Syllabus
Requirements Engineering
2008-2009

Dr. Stijn Hoppenbrouwers

NIII
Radboud University Nijmegen

1. Introduction

This is version 3.1 of the syllabus ("dictaat") for the Requirements Engineering course taught at the beginning of the second year of the BSc Information Science ("Informatiekunde") curriculum at Radboud University Nijmegen. In the course we also make intensive use of a textbook ("Use cases: requirements in context", by Kulak and Guiney; for more info see the course website at http://www.niii.ru.nl/~onderwijs/opleidingen/ - Cursussen op naam - Requirements Engineering). I may also use other texts, all of which will be made available on the website.

I draw, to some extent, on previous courses, notably "Domeinmodelleren", "Beweren & Bewijzen", and "modelleren van organisaties". However, the course has no officially set prequalifications.

This reader is a work in progress. It may be edited during the course, so make sure you have the latest version available, especially when you study for the exam. For the moment (and because my time is limited), I will only include teaching material without providing much of a context.

Please note that **you are required to read certain chapters of the Use Case book at an early stage in the course**, to prepare for the Case project. I will talk about the contents of the book only relatively briefly, and from my own particular perspective. You are expected to be thoroughly acquainted with the book's *full* contents. You really need to digest chapters 1-6 before you start working on the Case Project. If you do not read the book, that is your problem; I simply assume you do read it, and *in time*

1.1 Requirements Engineering: WHAT about it?

Simply put, Requirements Engineering (RE) is "the Art and Science of gathering and specifying what users and other relevant stakeholders demand and expect of a future software system".

> *"Basic system development thinking revolves round three main types of questions: WHY?, WHAT?, and HOW?"*

It all begins when someone wants something to be supported or made possible, and believes that creating some system will help achieve this. From the start, this includes HOW (the technical details of a specific, concrete solution), WHAT (the more abstract, essential functionality delivered by some system), and WHY (the problem or situation that is to be solved or improved).

So, presenting it the other way round:

WHY boils down to “what is the problem?” The problem could be, for example, “I hate that I cannot remember exactly where and when I traveled”.

WHAT relates to something that solves the problem. For example, “I could store information bout my travels somewhere, in a way that allows me to look it up and get to it later”. More in detail, it could be, for example, “to get a note block and pen and make a log of my travels, or to buy a palm pilot and make a digital log, or even to hire someone to follow you and keep records, so you can ask him later”. Note that there are usually some *alternative solutions* to a problem. In principle, even the finest detail in a solution is included in the WHAT. However, in this course we stop at some point. For example, details of interface (GUI) design are strictly speaking part of the WHAT, but are not covered by what is generally understood to be RE.

Another, very useful way of looking at the WHAT involves a *blackbox view*: describe what a system does (outside of the box: buttons to push and knobs to twist, or some more high-level description, and the order in which to do this to achieve some response), but not what the machinery inside the box looks like (the technical stuff that “makes it so”).

HOW makes concrete by what *means* the functionality is *delivered*. The blackbox is opened and we focus on what's happening inside. We cross into what is often called *implementation*: creating a real system that makes the functionality happen. We need to start programming, maybe buy or put together hardware, or even select actual people to perform tasks, and eventually we even have to make the whole thing operational (realization or deployment).

1.2 “Why-What-How^2”

I like to think about RE in a similar way. Basically, for RE you can simply ask: “WHY do we do it?”. The answer might be: “I hate that it often occurs that people involved in a software project don't agree about what it is they want, and that they did not consider it well enough beforehand, so people keep being disappointed or unpleasantly surprised”.

Once we know WHY, we can wonder WHAT exactly we want in RE, i.e. what we want to change in order to solve the problem. The RE WHAT-question then can be answered, for example, “I want a clear and detailed description of the wishes and needs of various stakeholders, documented in a certain way”. In more detail, they answer may tell you which sort of deliverables (documents) we want to end up with in answering the WHAT question for the project.

The HOW question in RE can be answered in much detail, and in many ways. What happens if you open the WHAT black box of RE and look under the hood? Which processes, plans, techniques etc. are supposed to get us where we want to be, i.e. get us the deliverables we want?

So put bluntly, RE is about the WHAT phase in system development (what will the system have to do for us?). It does however also require insight in the WHY (What is the problem we want to solve?), and often it also at the least involves an open eye for

the HOW (how can we build a solution, and is the system we want possible/affordable at all?). In other words: RE is always, somehow, *embedded in the larger context of system development*.

In chapter 1-3 of the textbook by Kulak & Guiney (I will refer to it as "[K&G]"), a good overview is given of the WHY (briefly), the WHAT (a lot), and a specific flavour of HOW of RE. Read the chapters and do it *timely*!

1.3 Some more about WHAT and HOW, and about genies and gnomes

Keeping apart (thinking about) WHY and WHAT and HOW is much more difficult than it may seem. First of all, people (you, me; stakeholders) are used to and taught to think very much in *solutions*. Thinking about WHAT often immediately triggers thinking about HOW. A whole set of jokes and stories about lamp-based ghost and genies are built on the relation between WHAT and HOW. For example, the following (politically incorrect) one:

> *A man walking along a beach stumbled across an old lamp. He picked it up, rubbed it, and out popped a genie. The genie said, "Okay...you released me from the lamp...blah, blah, blah. You get one wish!"*
>
> *The man sat and thought about it for a while and said, "I've always wanted to go to Hawaii, but I'm afraid to fly as I get a sick feeling within. Could you build me a bridge to Hawaii so I can drive over there to visit?"*
>
> *The genie laughed and said, "That's impossible. Think of the logistics of that! How would the supports ever reach the bottom of the Pacific? Also, think of how much concrete would be needed...how much steel!! No, you must think of another wish."*
>
> *The man said, "Okay," and tried to think of a really good wish. Finally, he said, "I've been married and divorced four times. My wives always said that I don't care about them and that I'm insensitive. So, I wish I could understand women, know how they feel inside, what they're thinking when they give me the silent treatment, know why they're crying, know what they really want when they say 'nothing,' know how to make them truly happy..."*
>
> *The genie said, "You want that bridge two lanes or four?"*

Thinking about HOW in direct relation to WHAT is alright and actually quite practical in many cases, but is hinders clear and thorough thinking about the WHAT in systems development: RE. This is mainly because it makes us skip any thoughts about *alternative* “WHATs”, and also about the tricky details of the WHAT. The WHAT has proven to be a rather difficult thing to get your head around; at first it sometimes feels as if there exists no WHAT, only a HOW. Do not be discouraged so easily; if you keep trying an essential WHAT always presents itself, and this will give you a distinct feeling of understanding the essentials.

For example: if I ask you “what does your word processor do for you”, chances are you can speak to me for quite some time about its features and what it looks like. However, if I push on and want to know what it really does *for you*, then things become more interesting. Does the machine help you write, or think, or get information across to other people? It somehow helps doing all of these things, but “functionality” often is very hard to talk about in a precise way (i.e. hard to *specify*); strangely, often much harder than the means deployed to get the job done. You assume that the Word Processor is the best way to go to get WHAT you want. Perhaps this is so. But in big and expensive systems development processes, things should not just be assumed or done lightly. In fact, people have become so fed up with the narrow-mindedness that can be caused by strict HOW-oriented development that they wanted to distinguish the WHAT from the HOW more explicitly –and came up with Requirements Engineering as a separate activity.

1.4 The Gnome Metaphor

I often use the following somewhat corny but effective metaphor to help myself separate WHAT from HOW. In case of doubt, imagine the HOW is dealt with by deploying gnomes (“kabouters”), who can manage just about anything you ask them. This gets rid of the HOW aspect and leaves you with the WHAT. In the Word Processor case, imagine you have a very literal-minded gnome living in a drawer of your desk who you can call on day and night to do for you exactly what your word processor would do for you. How (in exactly which words!) would you ask the gnome for help the first time you want this kind of support from him? How do you explain to him exactly WHAT you want, leaving the HOW to him? And most interestingly: can you think of any different THATs that you would actually like your word processor/gnome to deliver, that are not typically supported by current word processors? Can you come up with some truly new functionality? Then your fortune might be made (that is, if good old MS buys your idea rather than “borrowing” it).

1.5 WHAT before HOW, or what?

Separating HOW from WHAT is not the only thing that RE is about (as I said, WHY and even HOW do come in somewhere), but it is a key aspect of the “art” of RE. It may well be the case that you do need to think and talk about HOW in RE. Dealing with HOW is not *forbidden* as such! You just need to be able to *separate* it from WHAT, in your mind and on paper.

Common sense tells us that it is a good thing to think about WHAT before HOW (and even about WHY before WHAT). However, as we've seen in the joke, if you know in advanced that WHAT you want cannot be done, you might as well forget about it right away. This is why RE is not just about what everyone wants, it is ultimately also about finding a *realistic* package of requirements that everyone involved can agree on. This requires solid interaction with the people who do the building. In [K&G], the assumption largely is that RE takes place in close relation to actual software building. This is a terrific idea in real life, but in this course and our Case Project we will unfortunately not be able to simulate such a situation. Therefore, we are stuck with

WHAT without HOW. Still, you may be able to think about HOW at least a bit. As long as you keep them separated, and get your priorities right!

There is another reason why WHAT before HOW can seem a good idea: in case promises are made, even legal promises (contracts). The "contract-style requirements lists" that [K&G] dislike so much (for communicative reasons!) are used exactly for this. However, one always has to be very careful about promising things that cannot, in fact, be done. It may seem cunning to get people to promise things they cannot deliver (making them pay fines and so on), but in most practical cases, in the end this means *nobody* gets what they want, and everyone looses. And in many cases, this indeed happens (in class, I may tell you some rather sad stories about this).

1.6 RE and "design"

Often things are said about the relationship between RE and "design"; this course is no exception. On a general basis, it is important to point out that there are (at least) two different ways of interpreting "design" in this context, and that you encounter both all the time, and that this may cause great confusion. Most importantly, in the textbook the term "design" is used in a rather limited way that I find a bit unfortunate and confusing.

[K&G] tell you that RE is a system development activity that is strictly separate from Design. They use "design" quite strictly in the sense of "technical design" (supposedly HOW-related). However, there is also such a thing as "functional design", and if you come up with anything new in RE (for example work out a detail that a stakeholder did not come up with by herself), or make any "requirements choice" at all, you are in fact doing functional design. So if you are doing RE (apart from the pure one-way stakeholder-oriented information gathering bit, perhaps), doing some sort of design is in fact unavoidable. Keep this in mind, not just to avoid confusion about terminology, but also because it is important to realize that though you need much input from stakeholders, RE is still a creative activity in which the requirements engineer (that is you!) is *actively* involved and makes many small decisions.

1.7 Some notes about the textbook and this course

In Chapter 1 of the textbook, emphasis is put on two not-so-successful aspects of RE as it has been practiced:

- Contract-style requirements lists
- Prototypes

Such lists and prototypes are the chief arguments for [K&G] to prefer another approach: Use Cases. The book sometimes seems to be a bit of a crusade against lists and prototypes. While I do not disagree with [K&G] about their disadvantages, I would like to make it clear that neither contract style lists nor prototypes are "bad" in principle; in some cases they *are* excellent means to get some job done. However, in RE they are perhaps not the best things to put *central*. [K&G] mostly react to other,

existing “styles” of RE. In other words, other people have created and advocated whole approaches and methodologies for RE that are based on requirements lists or prototypes, and [K&G] try to make a strong case (American style) for Use Cases, so they have to say what they don't like about other popular approaches.

I do believe Use Cases are a great technique to put central in RE. However, I also take the liberty to add to the [K&G] approach some other techniques and ideas, fitting Use Cases snugly into a methodology that better suits the Nijmegen approach to systems development. So you'll have to be aware of the mix of two slightly different flavours of RE in this course: RE à la Kulak & Guiney and RE à la Hoppenbrouwers/ICIS[1]. We present the two flavours in the conviction that together, they work well –like an ice cream cone with two complementary flavours. Please be aware of the distinct flavours, however, since it will vastly improve your understanding of the fine points of the course content.

2. The structure of the final requirements deliverable

In this section I will *briefly* present the main deliverables we expect in the final requirements document of the Case Project, and the relationships between some of them. All but a few of the deliverables mentioned are also mentioned in [K&G]. However:

- I have added a number of items
- I am stricter than [K&G] about how various items (in particular the “key items”) relate to each other
- Though the documentation is essentially “informal”, in terms of coherence and consistency, I demand a standard that is almost formal (i.e. on par with mathematical texts). Your documentation should be a tightly interconnected set of sub-deliverables, with consistent concepts/terminology and no dangling ends. I am really serious about this.

Now let us have a look at all main items we expect in the requirements deliverable. The items mentioned can be seen as *required sections in the documentation*.

In chapter 3, we will return to the key deliverables (a subset of the deliverables discussed in chapter 2) and elaborate on *how to create them*. This is a somewhat different perspective.

2.1 About the “Introduction”

There is not much I say here about the Introduction. The main point I want to make is that the introduction of a report like the one you are producing has a clear functionality: to set the scene and provide a clear context for what follows. This

[1] The “Institute for Computing and Information Sciences” of Radboud University Nijmegen

means that it matters what the *audience* for the report is, what they *want*, and what they *already know or do not know*. Be relevant. To be blunt about it: actual bla bla will not be tolerated. An introduction is not just a header with some semi-random entertaining text underneath it. But on the other hand do not leave out essential stuff that belongs in a requirements document even if everybody involves already knows it. So be both *relevant* and *complete*. This is sometimes hard, but nobody ever told you it was going to be easy.

2.2 About the "Problem Statement"

This is the WHY bit of the case project, put in a somewhat negative form: what is "wrong"? What should change? As holds for all items: do not hesitate or forget to update this section as the project progresses.

2.3 About the "Stakeholder Analysis"

This includes items like "**user demography**" (what *types* of stakeholders (roles played) do you encounter in this case, and "**stakeholder list**" (actual, named stakeholders, i.e. specific people and the role(s) they play.). If there are only few stakeholders, that's fine; just provide a clear-cut and *relevant* listing and description of the stakeholders (users *and other* relevant stakeholders!).

2.4 About the "Mission and Vision (and Values)" section

This is a difficult section. It isn't even all that important, but I want you to have a serious go at it anyway. The information captured in it mostly comes from what [K&G] call the Chief Executive Sponsor (in the case study, that's probably me), but you should help him/her formulate it and at the very least you should somehow show you really understand it. As to the (often misunderstood) differences between the three items [K&G p56]:

- Mission— What the project will do (close to WHY)
- Vision— What the end product will be (close to WHAT)
- Value— What principles will guide the project members while they do what they will do and build what will be; the main rules of the game played.

In particular the "values" are almost impossible to make sense of in the limited context of our course and case study. I am therefore willing to reduce the item to "Mission and Vision".

2.5 About the "Statement of Work"

This is a tricky bit to include in the *final* deliverable, because the SoW changes as the project progresses, and is really obsolete once the project finishes. It is, in other words, a project management item. Try to seriously create a work estimation and

planning, also for your won sake, but to be honest a Statement of Work is not a very realistic item in our setting and is of limited importance.

2.6 About the "Risk Analysis"

For this, pretty much the same holds as for Statement of Work. Try to seriously describe some risks involved in your project without spending too much time on this. Note that this Risk Analysis is purely about *project risks*: risks that *you* do not make my demands within this case project. So it is *not* about risks inherent in the system that eventually may be delivered.

2.7 About the "Use Case Survey"

Now we're getting to the more important bits! The use case survey provides an integral overview of all use cases in the documentation. [K&G] are not always very clear about this. Some people are confused about the difference between the use case survey and actual use cases (each one of them structured by means of the use case *template*). In fact, there is a little bit of overlap, but they are very distinctly different deliverables. The survey mostly helps you keep the overview over the collection of use cases you make, especially in the early phases of the project. Here are the items it should contain:

- **Use case number**
- **Use case name**
- **Initiating Actor** Note that this is a subtype of the generic "actor" used in use case diagrams.
- **Description**. Elswhere in [K&G] also called "summary". Once you also have an actual use case, this should be an exact copy of the descriptions in each corresponding individual use case. If you do not have a proper use case yet, the description is the only `content' you have for it.
- **Completeness**. How complete is this use case at this point? In the final deliverable, completeness must be "full", but in earlier stages this will mostly be different.
- **Maturity**. Much like `completeness', but this concerns how well thought through the use case is.
- **Dependency** concerns on which items / aspects the use case is dependent. This is often a hard item to come up with; only include it if it is relevant, otherwise you can leave it out.
- **Source**: where did the information on which you base this use case come from? How much did you make up yourself?
- **Comments**: anything important but not covered by other items above.

A word of caution: `use case complexity', `architectural priority', and `business priority' are mentioned by [K&G] but you can leave them out.

In addition, you have to add the **use case diagram here, that includes (integrates) all use cases**, so the relationship between them can be viewed at a glance.

2.8 About the "Use Cases"

This is, of course, the *pièce de résistance* of our requirements document. Quite a lot is said about it in [K&G], thoughout the book. Here I will just give some brief comments on the "use case template" as presented below, in particular with respect to where we take a different point of view from that of [K&G].:

- **Use Case Diagram**: include a use case diagram for each use case before its textual description. Also see 2.7.
- **Use case name**: the name of the use case; choose it well! (see p87 [K&G]; "verb filter"). Do not be afraid to change the name at some point in your project (keep all documentation consistent!) if you come up with an improved, more meaningful name along the way.
- **Iteration**: which phase (filled or focused), which version (if you do versioning, which is recommendable).
- **Description (a.k.a. summary)**. Should always be *identical* to corresponding description in the use case survey.
- **Basic course of events (BCoE)**. The heart of a use case. The stepwise story of the interaction between actor and system.
- **Alternative paths**. In [K&G], these are intended to keep things "simple" from a user perspective; according to them, if you were allowed to provide a complex BCoE with "IF-THEN-ELSE" like structures, you would not even need the alternative paths. In [K&G] they are supposed to be used to write out clearly all important IF-THEN-ELSE path individually.
- **Exception paths**: "error handling" though never in terms of actual "error" messages (which are a "how" thing). They capture what interaction path happens if things go "not as intended".
- **Extension points**: only apply if you use extensions in your use case diagram. We do not find extension points very important.
- **Triggers**: what sets the use case in motion? The initial move of some actor? A moment in time being reached? Something going wrong? Note that this is not the same at all as a precondition!

Assumptions: relevant things to know that, however, do not apply to the system as such. Compare it with the *informal* assumptions that you were supposed to include in your Beweren & Bewijzen project in the first year. They are the opposite of Preconditions, in this respect. If assumptions are not clearly there, leave them out!

Preconditions are very explicit things that should be the case at the beginning of a BCoA, the "initial state". They can be compared to "formalized assumptions" (small and big "A's") in the Beweren en Bewijzen correctness statement (correctheidsstelling).

Postconditions can be compared to "formalized commitments" (small and big "C's") in the Beweren en Bewijzen correctness statement (correctheidsstelling).

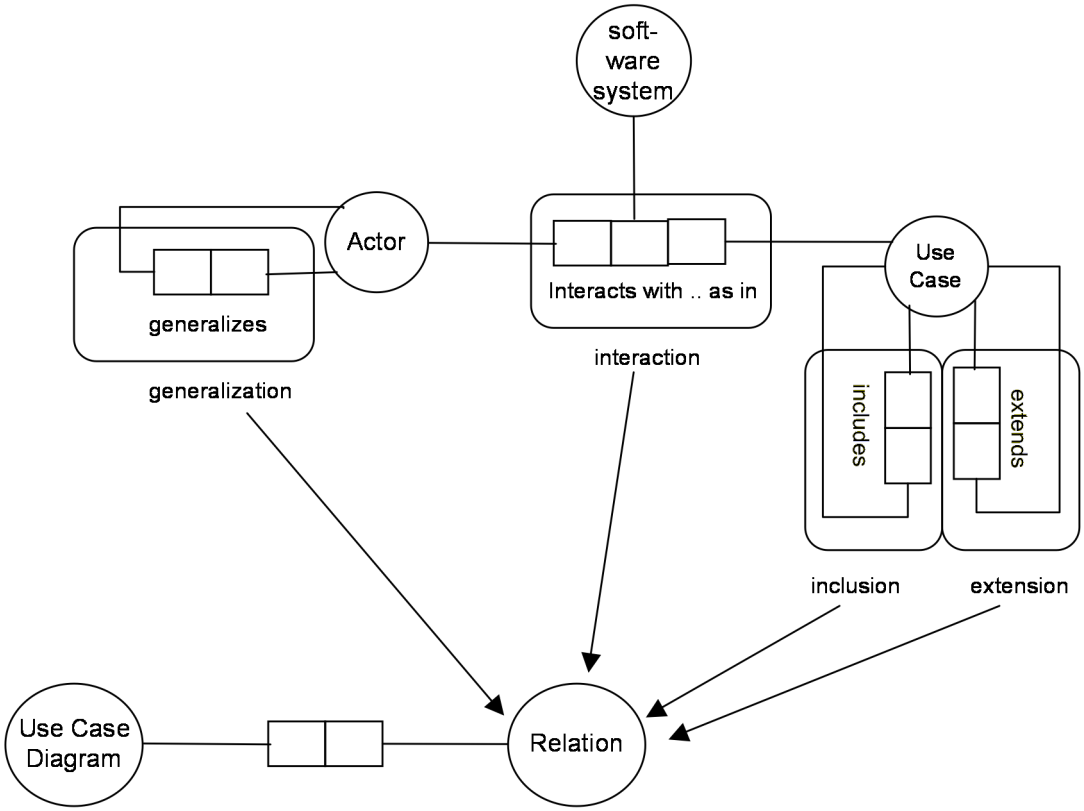
Related business rules. Clearly written in semi-formal language, but in relevant cases formalized in ORM and/or First Order Logic (this sometimes helps a lot to get things clearly specified!). [K&G] just talk about business rules that happen to be around anyway. I push it a lot further. I want you to formulate the relevant business

rules, from scratch if need be, possibly based on the domain model of the use case. How this can be done I will explain in chapter 3.

Author(s). Simply the author(s) of the use case.

Dates. Date of initial creation and dates of consecutive changes (possibly integrated with versioning).

Below, you can see an ORM conceptual model of my view on what basic (not all!) concepts are involved in use cases, and in particular use case diagrams.



2.9 About the "Scenarios"

Scenarios are concrete, instance-level descriptions of how a use case works.

Scenarios are mostly related to the BCoEs of use cases. A separate scenario has to cover each alternative path and exception path of a BCoE. So there will usually be *various scenarios underlying one use case* (n:1). Use these to *test* the various possible paths that a use case may take (tests are not explicit deliverables, but this does not mean you do not have to perform them! They are an important means of guaranteeing the quality of your use cases.)

If possible, do try and keep similarities visible between the structure of the BCoE/Alt Paths/Exc Paths and the structure of their matching scenarios. Also keep terminology/concepts consistent across use cases and scenarios.

If you use a fact-based approach to requirements engineering (i.e. looking at the instance level first), you may actually get some scenarios before you get use cases. Mostly, however, you will come up with scenarios afterwards. Note that for ORM-style domain models, their populations should be in line with the scenarios, since these both concern instances.

Above I describe the use of scenarios as end products in the requirements document. There is, however, a less formal use of scenarios, more oriented towards requirements *gathering*. In such cases, you ask a stakeholder to describe (at instance level, though you cannot normally ask them this literally), what they do when they perform a certain task. They describe an example of a task. This task may well cover various use cases! This means that you will have to cut the scenario up later. This kind of scenario is welcome in the documentation, but please keep them well apart from the deliverable type of scenario.

2.10 About the "Domain models"

[K&G] do not use Domain Models in their flavor of RE (though they mention UML Class Diagrams); we do want to see domain models, in ORM to be specific. In principle, UML class diagrams or ER diagrams might also be used here, but you all know ORM from the domain modeling course and we like it a lot, because:

- It has a completely formal basis; you could, for example, use ORM definitions directly to generate software (research concerning this is in fact being done within our institute, though not by me).
- It is fact-based: it includes populations (instance level) in its conceptual meta-framework and in its procedures
- It is, if used in full, much richer than UML class diagrams or even ER diagrams, but you do not necessarily have to use stuff like constraints, populations, type names, verbalizations, etc. straight away; you can build it up gradually, and even leave stuff out
- ORM is the basis of much other material we teach about systems development and modeling; using it makes the RE course more compatible with various other courses.

As for the use of ORM diagrams in Requirements Engineering:

- ORM is already widely used as a technique in RE, even in combination with use cases
- Use cases are very useful, but they just fail to provide information in enough detail to be able to hand over the requirements to the technical design people and say: OK guys, you build this please. Domain models in ORM provide just the last few inches of specification we like to see.

- Again, making (complete and fully specified) ORM diagrams really amounts to formalization. This we consider useful and important as a basis for sound functional and technical design.

So to make things absolutely clear: I do expect full ORM diagrams (including basic constraints!) of all use cases. However, we also expect that only a limited number of concepts occurring in the use cases is relevant for ORM modeling, and therefore that the actual models are not so complex. The point is: they have to be there, even if they are simple. Also provide some **population examples** in line with your scenarios! Example populations are not currently defined as a separate deliverable, but I am actually considering to do so. Note once more that the example population of a DM is closely related to the scenarios of a use case to which the DM belongs.

As a rule of thumb concerning "what concepts are relevant for ORM modeling in RE": we are primarily interested in domain models of the actual concepts *used in interaction of the user with the system*. In other words: concepts belonging to the UoD of the user as she interacts with the system. In still other words: important aspects of the interface language used when actors and stem interacts. It is likely that this set is extended with some related concepts that are in the user's UoD, in particular those needed to *formulate business rules*.

Concepts that should *not* be modeled are those that are only used in communication about the system (i.e. are in now way part of the user's UoD or the business rules). It could be useful to model them as well, but in the current course that would go too far. So don't!

2.11 About the "Business Rules"

The **business rule catalog** as suggested by [K&G] is quite appropriate. You are, however, obliged to semi-formalize the business rules. With your background knowledge, it should be easy to do so, especially if you already have an ORM domain model. Also note that formalizing business rules is often required in other system development and management activities: see the Business Rules Manifesto (placed on the RE website).

Interestingly, ORM is now one of the main techniques worldwide used to capture (formal) business rules, or at least the most basic, elementary rules in a domain.

In general, our suggestion is you that keep to the type of business rules catalog suggested by [K&G, p60-2], but also

- Make sure the terms you use when formulating the actual rules are compatible with the relevant ORM domain models.
- Use clearly structured, unambiguous sentences language with clearly indicated logical or mathematical operators like AND, OR/XOR, IF, THEN, NOT, <,+,-, etc.
- Always *also* include a formulation in plain natural language

Crucially, you only should include business rules explicitly related to some use case you describe. Also, as indicated, business rules may or may not have been explicitly formulated before your analysis started, but in principle, every organization works according to some rules. It is up to you to find and formulate them!

Finally, please do not be confused by the "business" in "business rules". Perhaps a better, more general term would be: "domain rules". They describe in considerable detail what should and should not be done in the organization (the environment in which the system-to-be-built will function, and which it will support). For example, consider a library. A general rule could be: "items borrowed have to be returned within 21 days from the day on which they were lent out, unless within 21 days from being lent out the loan is extended by the person borrowing the extended". Next, there may be rules that define when extension is possible or not, and so on. Note that all meaningful concepts in the rule have to be included in the DM of the use case(s) to which this rule is relevant, and also that for the rule to be properly modeled as a business rule, some semi-formalization will be required. Creating a DM for the rule is a good way of starting semi-formalization.

2.12 About the "Non-functional Requirements"

[K&G] claim that use cases are a good technique for capturing non-functional requirements. I do not agree. It might be possible to use use cases for non-functionals, but it seems to us a bit forced. Indeed, in the examples of non-functionals (the "-illities" etc.), [K&G] do themselves not use use cases. So the main advice here is: keep to the practices recommended in book when it come to non-functionals, but do not formulate them as use cases. You can find much more on non-functionals in the book.

2.13 About the "Terminological Definitions"

We expect all important terms in the use cases/domain models to be properly defined. ORM models do *not* define terms as such: they provide highly contextualized type-level descriptions of which terms occur, for example as "cats hate dogs". ORM diagrams say nothing about what "dog" or "cat" or "hate" mean as independent words. For this, terminological definitions are required. Together, these definitions can be called many things, e.g. dictionary, glossary, lexicon, vocabulary, ontology, terminology, and so on. We stick to the latter word: terminology.

Our minimal expectation for the terminology in your requirements document is a list of key terms (preferably, all terms that also occur in your domain models) and clear and useful natural language definitions with them. You can use existing dictionary definitions if you like (please provide source references), but do be careful: standard dictionaries have many limitations and they may not describe the exact meaning of some term that the stakeholder actually means/uses in the UoD. Often, it is much safer to write your own definition that is specially aimed at the specific context you are working in. If necessary, get information form stakeholders. After all, both you and the stakeholders can be expected to know what they mean with a certain word; if not, work on it.

Traditional term definitions have a simple format: a **definiens** (=that which is defined), a **genus** (the "supertype" of what is defined), and one or more **differentiae**: what distinguishes the definiens from other subtypes of its supertype. So for example, a dog (definiens) is an animal (genus) that can bark (first differentium) and wags its tail a lot (second differentium). Please note that these relations could be expressed in an ORM like manner, but that would go too far since terms like bark, wag, tail etc. probably do not as such occur in the UoD. So simply keep to the traditional definition in natural language.

In particular in the field of Business Rules Specification, there is a brand new initiative to find ways to better specify and communicate about the precise meaning of rules and terms. This initiative goes by the name SBVR: Semantics of Business Vocabulary and Rules. You could say it encompasses terminiological definitions, ORM, and rules. For a brief introduction to SBVR, see the SBVR-1 file on the RE website.

3. How to *create* the key deliverables

I am currently cooperating with Jeroen Roelofs (who's doing his Master's thesis on it) in working out a detailed process description that can help you in finding out *how* you can get the deliverables. It is a layered step-by-step description. I wish I had the time to write this chapter as a perfect fit to the course, but in this version the best I can do is include relevant paragraphs from a book chapter that is in print (to appear 2009).

IMPLEMENTING GOALS AND STRATEGIES IN A CONCRETE WORKFLOW LANGUAGE

In this section, we show how the framework presented thus far has been used in the implementation of a reference method for requirements modelling as taught in the 2nd year Requirements Engineering course of the BSc Information Science curriculum at Radboud University Nijmegen. Please note that the method as such is not subject to discussion in this paper, just the way of describing it. This section is based on work by Jeroen Roelofs (Roelofs, 2007). The original work focused strictly on strategy description; in this paper, some examples of related goal specification are added. The strategy description was implemented as a simple but effective web-based hypertext document that allows "clicking your way through various layers and sub-strategies" in the model (see below).

Case study and example: requirements modeling course method

The main goal behind the modelling of strategies of the case method was to provide a semi-formal, clear structuring and representation thereof that was *usable for reference purposes*. This means that the rule-based nature of the framework was played down, in favour of a clear and usable representation. A crucial step (and a deviation of the initial framework) was taken by replacing the plain directed graphs used so far by workflow-style models in the formal YAWL language (Yet Another Workflow Language: van der Aalst and ter Hofstede, 2005). Below we show the basic concepts

of YAWL (graphically expressed), which were quite sufficient for our purposes. We trust the reader will require no further explanation.

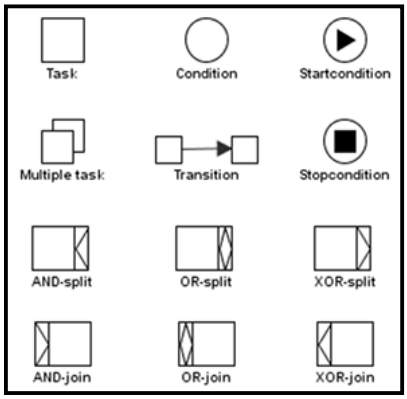


Figure 1: basic graphic concepts of YAWL

The main (top) context of the method is depicted in the following schema:

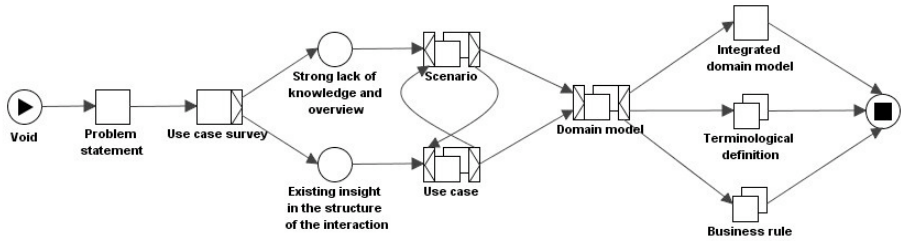


Figure 2: The top strategy context "Create a Requirements Model"

Note that in Figure 5, the square-based YAWL symbols correspond to the QoMo strategy framework in that they represent states (goals/situations). The actual strategies match the arrows between states: the actions to be taken to effectuate the transitions between states. In other words, the diagram is a very concise way of representing a strategy *context*. A useful operational addition to the framework is the use of conditions (circles) for choosing a goal (in going from "Use Case Survey" to either "Scenario" or "Use Case"): this was explicitly part of the existing method and possible in YAWL, and therefore gratefully taken aboard.

All arrows in the diagram have been labelled with *activity names* (which are another addition to the framework). Underlying the activities, there are strategies, which in turn consist of one or more *steps* (another addition). The complete strategy description of the activity "create requirements model" which is graphically captured by the top context (fig. 5) is the following:

1. **Create problem statement**
2. **Create use case survey**
3. **Create use case based on use case survey AND create scenario based on use case**
3. **Create scenario based on use case survey AND create use case based on scenario**
4. **Create domain model based on use case**
4. **Create domain model based on scenario**

5. **Create terminological definition**
6. **Create business rule**
7. **Create integrated domain model**

All steps listed are represented in boldface, which indicates they have underlying *composed* strategies (which implies that each step is linked to a further activity which is in turn linked to an underlying strategy). Concretely, this means that in the hypertext version of the description, all steps are clickable and reveal a new strategy context for each deeper activity. For example, if "**Create domain model based on use case"** is clicked, a new (rather smaller) context diagram in YAWL is shown, with further refinement of what steps to take (strategy description). We will get back to this particular strategy, but before we do this, some explanation is in order concerning the irregular numbering of steps above. The occasional repetition of numbers (3. 3. and 4. 4.) serves to match the textual description with the YAWL diagram: the XOR split and AND-join in figure 5. In addition, the two possible combinations of steps before the AND-join needed to be combined using an "AND" operator, but note that the activities linked by AND are separately clickable.

Let us now return to the "**Create domain model based on use case**" strategy. It concerns the creation of a "Domain Model" (ORM) based on a "Use Case", which (roughly in line with previous examples) boils down to a basic description of steps in making an ORM diagram based on the interaction between user and system that is described stepwise in the use case (please note the participants in the course are familiar with ORM modelling and therefore need only a sketchy reference process description). The related strategy context is a fragment of the one in the top context:

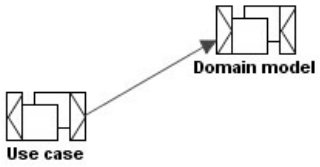


Figure 3: another strategy context –"create domain model based on use case"

Apart from this context, the underlying strategy is shown:

1. **Identify relevant type concepts in use case**
2. **Create fact types**
3. Create example population
   - Make sure the example population is consistent with the related scenario(s)
4. Make constraints complete

Steps one and two, represented in boldface, by way of more activities refer to more compositional strategies, so they are clickable and each have an underlying strategy. Activities 3. and 4. are represented differently, respectively signifying a *guided strategy* (underlined and with additional bulleted remark) and an *ad hoc strategy* (normal representation). A guided strategy is a strategy of which a description of some sort is available that helps execute it. In the example, this guidance is quite minimal: simply the advice to "Make sure the example population is consistent with the related scenario(s)". In view of our general framework, this guidance could have

been anything, e.g. a complex process description or even an instruction video, but crucially it would not be part of the compositional structure. In context of our case method, we found that a few bulleted remarks did nicely.

There still is the ad hoc strategy linked to the activity name "Make constraints complete" (step 4.). It simply leaves the execution of the activity entirely up to the executor. As explained, it is an "empty strategy" –which is by no means a useless concept because it entails an explicit decision to allow/force the executing actor to make up her own mind about the way they achieve the (sub)goal.

In addition to the strategy context diagrams and the textual strategy descriptions, the hypertext description provided a conceptual diagram (in ORM) for each strategy, giving additional and crucial insights in concepts mentioned in the strategy and relations between them. The ORM diagram complementing the "create domain model based on use case" strategy is given in figure 7i. In context of the case, the inclusion of such a diagram had the immediate purpose of clarifying and elaborating on the main concepts used in the strategy description. In a wider context, and more in line with the more ambitious goals of the general strategy framework, the ORM diagram provides an excellent basis for the creation of formal rules capturing creation goals. We will discuss an extension to that wider context in the next section.

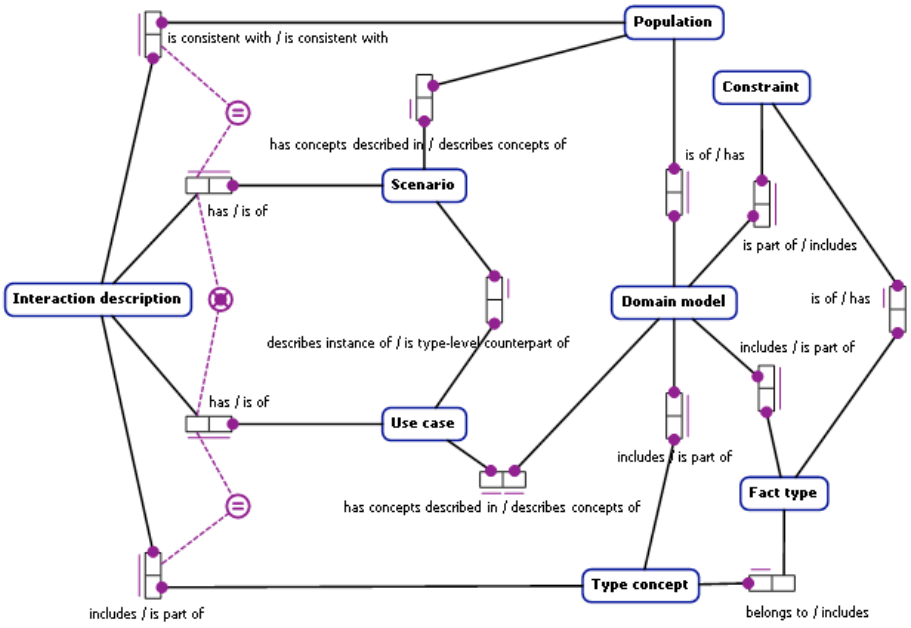


Figure 4: ORM diagram complementing the strategy description

Goal and procedure rules added to the case

The case strategy description as worked out in detail by Roelofs (2007) stops at providing a workable, well-structured description of the interlinked strategies and concepts of a specific method. Though has been found useful in education, the main aim of creating the description was to test the QoMo strategy framework. However, it could in principle also be a basis for further reaching tool design involving intelligent, rule-based support combining classical model checking and dynamic workflow-like guidance. In order to achieve this, indeed we would need to formalize the goals and

process rules of the strategy descriptions to get rules of the kind suggested in (van Bommel et al., 2006) and in section 6.4 of this paper ("the rules underlying a strategy frame"). We will go as far as giving semi-formal verbalizations of the rules.

Fortunately, such rules (closely related to FOL descriptions) are already partly available even in the case example: they can be derived from, or at least based on, the ORM diagrams complementing the strategy descriptions, and the YAWL diagrams that represent the strategy contexts. For example, fig. 6 corresponds to a (minimal) strategy frame as shown in figure 2. The corresponding goals is:

G1 There is at least one Domain Model

This is an instance-level goal. Next, we define the one situation that is relevant to the example strategy "create domain model based on use case":

S1 There is at least one Use Case

So we assume that one or more Use Cases have already been identified (presumably as a goal of another strategy) and that these are used as input for the strategy "create domain model based on use case". We now can weave a rule-based definition combining G1, S1, and various C-rules that correspond to the transitions captured in the strategy description. The key rules raising demands that correspond to steps in the strategy are represented in boldface.

S1 There is at least one Use Case (*situation*)
C1 SHOULD LEAD TO
G1 There is at least one Domain Model (*main goal*)
G1.1 Each Use case has concepts described in exactly one Domain model.
G1.2 Each Domain model describes concepts in exactly one Use case.
C2 SHOULD LEAD TO
G2.1 It is possible that more than one Type concept is part of the same Domain model and that more than one Domain model includes the same Type concept.
G2.2 Each Type concept, Domain model combination occurs at most once in the population of Type concept is part of Domain model.
G2.3 Each Type concept is part of some Domain model.
G2.4 Each Domain model includes some Type concept.
G2.5 Each Type concept that is part of an Interaction description of a Use case that has its concepts described by a Domain model should also be part of that Domain model (*goal underlying step 1*).ii
C3 SHOULD LEAD TO
G3.1 It is possible that more than one Domain model includes the same Fact type and that more than one Fact type is part of the same Domain model.
G3.2 Each Fact type, Domain model combination occurs at most once in the population of Domain model includes Fact type.
G3.3 Each Domain model includes some Fact type.
G3.4 Each Fact type is part of some Domain model.
G3.5 Each Fact type that is part of a Domain model should include one or more Type concepts that are part of that same Domain model (*goal underlying step 2*).
C4 SHOULD LEAD TO
G4 Each Fact type of a Domain Model is populated by one or more Facts of the Population of that Domain Model.iii (*goal underlying step 3*)

C4 SHOULD LEAD TO
G5.1 Each Scenario describes concepts of exactly one Population.
G5.2 Each Population has concepts described in some Scenario.
G5.3 It is possible that the same Population has concepts described in more than one Scenario.
G5.4 **Each Fact that is part of a Population which describes concepts of a Scenario should include at least one Instance concept that is included in that Scenario. (*goal underlying the note with step 3)***
C5 SHOULD LEAD TO
G6.1 **Each Fact type has some Constraint. (*goal underlying step 4)***
G6.2 Each Constraint is of exactly one Fact type.
G6.3 It is possible that the same Fact type has more than one Constraint.

Note that further restrictions could be imposed on G6.1, demanding explicitly that the constraints applying to a fact type should correspond to the population related to that fact type, and so on. This constraint is left out because it is also missing in the informal strategy description (step 4).

So far, our definition does not include temporal ordering. The following orderings are applied in the C-rules:

C1 *no restriction*

This reflects the achievement of the main goal, which lies outside the temporal scope of the strategy realizing it. For the rest, rather unspectacularly:

C2 occurs before C3
C3 occurs before C4
C4 occurs before C5

For a somewhat more interesting example of temporal factors, consider the XOR-split and AND-join in fig. 5. (splitting at “use case survey” and joining at “domain model”). Obviously, such split-join constructions involve ordering of transitions:

C1 occurs before C2
C1 occurs before C3
C2 occurs before C3 (under condition Y) XOR C3 occurs before C2 (under condition Z)
C2 AND C3 occur before C6

These expressions of rules covering YAWL semantics are rough indications; a technical matching with actual YAWL concepts should in fact be performed, but this is outside the current scope.
Finally, note that in the implementation, fulfillment of the main goal, “create domain model from use case”, is achieved even if the domain model is not finished. However, the unfinished status of the domain model would lead to a number of “ToDo” items. This emphasizes that the strategy is a *initial creation* strategy (bringing some item into existence), which next entails the possibility that a number of further steps have to be taken iteratively (triggered by validity and completeness checks based on, for example, G-rules), hence not necessarily in a foreseeable order.

Findings resulting from the implementation

The implementation led to the construction of a specific meta-model reflecting the key concepts used in that implementation (figure 8). We will finish this section by presenting the most interesting findings in the implementation with respect to the generic framework, at the hand of fig. 8.

Sources and products

The specific flavour of the implementation led to the introduction of the concepts *source*, *product*, *intermediate product*, *raw material*, and *void*. They were needed to operationalize the only goal/strategy category explicitly used in the implementation: creation goals. Situations (which are state descriptions) took the shape of concrete entities (documents) that typically followed each other up in a straightforward order: void or raw material input leading to products, possibly after first leading to intermediary products. Clearly, these concepts classified the items created; such classification emerged as helpful from the discussions that were part of the implementation process.

Use of YAWL concepts

YAWL concepts (and their graphical representations) were introduced to capture strategy context, while a simple textual description format was used to capture the stepwise strategy description. The YAWL concepts were very helpful in creating easy-to-read context descriptions. In addition, they helped in operationalizing the concepts required to capture the workflow-like transitions between states (i.e. between creation situations/goals). Whether YAWL diagrams would be equally useful in describing contexts for other types of goal (for example, validation goals or argumentation goals) remains to be explored.

In addition, YAWL concepts can be used as a basis for formal rule definition capturing the recommended order of steps. The formal underpinnings of YAWL would be extra helpful in case of automated (rule-based) implementation of the strategies, which was still lacking in the prototype (for more on this, see "further research": the "modelling agenda generator").

Activity descriptions, names, and steps

A simple but crucial refinement needed to operationalize the general framework was the introduction of the "activity" and "activity name" concepts. These allowed for the successful implementation of the recursive linking of *activities* to *strategy contexts* to *strategies* to *strategy steps* to further *(sub)activities*, and so on. We expect this amendment to be useful at the generic level, and henceforth we will include it in the main framework.

"Immediate" concept not used

The "immediate" concept was in principle included in the case study implementation but in the end was not used. We still believe it may be required in some strategy descriptions. The ordering in the creation strategies in the case is basic step-by-step. In more complex, dynamic setups, the availability of both immediate and non-immediate ordering still seems useful. However, admittedly the actual usefulness of the "immediate / non-immediate" distinction still awaits practical proof.

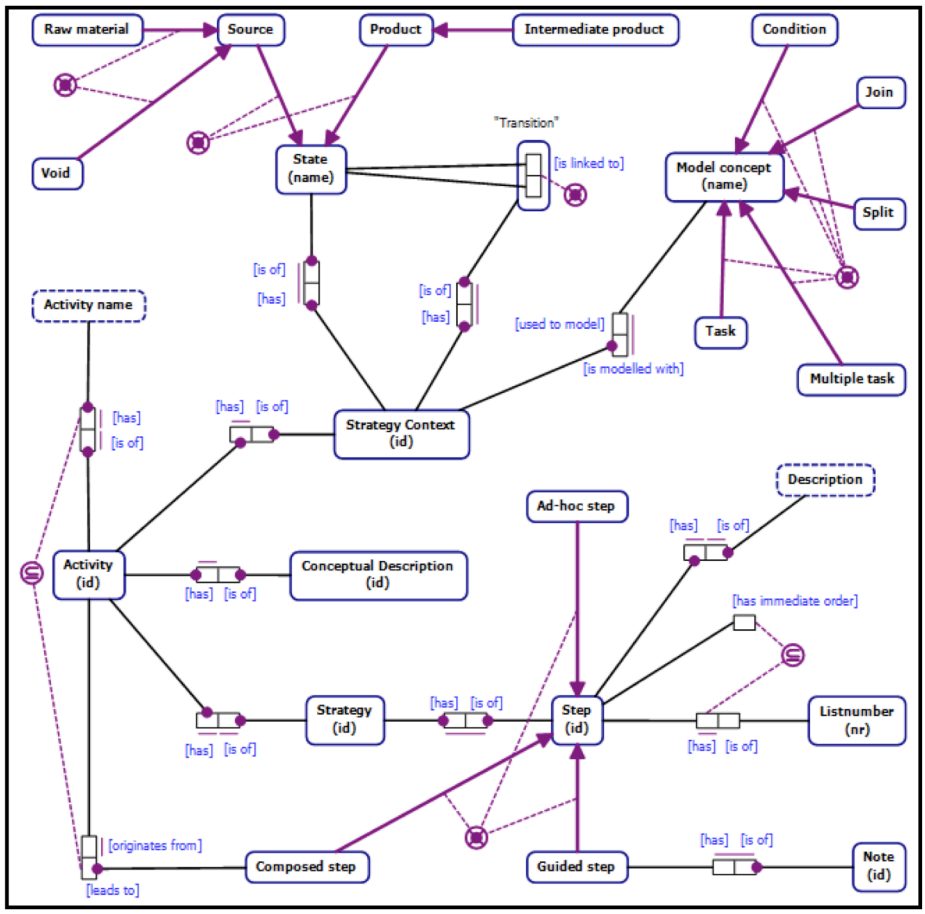


Figure 5: meta-model derived from strategy description case

Lessons learned from the case study

Apart from he conceptual findings discussed in the previous section, some other lessons were learned though the case study:

- Syntax-like rules can be successfully applied beyond actual modelling language syntax (which amounts to classic model-checking based on grammar goals) into the realm of more generic "creation goals" which may concern various sorts of artefacts within a method.
- The case has shed some light on the fundamental distinction between creation and iteration in dealing with creation goals. While iteration is essentially unpredictable and thus can only receive some ordering (if any at all) through rule-based calculations based on rules and state descriptions, for initial creation people do very much like a plain, useful stepwise description of "what to do": a reference process. Only after initial creation, the far less obvious iteration stage is entered. Also, if a robust rule-based mechanism for guiding method steps is in place, participants may choose to ignore the recommendations of the reference process. This can be compared by the workings of a navigation computer that recalculates a route if a wrong turn has been made.

CONCLUSIONS AND FURTHER RESEARCH

This chapter set out to present a plausible link between the SEQUAL approach to model *product* quality and our emerging QoMo approach to *process* quality in modeling, and to provide basic concepts and strategies to describe processes aiming for achievement of QoMo goals. We did not aim to, nor did, present a full-fledged framework for describing and analyzing modeling processes, but a basic set of concepts underlying the design of a framework for capturing and analyzing 2nd order information systems was put forward.

We started out describing the outline of the QoMo framework, based on knowledge state transitions, and a goal structure for activities-for-modeling. Such goals were then directly linked to the SEQUAL framework's main concepts for expressing aspects of model items and its various notions of quality, based on model items. This resulted in an abstract but reasonably comprehensive set of main modeling process goal types, rooted in a semiotic view of modeling. We then presented a case implementation of how such goals can be linked to a rule-based way of describing strategies for modeling, involving refinements of the framework. We added concrete examples of rules describing goals and strategies, based on the case implementation.

These process descriptions hinge on strategy descriptions. Such strategies may be used descriptively, for studying/analyzing real instances of processes, as well as prescriptively, for the guiding of modeling processes. Descriptive utility of the preliminary framework is crucial for the quality/evaluation angle on processes-for-modeling. Study and control of a process requires concrete concepts describing what happens in it, after which more abstract process analysis (efficiency, cost/benefit, levels of risk and control) may then follow. Means for such an analysis were not discussed in this paper: this most certainly amounts to future work.

Besides continuing development and operationalization of the QoMo strategy and goal framework for quality modeling by applying it to new and more complex cases, we need to push forward now to implementations that actively support our rule-based approach. An initial implementation, using Prolog and a standard SQL database, is in fact available, but has not been sufficiently tested and documented yet to report on here. This "modeling agenda generator" dynamically generates ToDo lists (with ordered ToDo items if C-rules apply) based on the model states as recorded in the repository. We will finish and expand this prototype, testing it not only in a technical sense but also its usability as a system for supporting real specification and modeling processes. In the longer term, we hope to deploy similar automated devices in CASE-tool like environments that go beyond the mere model or rule editors available today, and introduce advanced process-oriented support and guidance to modelers as required in view of their preferences, needs, experience, competencies, and goals.

REFERENCES

Van der Aalst, W. And ter Hostede, A. (2005): YAWL: Yet Another Workflow Language. *Information systems, 30*(4), 245-275.

Bommel, P. van, S.J.B.A. Hoppenbrouwers, H.A. (Erik) Proper, and Th.P. van der Weide (2006): Exploring Modeling Strategies in a Meta-modeling Context. In R. Meersman, Z. Tari, and P. Herrero, editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, volume 4278 of *Lecture Notes in*

Computer Science, pages 1128-1137, Berlin, Germany, EU, October/November 2006. Springer.

Chrissis, M.B., M. Konrad, and S. Shrum (2006): *CMMI: Guidelines for Process Integration and Product Improvement*, Second Edition. Addison-Wesley.

Halpin, T.A. (2001). *Information Modeling and Relational Databases, From Conceptual Analysis to Logical Design*. Morgan Kaufmann, San Mateo, California, USA, 2001.

---

i The ORM diagrams in this paper were produced by means of the NORMA case tool developed by Terry Halpin and his co-workers at Neumont University: http://sourceforge.net/projects/orm.

ii In expressing this complex rule, we use a controlled language called Object Role Calculus: see (Hoppenbrouwers et al., 2005c)

iii Rules G4 and G5.4 refer "Facts" and "Instance concepts", which are not included in figure 7 but in the ORM diagram (not presented in this paper) supporting a different strategy, namely "create domain model based on scenario". In the implementation, populations are defined as included in a domain model.