = @n;
          SELECT @output = "Done"
    ELSE  SELECT @output = "Unknown student number"
```

We are inclined to call the realization/implementation of an information machine an *information system*: According to the literature an information system has a Boundary, Users, Processors, Storage, Inputs, Outputs and Communication networks (see [12]).

The USs, UCs, SSDs, and the IM already shed light on the Boundary, Users, Inputs, and Outputs of the system under development. During continuous (and incremental) development of the functional requirements, these components will grow continuously.

Retrospection

We addressed the question how to come from initial user wishes to a running system in a straightforward, transparent, modular, and agile manner. In particular, we started to develop a sound underlying theory that grounds engineering approaches that tackle this broad problem (the complete development path for functional requirements from initial user wishes up to a running system). The theory we developed crosses the boundaries of several (sub)disciplines, e.g., requirements engineering, machine theory, and (database) systems development. We couldn't trace such an underlying theory that covers this broad problem completely (the whole development path for functional requirements, from initial user wishes up to a running system).

We placed the notions **user story**, **use case**, and **system sequence diagram** in line, and we linked the SSDs directly to the notion of an **information machine**: The set of SSDs of an application actually determine the *inputs*, the *outputs*, and the *output function* of the IM. By means of an example we showed how a user story directly leads to a set of inputs for an IM and, when the IM is implemented in SQL for instance, how such an input set in turn directly corresponds to a (stored) procedure. It indicates the modularity of the resulting system when developed in this way.

All in all, we started to develop a theory on *Continuous requirements engineering for information systems*, grounding some engineering approaches and crossing the boundaries of several (sub)disciplines, e.g., requirements engineering, machine theory, (database) systems development.

After discussing some aspects of *incremental* requirements engineering for ISs, we treated **continuous requirements engineering** for ISs. We also pointed out that an IM is a *blueprint*, and can have completely different **implementations**. We depicted a straightforward, transparent, and agile **development path** for functional requirements, from USs via UCs and SSDs to an IM, and then to an actual realization.

We depicted the relation with the fundamental **ANSI-SPARC three-level architecture**, but extended from databases to information machines in general: USs, UCs and SSDs belong to the **external level**, an IM belongs to the **conceptual level**, and implementations of an IM belong to the **internal level**.

Future Work

We want to extend our theory with additional, extended, and/or more complicated issues, such as sequences of inputs and corresponding outputs, complete induction for IMs (as a means to prove state properties of IMs), additional guidelines for developing use case texts, UC patterns, more complicated UCs and SSDs, further notions and terminology related to IMs, generalization and formalization of the CRUD-functions, dynamic constraints (i.e., constraints on state *transitions*), and interacting systems.

Acknowledgements. We thank the reviewers for their critical questions, which clearly helped to sharpen the paper.

References

1. G.G. Lucassen: Understanding User Stories. PhD thesis, Utrecht University (2017)
2. I. Jacobson et al: Use Case 2.0: The Guide to Succeeding with Use Cases. Ivar Jacobson Int. (2011) or https://en.wikipedia.org/wiki/Use_case
3. C. Larman: Applying UML and patterns. Addison Wesley Professional (2004)
4. https://en.wikipedia.org/wiki/System_sequence_diagram
5. F.T.A.M. Pieper: Data machines and interfaces. PhD thesis, TU Eindhoven (1989)
6. G.H. Mealy: A Method for Synthesizing Sequential Circuits. Bell System Technical Journal, 1045–1079 (1955) or https://en.wikipedia.org/wiki/Mealy_machine
7. J. Martin: Managing the Data-base Environment. Prentice Hall (1983) or https://en.wikipedia.org/wiki/Create,_read,_update_and_delete
8. P. Forbrig: Does Continuous Requirements Engineering need Continuous Software Engineering? REFSQ Workshops 2017, CEUR Workshop Proceedings 1796 (2017)
9. https://en.wikipedia.org/wiki/Daily_build
10. G. Booch: Object-oriented analysis and design with applications. Addison Wesley (1998) or https://en.wikipedia.org/wiki/Continuous_integration
11. ANSI/X3/SPARC Study Group on DBMS: Interim Report. ACM SIGMOD bulletin, vol. 7.2 (1975) or https://en.wikipedia.org/wiki/ANSI-SPARC_Architecture
12. L. Jessup and J. Valacich: Information Systems Today. Pearson (2008) or https://en.wikipedia.org/wiki/Information_system

All links were last accessed on 2018/02/16