Contract N° IST-034824

Workpackage 2
Automatic Code Generation from Models

Deliverable 2.2
Automatic code structure and workflow generation from natural language models

Project funded by the European Community under the "Information Society Technology" Programme

Contract number: IST-034824
Project acronym: OPAALS
Title: Open Philosophies for Associative Autopoietic Digital Ecosystems

Deliverable N°: D2.2
Due date: April 2008
Delivery date: April 25, 2008

Short description:
This report discusses model transformation scenarios from SBVR to code structure and workflows. Additionally SBVR is discussed from a linguistics point of view and a Natural Language to SBVR algorithm is introduced. Finally dynamic service composition based on SBVR and a repository for business models is described.

Author: SUAS (Raimund Eder, Thomas Kurz, Thomas J. Heistracher), UCE (Antonella Filieri, Mario Russo), IITK (Prabhakar TV), UniKassel (Hagen Peukert), UNIS (Alexandros Marinos), LSE (Stan Hendryx)
Partners contributed: SUAS, UCE, IITK, UniKassel, UNIS, LSE
Made available to: OPAALS Consortium and European Commission

Versioning		
Version	**Date**	**Author, Organisation**
0.1	10/03/2008 (first submission)	SUAS, UCE, IITK, UniKassel, UNIS, LSE
0.2	13/03/2008	SUAS, UCE, IITK, UniKassel, UNIS, LSE
0.3	08/04/2008	SUAS, UCE, IITK, UniKassel, UNIS, LSE

Quality check
1st internal reviewer: Paul Malone (WIT)
2nd internal reviewer: Frauke Zeller (UniKassel)
3rd internal reviewer: na

Dependences:

Work Packages	• Through informal information exchange, outcomes of WP1 directly influence the further planning of WP 2 Tasks. • WP3 deals with the core architecture of Digital Ecosystems in Task 3.6. This deliverable aims to contribute to this architecture research. Additionally, the security models of Task 3.9 could be considered to include into further prototypical implementations. The research on transaction models and dynamic service composition relates to work described in Section B.1. • In T5.2 the outcomes of this deliverable are tested and integrated by the responsible partner (TI). • WP6, among other aspects, studies linguistic concepts for Digital Ecosystems. The linguistic perspectives provided by this deliverable can be a starting point to find applications of natural language together with computer sciences in future research (Task 6.2). Additionally, Task 6.6 deals with an Evolutionary Framework for Language, in which partner IITK will create knowledge models using the SBVR metamodel. • The outcomes of Section B.2 can influence WP10, especially Task 10.12 which deals with Knowledge Models and Representations. Task 10.15 intends to find a feasible approach for a Distributed Semantically Searchable Repository, outlined in Section B.2.

Dependences:

Partners	The partners WIT, UCE, LSE, IT, UniKassel in the above described Workpackages and Tasks could benefit from reading this deliverable.
Domains	• The computer science partners SUAS, UCE, IITK, UNIS, LSE are responsible for the major parts of this deliverable in which their outcomes are explained. • Social science partner UniKassel elaborates on the linguistic perspective on SBVR and introduces an algorithm for Natural Language to SBVR conversion algorithm.
Targets	This deliverable mainly targets researchers in the computing domain as regards the model transformation to code structure and workflow models. Additionally the linguistic parts are interesting for researchers involved in linguistic research and research on SBVR. Although the protoytpes described in thsi deliverable should help SMEs in the long term, the research is not yet studied enough to support real life scenarios.

Contents

List of Figures

List of Tables

Listings

Executive summary

This work intends to systematically interlink natural-language-based specifications with underpinning formal mechanisms that allow automatic code-structure and workflow generation. The related set of concepts, methods, and tools is addressed and discussed in the light of feasibility and linguistic considerations. *Semantics of Business Vocabulary and Business Rules (SBVR)* is discussed and used as the metamodel for transformation of natural-language based specifications. Specifications expressed as models are the cornerstone of SBVR utilisation, thus the SBeaVer editor for the creation of vocabulary and rulesets is described in brief.

Model transformation of SBVR business models is discussed alongside with six different scenarios for model transformation. Subsequently, the *code structure generation prototype*, its components and its architecture are presented, allowing web-application generation with Grails as well as XPDL workflow generation. The aspects of service composition in conjunction with SBVR usage are covered, implementation principles and related caveats are discussed as well. Furthermore, a distributed domain model community repository for Digital Ecosystems is suggested that is based on concept lattices for organizing the users' conceptual schemes.

1 Introduction

The goal of WP 2 is the presentation of a viable methodology for the creation of programs from specifications written in natural language, thus building the bridge between natural languages and structured formal languages for code structure generation. This proposal describes how formal notation can enable the automatic generation of system models starting from a natural-language based specifications. As this is a very interdisciplinary work, Section 1.2 describes the nature of the group and is intended to help clarify the connections between chapters, workpackages and partners.

Most non-computer scientists are intimidated by at even the thought of trying to write computer code. It is as unnatural as coming into a foreign land of bits and bytes and no one understands your commands, or where you want to go, or what you are trying to accomplish. You do not have time to learn the native language of computers. You only wish you had a translator from your natural language that could take your own words and convert them into detailed coded directions that these machines could understand. If this were to occur you could simply write down simple commands in your own language and create a new wash cycle for your dishwasher, create a system for paying checks, or even program the essential elements of business systems with natural language. This is somehow the vision of this workpackage.

This task is complicated by the richness of natural language. When we talk to children or even our pets we have to use a vocabulary that is understood by the receiver. The words have to have specific meaning understood by both parties and often need to be used under specific rules. A special notation called Semantics for Business Vocabulary and Business Rules (SBVR) helps take certain structured natural language sentences creates a common language for the computer. An editor is used by the person to enter definitions and rule sets, thereby helping to give precise meaning to the words.

Having a unique and understood vocabulary is not sufficient even for natural language. The words have to combine in special ways into sentences and paragraphs that make sense or have meaning. In the computer world an umbrella term to describe structured specifications, i.e. models, transformable to more specific models that computers can run is referred to as Model Driven Architecture (MDA). From abstract concepts the meaning must be translated into code that is specific enough to create all the actions and contingencies necessary to accomplish the tasks we are asking of the automated environment.

So before one gets lost in the acronyms of computer science, it is good to know that the structure that is presented is rather straightforward. We start with natural language that must be translated by SBVR into a common human/computer lexicon. These SBVR expressions are far less rich than natural language but this scale-down approach helps with subsequent steps in code structure and workflow generation. These common words are sent to programs that help to put these words structures into code that creates an equivalent meaning. There are many alternatives to developing in the MDA approach and they are discussed in this proposal as well. The final output of this process is code structure and workflows based on both the logic and an agreed upon limited natural language vocabulary.

So the proposal meets the need of the experts that know technological systems but up until now have not been able to interact directly on a computer coding level with the machine environment in which they operate and maintain. This project brings expertise from both the linguistic field and computer science field to address this important goal.

1.1 Aims of this paper – success criteria

This document proves helpful if it achieves the following objectives. The references to the certain chapters ease a focussed search for specific information.

- to sketch the collaboration work and issues regarding the interdisciplinary work connected to WP2 and SBVR (Section 1.2, Chapter 4)
- to provide a well-structured discussion of linguistic aspects for the use of natural-language based systems, specifically in the context of SBVR (Chapter 2)

- to deepen the understanding of the options and challenges in using natural-language based approaches for code structure and workflow generation (Chapter 2)
- to enable the understanding of the model-driven conceptualization (including model transformations) for stakeholders from computer science, information technology and, moreover, other non-technical disciplines (Section 3.1)
- to explore a concept lattice that organizes the conceptual schemas introduced and the important interrelation with the Meta-Object Facility and Model Driven Architecture (Section 3.1.1)
- to describe the underpinning concepts and the prototypical implementation of the "code-structure generator prototype" (Section 3.3)
- to demonstrate automatic workflow generation (Sections 3.4 and 3.5)

1.2 How-to read this document and overview on SBVR collaborations

This deliverable is not just a documentation of the work done by one or two **partners**. The six different partners and nine persons involved in writing this document have their expertise in computer science, business, natural science, linguistics and software engineering. They are living in five different countries and have their origins in even more disperse cultures. Consequently, the work done here and documented in this deliverable is much more than just the sum of the technical outcomes. We think that this diversity makes this work interesting, even beyond the technical and scientific outcomes in a real network of excellence.

The **structure of the document** at hand is similar to our way of dealing with the diversity in the work group. In Chapter 2 and Chapter 3 the labels with the names of the institution indicate that the chapters are the core research outcomes of the respective institutions. Although all partners contributed to a common goal, these sections allowed each partner to point out the specific contribution to the workpackage in the corresponding scientific or technical depth. The introduction and conclusion as well as the outlook including the reference to the joint sourceforge project SEPIAX (see [6] and [7]) are documenting the mutual efforts for the workpackage.

Beside the work on individual tasks, the following methods of **interaction** were chosen: 1) Face-to-face meetings, 2) exchange and collective work on documents and wiki pages, 3) individual chats between the partners, and 4) the periodical SBVR chat. As regards communication, two concepts proved very useful. First, the setup of a rather small group, defining terms and a common understanding and way of interaction and then, successively, growing the group. The main reason for that is that it proved to be easier to exchange ideas and heterogeneous vocabulary in small groups than starting with broad discussions in a large group from the beginning on. Second, the almost weekly SBVR chat was a good communication approach, because meeting minutes were documented at each session and for more information, the chat-logs are available online for the whole community. Until April 2008, 19 such chats with the title *Languages and Models* have been held and documented in the OPAALS wiki [8].

Additional, to the work in WP2, **other workpackages** are related to the work done so far and partners also participated partially at the Language and Models chats. Especially, WP6 with the focus on Natural Language and evolution of language had an interest in the outcomes of WP2. As UniKassel is involved in WP2 and WP6, there was a good exchange of ideas and parts of the outcomes are documented in Chapter 2. Unfortunately, the interaction with WP1 was not as strong as planned for the first phase of OPAALS. The lack of resources in WP2 required a replanning and it came out that a strong collaboration at this stage would have been premature. Nevertheless, for phase II we can think of additional collaborations with WP10 - Sustainable Research Community Building and the Open Knowledge Space.

As mentioned above, the **SEPIAX project** started to be the collaborative space for all SBVR and WP2 related activities. All sourcecode, documented in this deliverable is uploaded there as separate sub-projects. Parts of them, like the *SBVR2UMLActivity* service and the *SBVR2Grails* service are already working with the *SBEAVER* service (see [9]), which is the SBVR editor of choice. Parts of the integration are still work in progress and will be tasks in phase II of OPAALS. The documentation of the outcomes is already started as a `http://Typo3.org` based webpage at `http://sepiax.org/`. Figure 1.1 shows a screenshot of this page.

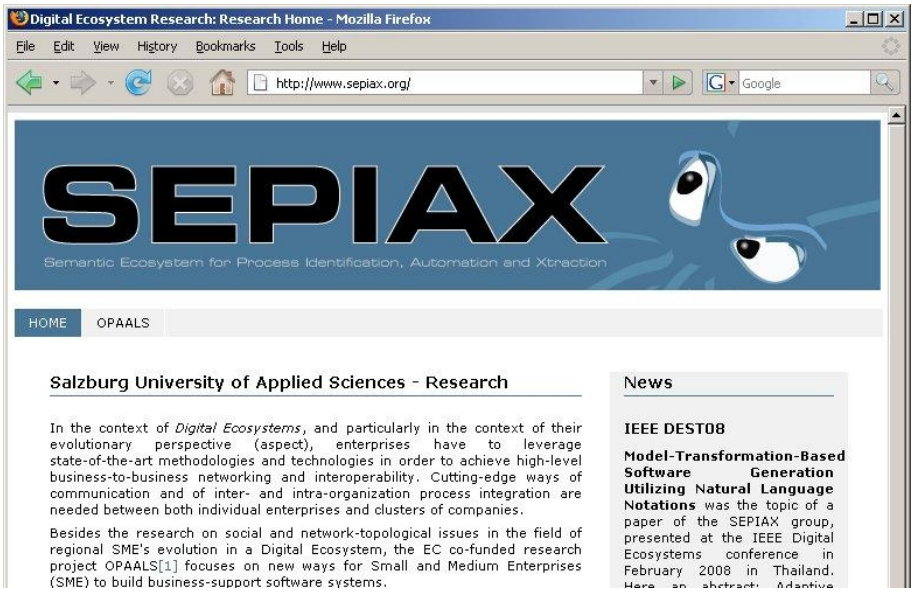


Figure 1.1: SEPIAX webpage.

1.3 Overview of the Report

After a short introduction of natural language processing and its current limitations, Chapter 2 provides a discussion of the concepts needed for natural language processing. SBVR as the metamodel for specifications and SBVR Structured English as the representation format of these specifications is explained and a transformation algorithm is introduced which allows to convert natural language specifications into SBVR specifications. There is no explicit introduction to SBVR because SBVR was already introduced in earlier deliverables of this workpackage and there are of course references to the full specifications as well as other SBVR material.

In the following, Chapter 3 explains the different methods of model transformation studied in the context of WP 2. Beside a short introduction to the MDA approach as an underlying concept for software development, Section 3.2 outlines several scenarios for automatic code structure and workflow generation coming out of natural language based models. The three methods described here in more detail are (a) SBVR to Grails Application, (b) SBVR to XPDL Workflow and (c) SBVR to UML-Activity Diagram. Beside the conclusion and outlook Chapter 4, the Appendixes A, B and C sum up additional material related to the WP2 and SBVR activities as well as some sourcecode which acts as the basis for further development.

2 Linguistic Perspective on SBVR [UniKassel]

After a short analysis of natural language and its current limits, Section 2.2 aims at investigating the SBVR model in view of its attractiveness to natural language. The SBVR meta-model claims to be restricted to semantics leaving apart the key functionality of human languages, which is syntax. Foremost, we have to concentrate on the question as to whether SBVR can handle natural language without recourse to syntactic information. Indeed, this objective is crucial because it involves a critical analysis of SBVR as well as some kind of evaluation.

The third Section 2.3 is reserved for some suggestions for improvements and a proposal for how SBVR can be integrated in a larger framework of natural language processing containing a preliminary syntactic analysis. Finally, we will outline a proposal for the first prototype.

2.1 Natural Language Processing and its Current Limits

Dealing with information based on processing of natural language carries two main risks. First, there is still no natural language that could be theoretically described in its entirety up to the present day. The richness of natural language has not yet allowed finding a complete account of its very nature. This makes it extremely difficult to propose appropriate algorithms that cover most of the complexities of natural language structures. Second, there seem to be different conceptions of what natural language is, so that, researchers in different domains approach the problem differently or identify different core issues about natural language. Therefore, language processes will be modelled at incompatible levels of abstraction. To avoid

the latter, we can simply state what natural language is or what we agree upon that it should be, and which problems might occur when modelling it (Section 2.1). The former cannot be easily circumvented. The only option to diminish this risk is to start small and work our way up tackling each upcoming problem sequentially (Section 2.2).

As pointed out above, this first Section aims at being clear on the term 'natural language'. Having done this, we will look at typical natural language systems and its components. Then, we will see how the core elements will interact with each other and whether we can afford to ignore some of them.

Each textbook offers a definition of language that might differ slightly. However, they are all very clear on one point: natural language is the language used by humans with all its aspects.[1] Definitions of 'natural language' as opposed to the general concept of 'language' are used to mark off artificial languages, pidgins or animal talk and there we do not encounter the problem of diverse conceptions. To sum up, the common denominator of all definitions is: "Language is thought of as the uniquely human part of a broader system of communication that shares features with other animal communication systems" ([10]). It is important to keep in mind that we understand natural language as the phenomenon that it is. Defining a simplified version of natural language cannot be called natural language since, by definition, the boundaries to artificial and animal languages become blurred. To be precise, how then will natural language differ from animal communication systems or even quite elaborated forms of pidgins? So, within our research community, we agree with the conventional linguistic use of natural language. As such it has the following properties:

- Languages are used by all human beings.
- Languages cannot be "primitive". They are equally complex and capable of expressing any idea. Vocabularies can be extended to new words and concepts.
- Languages change throughout time.
- Meaning and sound pattern are mostly arbitrary.[2]

[1] The two keywords here are "human" and "all". "all" refers to the entirety of all subsystems of natural language: semantic, syntactic, morphological, etc. processing; and "human" refers to our species.

[2] Except onomatopoeic words.

- The grammar of languages can be described in terms of rules (but not completely).
- All languages contain a set of (possibly discrete) units that combine to meaningful elements (possibly infinite) which may be further combined to complex meanings in sentences (definitely infinite).
- Languages have syntactic categories as nouns and verbs and semantic properties such as male/female.
- Languages have the capacity to negate, form questions or commands, referring to the past, present and future.

As should be clear from this small subset of properties defining natural language, it is a challenge to process all the information given. Simply put, it is impossible at the current time. Consequently, the question is, how can we constrain a model of natural language with regard to the SBVR meta-model? A good starting point on natural language processing systems and their basic requirements is to look at some of the general architectures in this field claiming to be efficient and successful in recognizing human language. Here, dialog systems seem to be adequate because they involve all the main challenges that researchers of Natural Language Processing (NLP) try to solve.[3] Figure 2.1 reveals the typical design of a dialog system.

Indeed Figure 2.1 gives just a vague idea, but can spare the details for the purpose of this section. The upper stream introduces the components of natural language recognition systems matching below with its counterparts of language generation or production systems. For reasons of simplicity, the modules of speech recognition and synthesis can be neglected. By the same token, the dialog management is only slightly tangent to our concern because it provides for contextual analysis, pragmatic information and algorithms that align the input representations with relational databases or some alternative form of knowledge base. This leaves us with the syntactic and semantic modules as input or output respectively. Again, we concentrate on the input of natural language, that is, transferring natural language input into some machine-readable representation. For the moment, it will not concern us whether the input adds or exploits (queries) knowledge.

[3]Machine Translation left aside because it is still subject of the ongoing debate whether syntactic and semantic mapping of two different languages should be put into the dialog management.

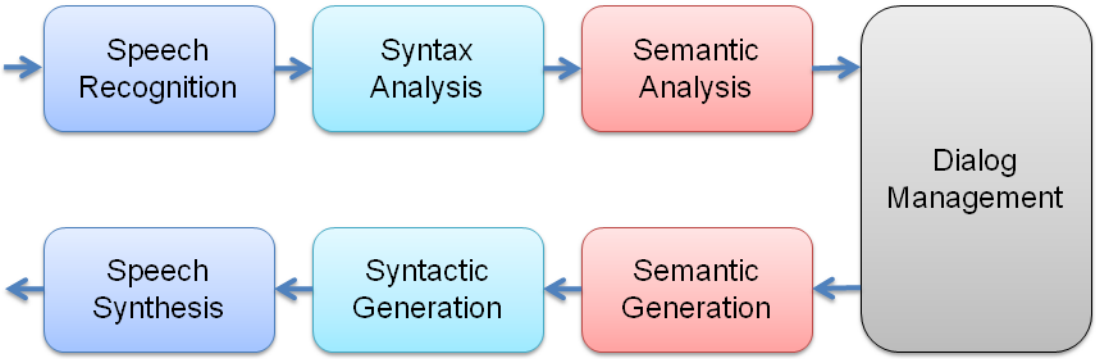


Figure 2.1: Typical architectures of a dialog system, Source: [1]

2.2 Linguistic Analysis of SBVR

Roughly, the SBVR editor is supposed to fulfil the same task as the syntactic and semantic modules in dialog systems (see Figure 2.1, Section 2.1). It is definitely worth questioning the overwhelming harmony and agreement of the linguistic community in this respect. There is hardly any successful NLP - System reported in the literature that would disregard one of the core elements: syntax or semantics.[4] In this logic, language is often seen as a lexicon and a combinatorial apparatus ([11]). With regard to the discussion, we have to ask whether other alternatives are worth exploring or whether we can exclude this idea from the beginning, which means following suit with mainstream computational linguistics. Clearly, the principle of parsimony applies and therefore all components should be left out that do not contribute to the objective of finding the correct relation between input and stored representation. In short, if a syntactical analysis does not add essential information, we can discard it.

In spite of taking a functional stand, it might be instructive not to disregard the human processing system entirely. In fact, the sequence of the components of dialog systems is not arbitrary. [5] Syntactical analysis precedes semantics. This is also true for humans. ELAN (early left anterior negativity) gives evidence for a first preliminary syntactic analysis after a few milliseconds ([14], [15]). Obviously before any meaning can be inferred, some kind of syntactic representation is required

[4] There is one that disregards both: ELIZA, but the success of the program is due to different reasons; likewise its objective.

[5] [3, 247], [12], [13]

to set the stage for semantic processing. Hence, for humans we can state that meaning needs a structure. Thus, structure carries meaning. And in linguistics, this structure is called syntax.[6] The question now narrows down to: is the information that syntax provides for meaning necessary for a machine? In other words, will a strategy of placeholders and analysis of the sequence of tokens relating to logical representations suffice not to lose important information?[7] To answer the question, we will examine some syntactic properties of natural languages that go beyond word order: structural ambiguity, transitivity of verbs and nouns, passive constructions, and relative clauses. Each of these syntactic peculiarities will be introduced with the example shown in Figure 2.2.[8]

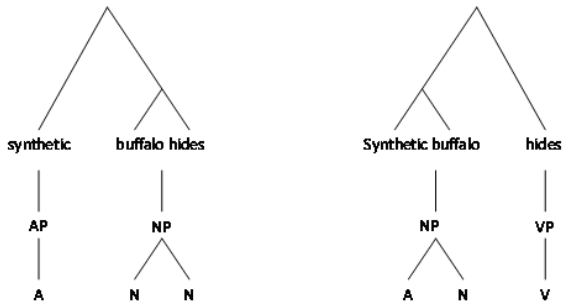


Figure 2.2: Examples adapted from [2].

Starting with structural ambiguity, we investigate the sentence or fragment:

a) synthetic buffalo hides.

Again, this example is taken as an illustration and therefore kept simple. One might not see the relevance to an NLP - system right away. It is no problem, whatsoever, to make it more realistic. Probably, a short glance into today's newspaper will provide you with more instances:

b) I would like to sell an antique desk suitable for lady with thick legs and large drawers[9]

[6] As a simple example: John kills Mary is different from Mary kills John.

[7] The sequence of tokens really is part of the syntax (word order). Since SBVR also includes this structure, we will leave it here vaguely defined as part of semantics as well.

[8] Examples adapted from [2, 122]

[9] Putting this sentence in a knowledge base for SMEs using a NLP system without syntactic analysis may surely harm the business since the logical representation might define the target group as ladies with thick legs.

c) We will oil your sewing machine and adjust tension in your home for ...

d) The boy saw the man with the telescope.

e) Bill wants the presidency more than Hillary.

They all have in common that the two[10] meanings cannot be understood using semantic knowledge. Here, only the structural analysis will help. This is also true for any algorithmic translation into machine-readable style. Neither the order of the words nor the semantic content of each part gives the clues necessary. On the other hand, syntactic rules of English combined with statistical information could solve the ambiguity problem quite reliably. In the above example it would possibly suffice to match the tagged pattern with Standard English sentence patterns or rules respectively without relying on more elaborated probabilistic algorithms.

As opposed to structural ambiguity, the sentence

f) This will make you smart

also reveals two meanings. The difference, however, is that the ambiguity does not originate in the hierarchical organization of the sentence, but merely in one word "smart" ("burning sensation" vs. "clever"). For this kind of ambiguity efficient semantic formalism do exist and the ambiguity can be solved using semantics only. The reason why it works here and not in the examples a) through e) is the structural representation. In f) the phrase structure tree will be equal for both meaning representations.

Another syntactic issue that can be raised is transitivity. Transitive verbs take one or more complements (direct objects). While a sentence like

g) The girl found the box.

aligns with one possible SBVR structure and

h) I sleep.

accords yet with another latent structure, there is no structure provided for

i) The boy puts the milk in the fridge.

[10] One could argue that the last example even has three possible interpretations.

The verb 'find' is transitive because a direct object is necessary for the sentence to be grammatical.

j) *He finds

is not a correct expression, alone. 'sleep' is intransitive. It does not take a complement and can stand by itself. So far we do not have a problem. 'put', however, requires two complements.

k) *They put the milk.

l) *They put in the fridge.

are not grammatical sentences of English. The information necessarily comprises the 'what' and 'where'. 'feel' even needs an entire sentence or at least an adjective phrase. 'tell' cannot stand without both a noun phrase and a sentence. Nouns as 'sympathy' or 'belief' may take complements or not. All these very common constructions cannot be processed correctly using the SBVR editor in its present configuration. Still, these are not insurmountable obstacles as of yet.

Once more we like to ask if semantics alone could solve this without syntax. The information of transitivity can be stored together with the concept as an entry in a logical representation. This can also be added to the SBVR algorithm. Problems will occur if words like 'think' or the nouns mentioned show up. Here complements may be required or not. There is no mechanism known to us which enables a system based on semantics to decide which representation to choose (with or without the complements). The reason is clear-cut. A semantic system has got no representation of the subsequent structure of the sentence. Specifically it cannot decide whether the phrase following is a complement or some independent phrase. In terms of a possibly updated SBVR editor, the placeholder algorithm cannot decide whether to match the input with the second concept or with the extension of the lexical entry.

The third argument raised here are passive constructions. As pointed out earlier, word order really is part of syntax, but we accept it here as semantics. Word order by itself is not a very reliable source for relating meaning to logical representations. The examples m) to o) illustrate the idea

m) John kills Mary

n) Mary kills John

o) John was killed by Mary

Relying on the order of words, one could define the first word as the agent (the one who performs an action) and the last word as the theme (the one who undergoes an action). This is true for active constructions. The passive interchanges agent and theme, as sentence o) shows. Hence, a semantic system needs some kind of algorithm to distinguish active and passive sentences, which gets us right into the domain of syntax. Even if the program checks for the verb, it still is not easy using the semantic knowledge. Consider the example given in p).

p) John was walking by the store

Now we touch the area of morpho-syntax and think in terms of grammatical morphemes (-ed and -ing) to solve the problem. A semantic representation should store each token (walk, walked, walking) separately for each entry. A syntactic representation would store a production rule for all entries and the single instance as an entry. From this perspective a syntactic module is more parsimonious and should be preferred. To make the case clear, relative clauses are taken into consideration.

q) The car sold to the man passed by the truck.

A placeholder strategy will run into difficulties with relative clauses in general. To grasp the correct meaning one needs the structural information of the sentence. In this particular example, mapping onto placeholders can easily be confused with passive construction or progressive forms as in p). The word order is not at all a reliable cue because the length of the subordinate clause is arbitrary. Furthermore, the subclause may contain the same placeholders as the main clause (article/quantifier, concept, verb in SBVR). Going back to the initial question, i.e. should we take conventional NLP models as a basis for further research, we may answer it in the light of the short survey given. Provided that we talk about a system that aims at processing natural language, syntactical analysis is inevitable. This is well founded in the literature. As an example out of many, we like to cite [3]. They remark that

> ...the creation of rich and accurate meaning representations necessarily involves a wide range of knowledge-sources and inference techniques. Among the sources of knowledge that are typically used are the meanings of words, the meanings associated with grammatical structures, knowledge about the structure of the discourse, knowledge about the context

in which the discourse is occurring, and common-sense knowledge about the topic at hand.

From here we should be able to make a fair judgment to which degree SBVR is capable of handling natural language input. The creative aspect of language spans the entire inventory of natural language: phonology, morphology, semantics, pragmatics, discourse, and syntax. A language without syntax cannot be natural. Syntax is especially important since it enables us to be parsimonious. A finite set of rules allows for the expressing and understanding of an infinite set of ideas. Since the world we live in requires us to express uncountable ideas, we need the syntactic mechanisms. Restricting us to semantics would require a separate symbol for each new idea. As a consequence, to achieve the same result, that is, expressing/understanding an infinite set of ideas, would lead to an unlimited set of semantic symbols or concepts. How would a machine handle that? As a consequence, SBVR, in its present release, should be expanded with syntactic information processing devices. Its semantic structures are represented on predicate logic statements. These could possibly be used once the back-end to subsequent business modelling processes needs the SBVR output as its input. As of now, one could also keep the main structure of the meta-model of rules and fact types.

2.3 Specification of an NL-SBVR Transformation Algorithm

Figure 2.3 depicts a possible solution to the problem. First a syntactical analysis will parse the input to an unambiguous phrase structure from where the information can be carried over to the SBVR - system if the structure accords to the SBVR standards. Otherwise conventional semantic analyzers will transfer it to a semantic representation. From there, logical operators have to be added.

Of course, we cannot solve all of the peculiarities of natural languages described above (Section 2.2). As stated in chapter 2.1, we have to start small and work our way up. Consequently, we are inclined to focus on the most important issues and offer a solution that is, on the one hand, feasible with respect to time and resource constraints and, on the other hand, most suitable to the needs of the usage of a natural language-based processing system. So we do not claim to give a perfect natural language front-end taking into consideration the most sophisticated and recent

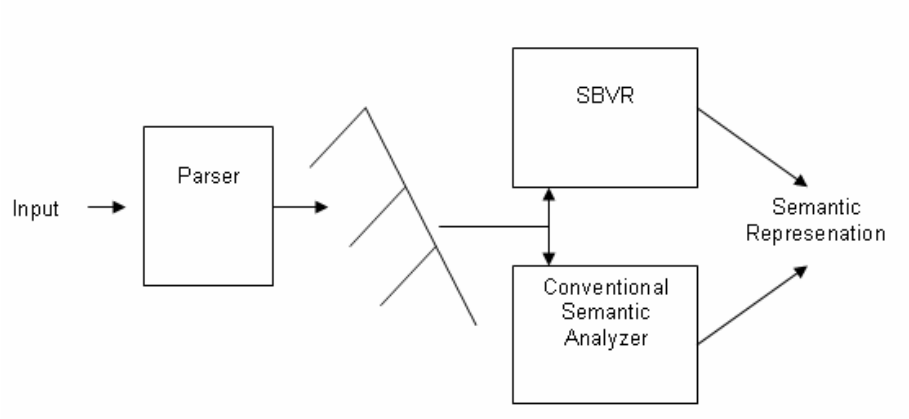


Figure 2.3: Suggestion for improvement of SBVR (adapted from [3, 547]).

technologies in the field, but we do believe that we can propose the best solution for the given problem, that is, mapping complete natural sentences to the very particular structure of SBVR fact types. It is not the purpose of the present investigation to include other issues arising when integrating the SBVR meta-model into natural language environments. As a first step from many, we restrict ourselves to extracting the vocabularies from complete natural language sentences. Moreover, embedded sentences as subordinate clauses, some passive constructions and transitivity will not concern us at this initial stage. For reasons of simplicity we will also leave out verb phrases that act as noun phrases as in

r) Keeping up with international competitive forces is one of the main problems of SMEs.

and stick to the more likely examples. So, the burning question is how to extract the very specific SBVR fact types from a natural language sentence. First, we have to choose an appropriate parsing algorithm. The parser should meet some criteria. Above all, the program should detect a wide variety of syntactic constructions since this area of application is large and should cover all of the peculiarities described above. This would suggest rule-based parsing technologies. The second criterion, however, could be its adoption in web based infrastructures. This runs counter to the huge memory taken up by rule based systems and rather points to stochastic algorithms. Having carefully investigated the performance of some dozens of representatives from both technologies, eventually we came to the conclusion that the

probabilistic parser would be the best choice. Here the Stanford Parser by [16] showed overwhelmingly more favourable results for the tested material.

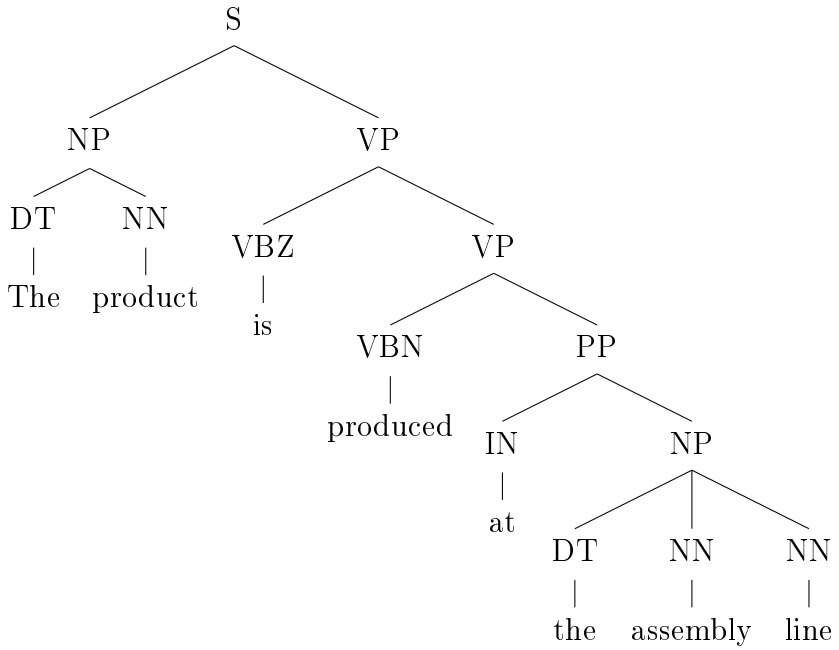
The parser outputs the structural interdependencies within an English sentence. The main idea of the prototype algorithm is to use the syntactic structures that, by definition, are always true for every English sentence. From there, the algorithm checks the most likely scenarios. Regarding the first condition, each English sentence consists of a noun- and verb-phrase. The order of both phrases is fixed.[11]

`S → NP VP or S((NP)(VP))`

To illustrate, the sentence:

s) The product is produced at the assembly line.

has got the following syntactic structure in tree notation:

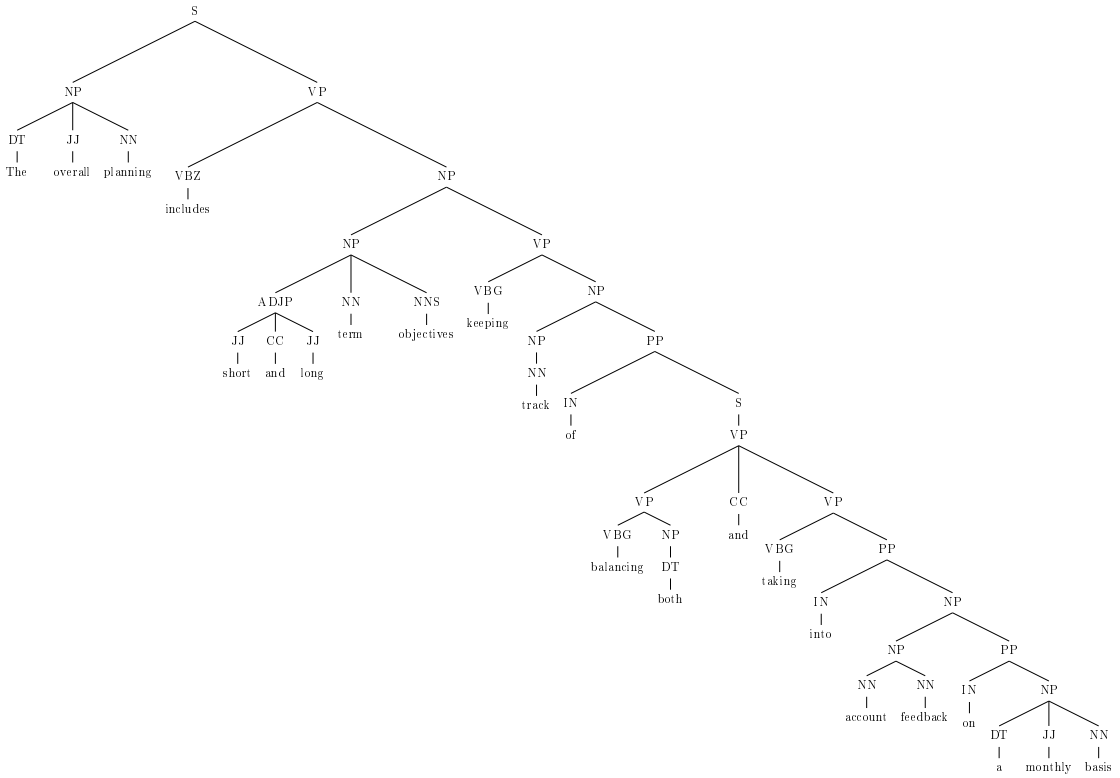


It is easy to see that the initial NP und VP knots dominate all other constituents. Still, a more realistic example reveals that several NPs and VPs can be nested within each other.

t) The overall planning includes short and long term objectives keeping track of balancing both and taking into account feedback on a monthly basis.

[11] Subordinate clauses follow the same pattern even though there are a few exceptions. Here the first NP is the part of the main sentence that it relates to. For the first prototype we will not elaborate on these specific structures.

In tree notation:



Here the problem of complexity becomes somewhat more apparent. From the above tree diagrams, one can derive some more general patterns of syntactic structures. Interlaced NPs and VPs contain at some level below the subject, the main verb and possibly an object. It is only this information that we need for the SBVR-syntax, but as one can see from the sample trees, it is far from trivial. Looking at the given tree above, it is clear that the algorithm cannot simply search for an NNx or VBx. There are some seven instances in the case of NNs. The structural path will solve most of the riddle. However, only the most common tracks amount to some hundreds.

As a representative illustration of how to handle the structure, we like to pick out the Participle and Gerund constructions. SBVR neglects complex syntactic information and narrows sentences as given in u) through x) down to [car] [is produced] [assembly line].

u) The cars have been produced in an assembly line.

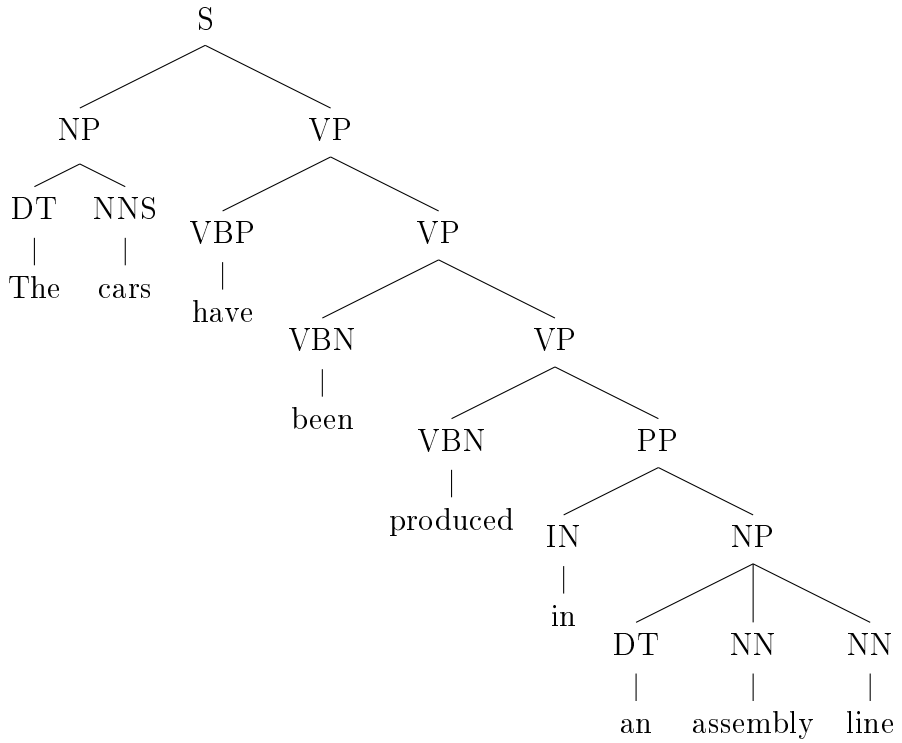
v) The car has been produced in an assembly line.

w) The car is produced in an assembly.

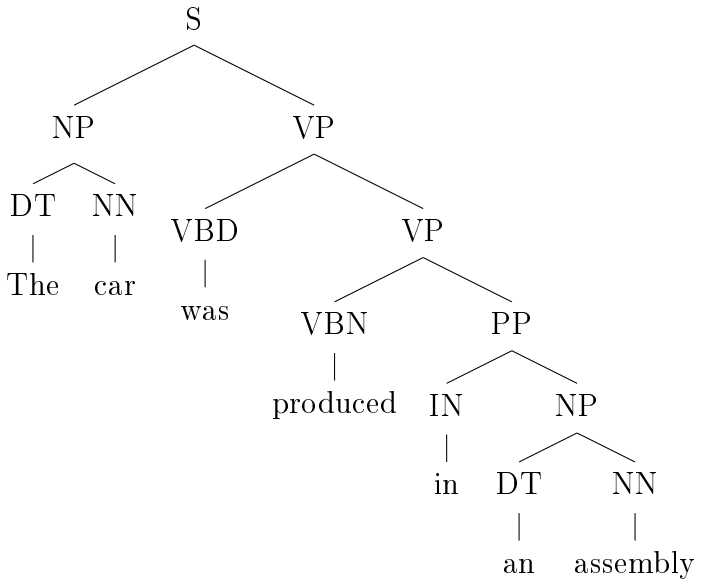
x) The car was produced in an assembly.

etc

The syntactic information from sentence u) is given below



This structure is in terms of the SBVR requirements somewhat close to that of x):



The algorithm in pseudo-code considers most of the variants of this kind of participle constructions:

```
If (first VP has VBP or VBZ and VP and the last VP has VBN and VP)
Do
output ‘‘is’’ + VBN or VBD of the last VP
Else
Do nothing
```

Implementing this in a Java Program then is easily done as a routine since it is somewhat like a translation exercise. The source code for the structure above is included in Appendix C.1.

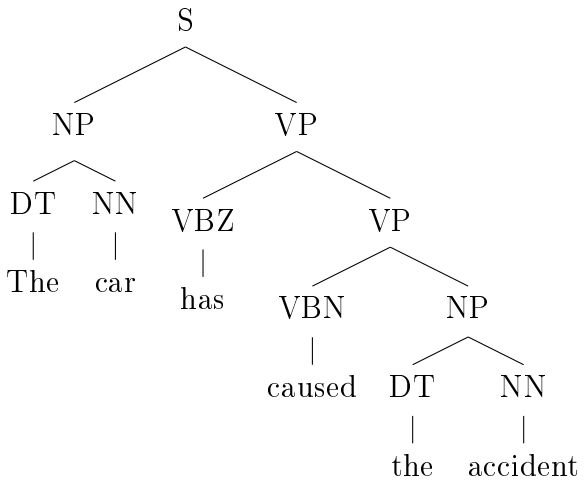
Equally, the following statements can be transferred by the same manner.

```
If (first VP has VBP or VBZ and VP and this VP has no VBN and VP)
Do
output ‘‘is’’ + VBN or VBD of the last VP
Else
Do nothing
```

This statement covers a sentence containing a simple Past Perfect construction as in y)

y) The car has/had caused the accident.

It should output [car] [caused] [accident]. Below is a description for an algorithm that explains how to transform the Simple Past markers to 3rd Person Singular. Even though the syntax tree looks very much like its passive pendants above, one should not combine the two since it may not find the correct path.



Likewise, different constructions using the Gerund can be processed.

z) The car is producing a lot of smoke.

is supposed to be simplified to [car] [produces] [smoke] in SBVR notation. The following statement takes care of such constructions.

```
If (first VP has VBZ or VBD and VP and this VP has VBG)
Do
substitute ‘‘ing’’ of VBG of the last VP by ‘‘(e)s’’
Output the manipulated VBG
Else
Do nothing
```

Thus, the first move is made to break the ground for further inquiry into fine-tuning the existing structure and taking up new challenges as subordinate clauses, the remaining passive constructions[12] or transitivity related issues[13]. Fine-tuning can be done for turning plural constructions into the singular form that SBVR needs. The rule of the English Plural can be described in linguistic terms quite reliably: Add the s-Morphem to the word unless it ends with a sibilant (then add [əz]) or it is an exception. However, this definition presupposes phonological encodings that are non-existent in writing and exceptions make a definition extremely vague. Exceptions, to begin with, do occur in English more often than computational linguistics would like although we are not aware of them. Still, they do not come in such numbers that they could not be enlisted and checked. They comprise plural words as sheep or milk, words adopted from Greek or Latin as phenomenon or criterion or words that cannot be squeezed in any pattern as child, ox or mouse. More problematic is a non-phonological definition of sibilants. Phonologically, English sibilants are all voiced and voiceless alveolar and palatal fricatives and affricatives (/s/, /z/, /tʃ/, /dʒ/, /ʃ/, /ʒ/). The problem is that there does not exist any one-to-one relation to the graphemic transcription. For example, badge or garage end with “ge” in writing, but with a sibilant phonologically match ends in “ch”, bush in “sh”, buzz in “z” and bus in “s”. They all take the [əz] - Allomorph. From the short survey given, we can derive a simplified rule for the graphemic description of the English plural, that is, considering the last two letters for defining the plural allomorph.

[12] Those in which agent and theme have to be exchanged as in “the car was stolen by the thief” in SBVR: thief steals car

[13] The latter cannot be handled by SBVR itself because it does not take more than one object. Hence this is left to some related semantic analyzing structure.

Finally, we will add also cases ending in "x" even though the rule does not apply to all of them. Nouns ending in "y" are, again, problematic when considering words like fly and boy. The "y" changes to "ie" when preceded by a change in consonant clusters. Luckily, the graphemic transcription in English has got advantages. One can neglect the voicing of the final consonant. For all consonants but the "s" is added to the end of the noun. Now the rule can be stated as follows[14]:

1. Add "s" to the end of the noun when it does not occur in the list of exceptions
2. Insert a "e" before the plural morpheme "s" when the NNx ends in "s", "z", "x" "sh" or "ch"
3. Insert a "ie" before the plural morpheme "s" when the NNx ends in consonant + "y"

To "depluralize" a graphemic word, the algorithm could work when reversing it. So we may say

1. If the noun is not in the list of exceptions, delete the final "s".
2. If a "e" is the last letter, delete it if it is preceded by "s", "z", "x", "sh" or "ch"
3. If consonant + "ie" are the last letters, substitute them for "y"

One has to be careful not to simply delete the final "e" without checking its predecessors because they are a whole range of nouns (lathe, love, etc.) that would not be correctly written.

Another positive aspect about English is that the plural rule can be applied to turning an infinitive verb to its third person singular representative, which also needs some fine-tuning in the existing version of the algorithm. There are indeed again exceptions that have to be handled with lists. The best known example are the forms of be, but also go counts among an exception in the existing algorithm.

Remaining with verbs, some more improvements can be done for dealing with irregular past and participle forms. Except for some neural network implementations, there is no other choice of handling the irregular forms as to save all of them in a list. The positive side of the alphabetic transcription of English texts is an easy handling of the regular plural. Again, the crucial aspect of voicing can be neglected as well as verbs that end in /t/ or /d/ respectively and the rule is straightforwardly applicable to all regular verbs:

[14] See [3, 67] for two finite state automata for English nominal and verbal inflection.

1. If the verb is not in the list of irregulars and it does not end in “e”, add “ed”, otherwise add “d”.

Last, we like to mention a general problem. As we have already pointed out, each parsing technology has its shortcomings. A shortcoming of all parsers so far is to group different words to one syntactic category when they also could be perceived as different categories as in “go on”, which is really a verb; however, it is represented as a verb phrase consisting of a verb and a prepositional phrase. Most of the time, the prepositional phrase is assigned to a subsequent structure. Once the parser outputs the wrong structural relations, the algorithm will produce incorrect results as well. In the present case, it would only display one part of the verb, go, leaving the second part, on, out.

3 Code Structure and Workflow Generation

For code structure and workflow generation out of SBVR models, six different scenarios of model transformation[1] are considered, from which three different approaches were prototypically implemented to demonstrate the applicability of these approaches within the Digital Ecosystem context. Before this chapter elaborates on these scenarios, a short introduction on the underlying MDA approach is given. In the following, this chapter introduces the six scenarios first and then provides the technical details of the related work performed in WP2. Figures 3.1 and 3.2 visualize these six different scenarios of code structure and workflow generation which will be described in the following sections. The first two scenarios (1 and 2) were considered as optional and were not implemented as part of this deliverable. The work related to scenario 3 was performed by IITK. SUAS adressed scenario 4, while a subcontractor of UCE dealt with scenario 5. Scenario 6 was only discussed in brief.

3.1 Software development: the MDA approach [UCE]

Methodologies for software development (e.g. RUP, Agile) typically start from the formulation and description of the problem to be solved through some sort of formal and technical language (typically UML) for requirements gathering. Starting from this point, software developers transform requirements into code with a relatively routine process.

Nevertheless, the actual difficulty lies in the previous step, that is, describing problems and expected functionalities. Stakeholders, in particular domain experts or customers, involved in software development usually express their ideas and re-

[1] for an introduction about model transformation see [17]

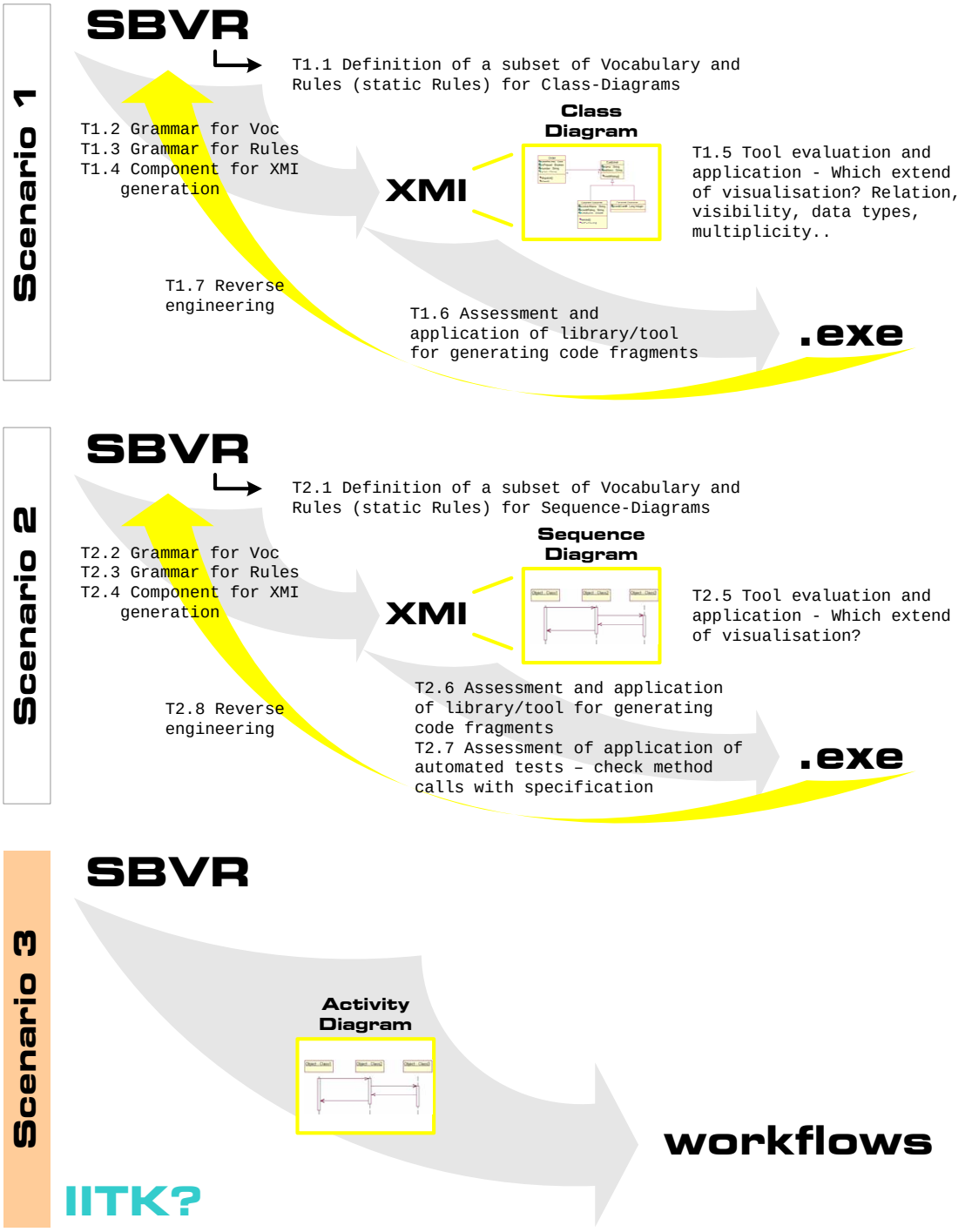


Figure 3.1: Scenarios 1-3 for SBVR Code and Workflow Generation

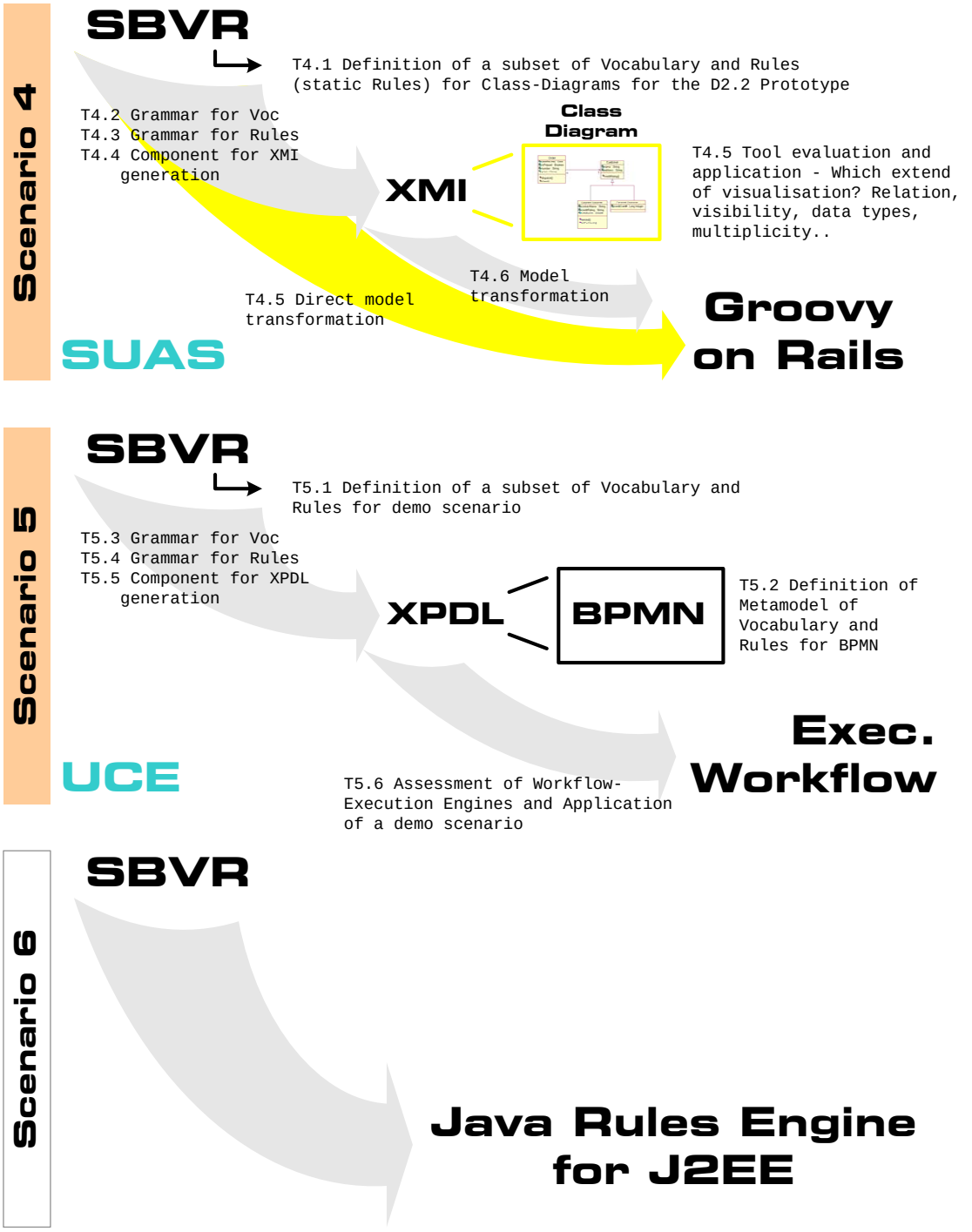


Figure 3.2: Scenarios 4-6 for SBVR Code and Workflow Generation

quirements using a language very close to them (natural language), but not formal enough to be interpreted in a clear and unambiguous way. Indeed natural language, though providing great communicative power and expressiveness, is rich in complex constructs and semantic ambiguities, as discussed in the previous chapter, thus lacking in formalism.

As a consequence, a very important role is played by requirements analysts that act as a sort of translator between stakeholders and software developers. Obviously, this implies a large effort in order to interpret and understand real meanings and concepts hidden among stakeholders' words since rarely they share concepts and meanings. Conversely, as envisioned by MDA (see next section), requirements gathering through the creation of machine-readable documents needs precision and formality (i.e. to be compliant with a meta-model or a grammar), with a consequent loss in richness of meaning and expressions. Bridging the existing gap between business people language and formal languages used for software development, represents a fundamental issue for an effective and complete software life-cycle management. In this respect, the MDA approach could be a very interesting choice for balancing these opposite needs. Indeed, this approach can provide a powerful means to use very high-level language in order to create consistent, unambiguous and formal business models.

3.1.1 The Model Driven Architecture (MDA)

The Model Driven Architecture (MDA) is an evolution of CORBA/OMA that addresses integration and interoperability of software systems during their entire life-cycle, from requirements gathering and system design, through implementation and deployment, to maintenance and evolution. Such phases and the workflow between them heavily affects the degree of reusability, interoperability, portability and maintainability of the target software system. The Model Driven Architecture (MDA) aims to be the answer for these problems, by defining *an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. To this end, the MDA defines an architecture for models that provides a set of guidelines for structuring specifications expressed as models* [18]. The MDA defines a modelling stack that starts from high-level but machine-readable descriptions and, through level-to-level transformations is able to generate low-level models (executable code). Another very interesting characteristic of the MDA approach is the retroactive link

from low-level models up to the more general ones, so that any change in the code can be directly and automatically reported in a variation of the correspondent documentation. The following is a brief description of the MDA modelling stack:

CIM (Computation Independent Model): A computation independent model is a view of a system that does not show details of the structure of systems. A CIM is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification. It is assumed that the primary user of the CIM, the domain practitioner, is not knowledgeable about the models or artifacts used to realize the functionality for which the requirements are articulated in the CIM. The CIM plays an important role in bridging the gap between those that are experts about the domain and its requirements on the one hand, and those that are experts of the design and construction of the artifacts that together satisfy the domain requirements, on the other [18].

PIM (Platform Independent Model): A platform independent model is a view of a system that exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type. A very common technique for achieving platform independence is to target a system model for a technology-neutral virtual machine. A virtual machine is defined as a set of parts and services (communications, scheduling, naming, etc.), which are defined independently of any specific platform and which are realized in platform-specific ways on different platforms. A virtual machine is a platform, and such a model is specific to that platform. But that model is platform independent with respect to the class of different platforms on which that virtual machine has been implemented. This is because such models are unaffected by the underlying platform and, hence, fully conform to the criterion of platform independence [18].

PSM (Platform Specific Model): A platform specific model is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform [18].

As stated above, MDA allows automatic transformations between models of different levels. Transformation stands for automatic generation of an objective model

starting from a source model of the same system. In order to define a transformation, a set of transformation rules is necessary; the rules describe how a model formalized in a given language can be transformed in another model formalized in another language. First of all a transformation has to preserve the original meaning of the model and the rules must be unambiguous in order to allow a consistent transformation.

3.1.2 The MDA approach for software development process

A software development process that leverages the MDA approach is based on the distinction and separation between system functionality, structure and behaviour, and their implementation. The system is described by different formal models that have different levels of abstraction and that must be specified by different stakeholders. The requirements gathering phase produces a high-level documentation (CIM) that is represented by a language near to the one domain experts use in their everyday life. The subsequent analysis phase starts from the CIM and produces a PIM model thanks to predefined transformation rules. After that, the PIM can be transformed in one or more PSMs, according to the adopted technologies. In the subsequent phases the implementation code can be directly derived by the PSM models. The different abstraction levels that characterize the models allow the stakeholders to work independently on different abstraction levels whilst remaining always synchronized with each other since, in an MDA specification of a system, CIM requirements should be traceable to the PIM and PSM constructs that implement them, and vice versa (backtracking).

Note that the transformation process can be automatable by specific tools, but may require additional information that is not included in the starting model.

MDA adoption implies several advantages. From a productivity point of view, the developer's focus is no more on manual code generation, but on PIM development, leveraging automatic transformation to PSMs in order to obtain different platform-specific implementations. Working at PIM level means that technological details may be omitted and that the developer can work on a model more similar to reality than pure code. As a consequence it is possible to develop better and faster. Moreover the MDA approach simplifies the management and maintenance of software documentation since high-level formal models are easy to understand and directly relate to the corresponding lower abstraction level models, down to the implementation code.

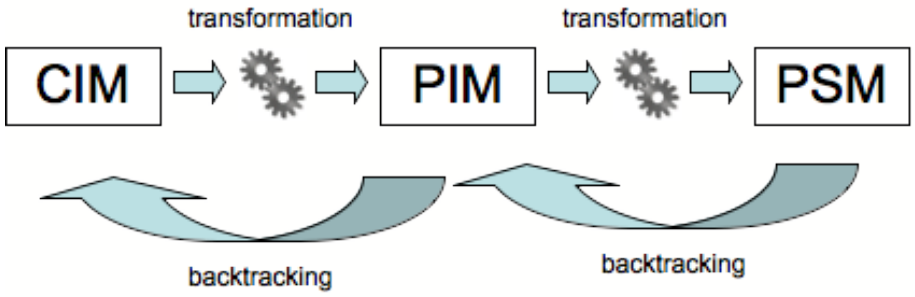


Figure 3.3: Main phases of MDA Development Process

3.2 Code Generation Scenarios [SUAS]

Scenario 1

This scenario deals with the generation of UML diagrams or XMI representations of UML diagrams in order to generate class stubs. With a special set of Vocabulary and Static Rules it should be possible to describe classes sufficiently so that for example an XMI representation can be described and generated. A considerable effort would be necessary to define the vocabulary, the parsing- and XMI-generation components so that the generation of UML class diagrams and therefore code stubs would be possible. We considered also reverse-engineering the processes which would make it possible to generate a code description in a natural language. This could enable people without knowledge of the programming language to read the structure of the code.

Scenario 2

By adding Vocabulary and Rules to Scenario 1, also Sequence diagrams could be described. Therefore, the Vocabulary and Rule set needs to be extended. Furthermore, we suppose that parts of the Vocabulary and Rules of Scenario 1 are needed as a basis in order to 'understand' items like classes and methods. It is interesting from our perspective to define sequences in the design phase of a component and test the method calls during the execution of the program against this specification. Nevertheless, from a software engineering point of view, only the generation of stubs is currently possible here and manual effort is needed to generate the code itself.

Scenario 3

From the perspective of Model Driven Architecture (MDA), SBVR is at the level where Computation Independent Models (CIMs) of a system are developed. The motivation behind the development of SBVR is to generate business designs which are CIMs in MDA, whereas UML is used to generate the Platform Independent Models (PIMs). This scenario maps some specifications written as CIMs to some other specifications which are PIMs. For details about this Secnario see Section 3.5.

Scenario 4

This scenario was prototypically implemented by SUAS for this deliverable. It should act as a feasibility study where we can evaluate the applicability of SBVR for application structure generation in an SME context. As the presentation of SBVR for SMEs needs to be simple and easy to adopt, we have chosen Groovy on Rails as the application framework of choice. It is web based and enables easy and fast prototyping. More details can be found in Section 3.3 of this work.

Scenario 5

Scenario 5 might have the potential to be one of the most promising approaches toward automated code and workflow generation based on SBVR. The work outlined here was performed by a sub-contractor of UCE with contributions from SUAS. The approach combines the definition of a workflow with its execution. Depending on the granularity of the components the execution can be seen as code generation OR workflow generation and consequently execution. Based on a subset of SBVR Vocabulary and Rules as well as their parsing components, SBVR is converted into the XML Process Definition Language (XPDL). The visual representation of XPDL is the Business Process Modeling Notation (BPMN). Therefore, a SBVR metamodel for BPMN has to be developed in parallel with the parsing and conversion components. A very narrow example, including the execution of a workflow in a Workflow-Execution Engine was initially planned for this deliverable. For details about this scenario see Section 3.4.

Scenario 6

Beside the approaches presented, the mapping of SBVR rules to Java Rules Engines for J2EE has to be mentioned here. Although this approach should be considered

in more detail, it is out of scope for this deliverable as we focussed on the scenarios above.

The following sections introduce the technical details of the implemented approaches described by the individual partners. The first part deals with the code structure generation of scenario 4. Later on the workflow scenarios 3 and 5 are explained.

3.3 Code Structure Generation Prototype [SUAS]

This Section deals with the technical details of the code structure generation prototype using the Grails framework as introduced in scenario 4. Firstly, the existing open source components we leveraged are introduced in order to provide the basic knowledge about the grounds for the prototype to the reader. After that, the architecture of the code structure generation prototype is explained in detail and finally the integration into the existing tools is outlined.

3.3.1 Components in Detail

Many commercial and open source tools as well as frameworks exist for Model Driven Engineering and Code Generation. As the project's policy and therefore the prototype's policy requires the use of open source licensed frameworks, only such components were considered in the selection process. The following sections introduce the frameworks and components used to develop which were not already introduced or developed in the predecessor project DBE (such as the DBE Servent[19]). The components will be introduced in a bottom-up approach as components can be the foundation of other components. The MDA approach as the foundation of the prototype will not be introduced as it was already outlined in Deliverable D2.1[17]. For an MDA overview see [18]. More details can be found in [20], [21], [22].

Groovy and Groovy on Rails

Grails[5], the acronym of Groovy on Rails, is the Java port of the widespread Ruby on Rails[2] (Rails) web application framework. Both frameworks help to generate database-backed web applications and strictly rely on the Model-View-Controller

[2] http://www.rubyonrails.org/

pattern and follow the FLOSS[3] philosophy. That Grails is the Java port is not fully true as only the framework itself is programmed in Java. Applications built using the Grails framework are developed in the Groovy[23] language, which itself is a standardized scripting language running in the Java Virtual Machine (JVM). The following Section introduces the basic concepts of Groovy as a foundation for Grails, which is explained afterwards.

Groovy is an agile, dynamic programming language for the JVM standardized through the Java Community Process in JSR-241[24]. Although the syntax of Groovy is very similar to the Java syntax, it borrows concepts from other languages like Python, Ruby and Smalltalk. Syntax differences are out of scope of this document and can be found in [23]. Further differences are changed default imports (which packages can be used in the source without importing the containing packages explicitly) and enhanced language features for example closures (which will be introduced in Java 7.0 Dolphin), native syntax for lists and maps and native support for regular expressions.

Currently Groovy is used in a considerable number of open source software containers (e.g. Apache ServiceMix and OpenEJB), web frameworks and testing tools. Additionally Groovy is used to build rules engines, MDA tools and even as a foundation for CHSM[4], a language system for specifying concurrent, hierarchical, finite state machines (an implementation of "state charts") to model and control reactive systems.

The advantage of scripting languages, like Groovy, in Digital Ecosystems is their dynamic nature. Scripts can be changed and recompiled during runtime. But as Groovy itself is developed in Java and because it is running in the JVM, all libraries available in Java can be used in Groovy as well without additional plumbing efforts. For this reason scripting languages were chosen for web frameworks like Grails. The following paragraphs introduce the Grails framework as the platform for the code structure generation prototype.

Grails

Grails claims to bring the *coding by convention*[5] paradigm to Groovy. By using the *coding by conventions* paradigm, standard naming conventions and directory conventions are used to construct applications instead of merely creating configuration

[3] free/libre open source software

[4] http://chsm.sourceforge.net/

files. Only in the case that naming conventions fail, configuration files with mapping information have to be created. The advantages of this approach are firstly that developers can quickly learn to use a framework by learning the convention, secondly that uniformity is promoted such that developers working on different projects using the same paradigm are able to understand other projects quicker, and finally that no configuration file changes are needed during refactoring processes. Therefore this paradigm is more dynamic than the static configuration approach. But there are disadvantages as well: the first problem is that it is nearly impossible for developers to use such a framework without being familiar with it. Secondly, by shifting the responsibility from the developer to the framework, such components are larger compared to lightweight frameworks configured by XML files. Finally it is harder to refactor existing software projects to follow such frameworks' convention. The *coding by convention* will be explained in this context in more detail when the prototype is explained and the output of the model transformation will be introduced, as this output has to follow the conventions enforced by the Grails web application framework.

Besides providing a convention for creating the web application files, Grails provides a Maven script to the developer in order to create the *Grails-Source*. Maven is an open source project management and build tool supported by the Apache Software Foundation. With commands like *grails create-app* or *grails create-domain-class*, wizards are started to create the application stub or domain classes. For details about Apache Maven see [25].

Grails' convention for the directory structure is outlined in Listing 3.1. The prototype creates files in the *domain* and *controllers* folders as all other necessary information will be extracted out of the provided information by the Grails framework.

```
1 PROJECT_HOME
2     + grails-app
3       + conf              ---> location of configuration artifacts
4                                like data sources
5           + hibernate       ---> optional hibernate config
6           + spring          ---> optional spring config
7       + controllers       ---> location of controller artifacts
8       + domain            ---> location of domain classes
9       + i18n              ---> location of message bundles for i18n
10      + services          ---> location of services
11      + taglib            ---> location of tag libraries
12      + util              ---> location of special utility classes
13      + views             ---> location of views
14          + layouts         ---> location of layouts
```

```
15   + lib
16   + scripts              ---> scripts
17   + src
18       + groovy               ---> optional; location for Groovy source files
19                                  (of types other than those in grails-app/*)
20       + java                 ---> optional; location for Java source files
21   + test                     ---> generated test classes
22   + web-app
23       + WEB-INF
```

Listing 3.1: Grails directory convention [5].

To generate a simple application it is necessary to create domain classes and controller classes. The domain classes are the starting point for any business application. Besides encapsulating the state about business process they also implement the behaviour of the business entities. Additionally the domain classes relate to one another via relationships. Domain classes are essentially normal classes with injected *id* and *version* properties and optional parameters to define validation and relationship information. A very basic example of a Grails domain class can be found in Listing 3.2. The controller class has to be created in an analogous manner: by creating the controller class or invoking the Grails wizard. Detailed information about that can be found in tutorials and references on the Grails homepage [5].

```
1 class Book {
2     static transients = [ "digitalCopy" ]
3     static constraints = {
4         releaseDate(nullable: true)
5     }
6
7     String author
8     String title
9     Date releaseDate
10    File digitalCopy
11 }
```

Listing 3.2: Grails domain class example [5].

As the basic components of Grails applications are introduced, the following Section explains the Eclipse Modelling Framework (EMF) - one basic building block of the model-to-code-transformation explained in more detail later in this report.

Eclipse Modelling Framework

The Eclipse Foundation is a non-profit organization supporting the Eclipse open source community, whose projects are focussed on development platforms, tools, frameworks, etc for building and deploying software. Details about the foundation,

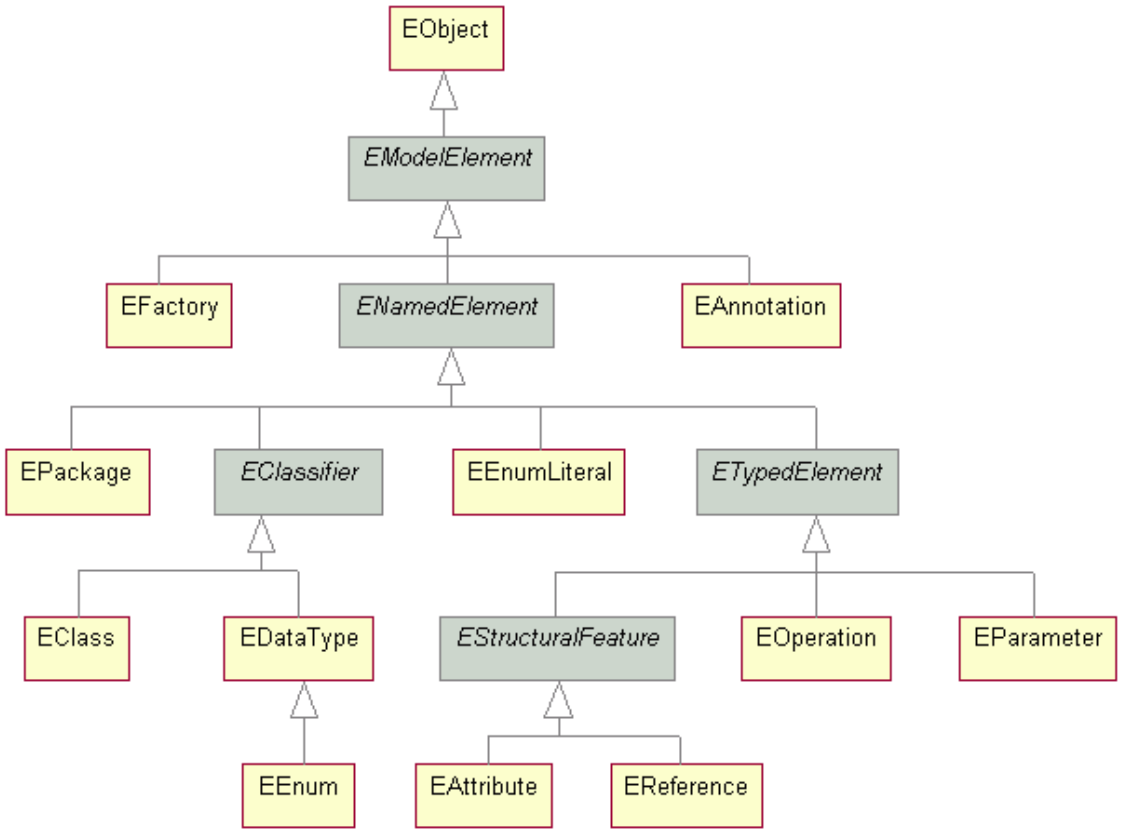


Figure 3.4: Ecore Metametamodel.

its own license and the history can be found in [26]. As the Eclipse Foundation supports a large number of projects related to software engineering, among them the Eclipse Platform[27] and the Eclipse IDE, it is not possible to explain all components used to generate the prototype. Therefore this Section only gives an introduction to the modelling components of the Eclipse Foundation - packaged together in the Eclipse Modelling Framework Project (EMF) [28],[29].

While EMF started as an implementation of the Meta-Object Facility (MOF) it has evolved to an independent M3 implementation named *Ecore*. EMF consists of two fundamental frameworks: the core framework and *EMF.Edit*. While the core framework provides basic generation and runtime support to the user for creating implementation classes of models, EMF.Edit adds support for the generation of adapter classes that enable the developer to rapidly create viewers and editors for the model. As EMF.Edit was not used it will not be explained in more detail in this deliverable.

Figure 3.4 shows the diagram of the Ecore metametamodel. Besides Ecore as a meta model for describing models, EMF also provides runtime support for these models. This means classes generated using EMF out-of-the-box, provide mechanisms for change notification, persistence support with default XMI serialization, and a reflective API for manipulating EMF objects generically. For example if Java classes are automatically generated using a Ecore model, for each class an interface and an implementation class is generated. The generated code of the implementation class already contains code for the functionalities mentioned above. Meta information in the code, so called *annotations*, allow the distinction between automatically generated code and manually developed code. This guarantees that the generated parts of the code can be updated if the model changes, without breaking developed code. Additionally, native support of XMI allows direct serialization and deserialization of generated model classes.

OpenArchitectureWare

OpenArchitectureWare (oAW) is a Model Driven Development (MDD) platform built on top of EMF. It uses Ecore as the base metametamodel and extends the capabilities of EMF with components for model transformation and modelling frontends. In the context of this paper, only the model transformation capabilities will be introduced. A comprehensive user guide can be found in [4].

oAW distinguishes between (1) Model-to-text (M2T), (2) Model-to-model (M2M) and (3) Text-to-model (T2M) transformations. The common denominator is a workflow engine which is used to plug different transformations together to a chain of transformations. How this workflow engine plumbs together the different components of the prototype is summarized in chapter 3.3.6.

While at the first glance the T2M case seems appropriate to parse the SBVR statements, it is not applicable as the relevant component, Xtext, is based on a specified language to create a model in textual format. Therefore, an oAW independent parser was developed which will be described in Section 3.3.4.

The leveraged components of oAW for the prototype are the already-mentioned workflow engine on the one hand, and the Model-to-text (M2T) engine on the other hand. The M2T part of oAW is represented through the Xpand2 template engine and its related Xpand template language. Xpand is a specialized template language for model transformation (in contrast to general purpose template languages like eXtensible Stylesheet Language (XSL). With Xpand it is possible to apply an input

template to input data that produces more than one output file (e.g. different class files). Additionally Xpand supports aspect oriented templating. Details about Xpand can be found in the reference included in the oAW user guide [4]. The template used in the Grails prototype is attached in the Annex C.3 and described in Section 3.3.5 of this work. Table 3.1 lists the most important elements that can be used in Xpand templates, named Import, Define, File, Expand, For, Foreach, If, Let and Error.

3.3.2 Architecture of the Prototype

The code structure generation prototype consists of two main components that are directly interlinked. The first component is the SBVR parser which transforms the model expressed in SBVR into an intermediate model. This intermediate model is based on a meta-model created using EMF. The second component takes the intermediate model in the XMI format as input and performs a transformation into the code structure. As the interface format is a model corresponding to a defined meta-model explained in Section 3.3.3 in the standardized XMI format, both components can be easily replaced with, for example, another output transformation. In order to give a proper overview, firstly the meta-model for the intermediate model will be introduced. After that the first component responsible for the parsing of SBVR statements and transforming to the intermediate model will be explained. After that the code structure generation will be outlined and finally the workflow plumbing these steps together will be described.

3.3.3 Intermediate Meta-Model

This section explains the meta-model of the intermediate model to help understanding the transformation performed by the code structure generation prototype. Before that two terms have to be introduced in order to be clear in the further terminology:

Intermediate Model The intermediate model represents the concepts described by the parsed SBVR statements. The intermediate model can be seen as the working data used in the prototype. Consequently the intermediate model conforms to the meta-model described below.

Intermediate Meta-Model The meta-model of the intermediate model defines the structure how the intermediate model is organized. This meta-model is

IMPORT	Using the IMPORT statement, devlopers do not need to use fully qualified names of your types and definitions as whole namespaces can be imported.
DEFINE	The central concept of Xpand is the DEFINE block, also called a template. This is the smallest identifiable unit in a template file. The tag consists of a name, an optional comma separated parameter list as well as the name of the meta model class for which the template is defined.
FILE	The FILE statement redirects the output generated from its body statements to the specified target.
EXPAND	The EXPAND statement 'expands' another DEFINE block (in a separate variable context), inserts its output at the current location and continues with the next statement. This is similar in concept to a subroutine call.
FOR	The definition is executed for given number of times.
FOREACH	This statement expands the body of the FOREACH block for each element of the target collection that results from the expression. The current element is bound to a variable with the specified name in the current context.
IF	The IF statement supports conditional expansion. Any number of ELSEIF statements are allowed. The ELSE block is optional. Every IF statement must be closed with an ENDIF. The body of an IF block can contain any other statement, specifically, IF statements may be nested.
LET	LET lets you specify local variables.
ERROR	The ERROR statement aborts the evaluation of the templates by throwing an XpandException with the specified message.

Table 3.1: Xpand Statements (adopted from [4])

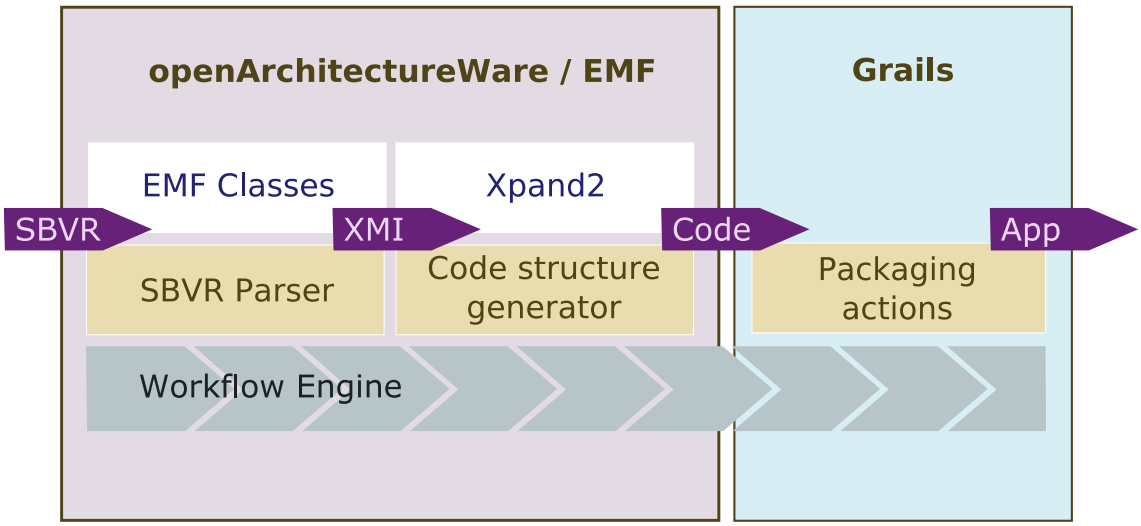


Figure 3.5: Architecture of the prototype

created using the EMF framework introduced above and itself conforms to the underlying Ecore meta-meta-model.

The UML diagram in Figure 3.7 shows the Meta Model of the Prototype which uses the following four different classes:

Model The Model class serves as the root element and contains all Entities of the intermediate model. The Model is, like all other intermediate model classes, of EMF type EClass and contains all Entity-Entries of the model. The Model can contain two additional parameters: a name and a package. While the

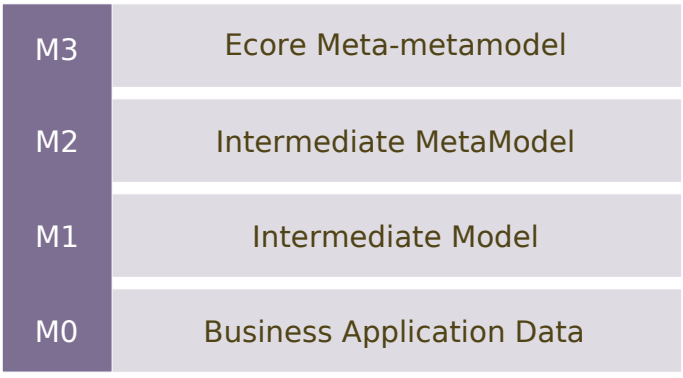


Figure 3.6: Meta-Level Overview

name is used as a general identifier, the package specifies the root-package for the generated source files.

Entity The Entity class is used to encapsulate the concepts defined in SBVR and contains the name of the concepts and which attributes are used to identify one individual concept (e.i. one instance of the Entity class). In parallel to that the Entity class encapsulates its own Attributes and References to other concepts.

Attribute Attributes are used to model data elements of Entities or SBVR-Concepts. Attributes can have different data types but in the case of the prototype all attributes are treated as text. Future developments of the prototype can include different datatypes. In Figure 3.8 it can be seen that the intermediate model already includes different data types.

Reference As the word *reference* already implies, they are used to model relationships between different concepts. A reference always expresses that one concepts references another concept. The simplest case is a one-to-one relationship which is currently fully functional in the prototype. N-to-m relationships can be modelled in the intermediate model but are not fully working in the code structure generation component. Additionally by using terms like *must have* or *can have* the modality of the relations can be influenced as needed.

It is not complex to produce code using the generation tools of EMF to generate and manipulate models based on that meta-model. The structure of these generated classes can be seen in Figure 3.8 which is a screenshot of the classes in Eclipse representing the meta-model. Using these classes, the SBVR parser described in the next Section creates the intermediate model out of SBVR statements. This intermediate model can then be further processed or serialized into the XMI format.

3.3.4 SBVR Parsing Component

Listing 3.3 shows basic SBVR statements that are used as the input model for the exemplary transformation. At the beginning of the Listing the three concepts *Driver*, *Car* and *Wheel* are introduced. After that facts about these concepts are used to add attributes and references to the concepts in order to express relationships.

The concepts and facts are read by a parsing component specifically developed for this purpose. This component reads the statements and transforms the data

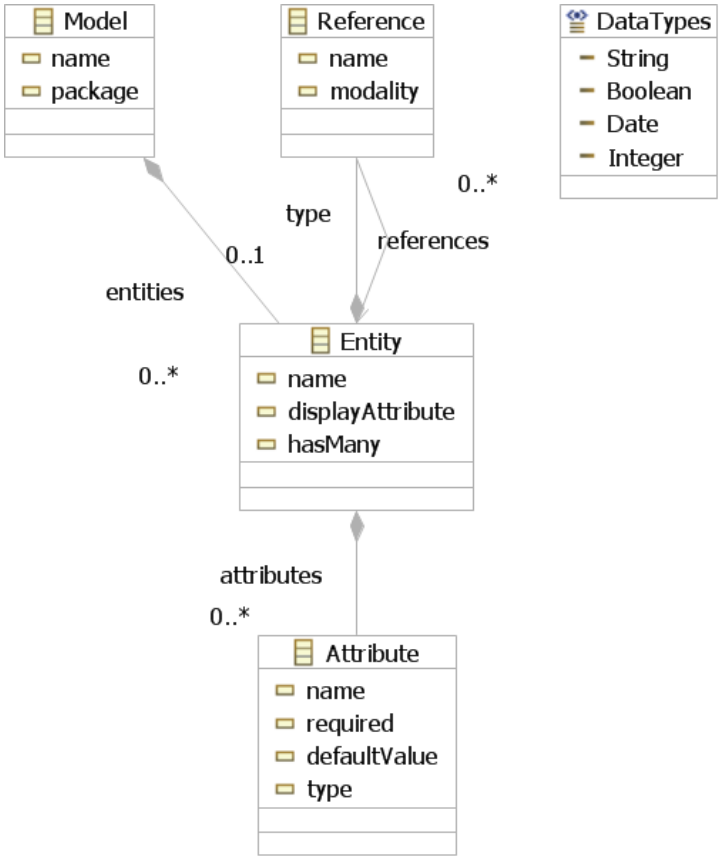


Figure 3.7: UML Graph of the Grails Prototype Meta Model.

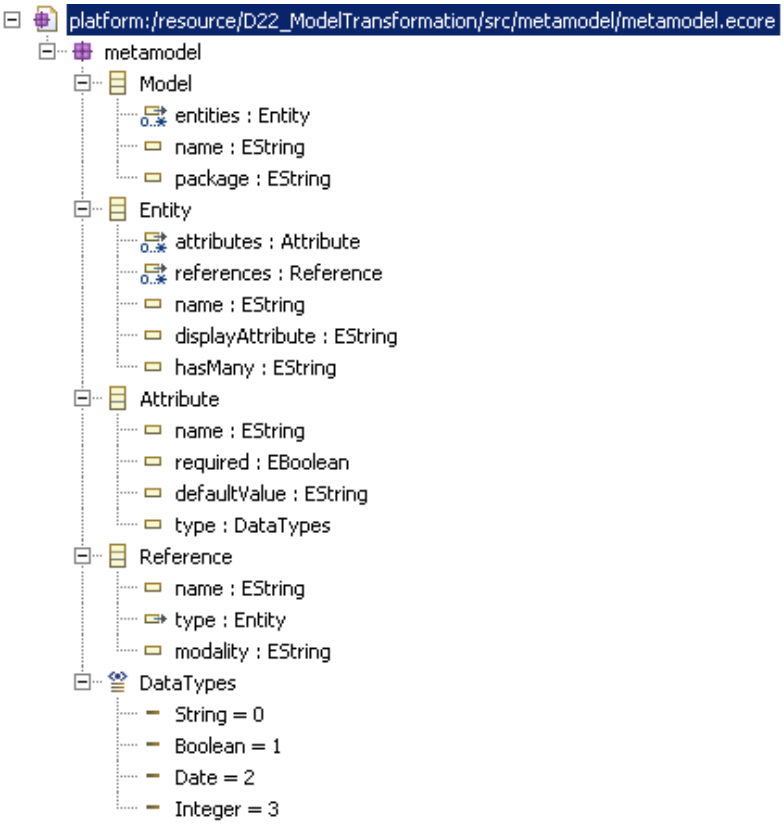


Figure 3.8: Meta Model of the Grails Prototype.

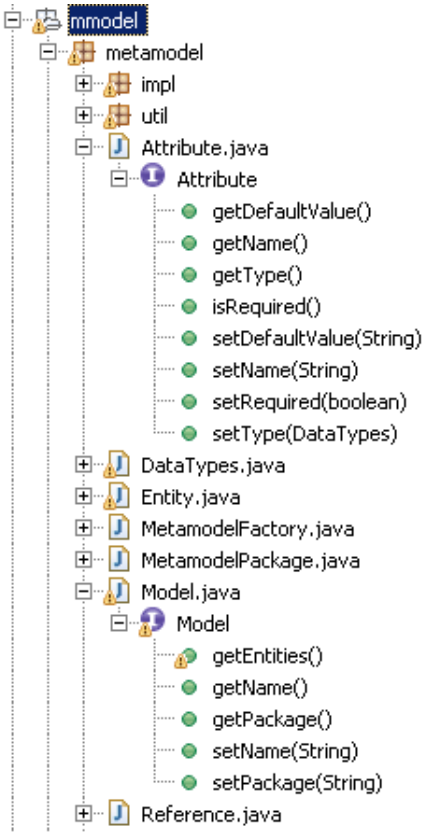


Figure 3.9: Generated Classes of the Grails Meta Model.

into the intermediate model which was already described in Section 3.3.3. The intermediate model is then fed into the next component in the standardized XMI format which can be seen in Listing 3.4. The next Section elaborates on the code structure generation out of the intermediate model.

```
Driver
Car
Wheel
Car has type
Car has power
Car has Driver
Car must have 4 Wheel
Wheel has manufacturer
Wheel has type
Driver has firstname
Driver can have middlename
Driver has lastname
Driver has age
Driver has gender
Driver is identified by lastname and firstname
Car is identified by type
```

Listing 3.3: Sample SBVR statements

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
 xmlns:metamodel="http://www.example.org/metamodel" name="MyModel"
 package="">

 <entities name="Driver" displayAttribute="lastname + ' ' + firstname">
  <attributes name="firstname" required="true" defaultValue="" />
  <attributes name="middlename" defaultValue="" />
  <attributes name="lastname" required="true" defaultValue="" />
  <attributes name="age" required="true" defaultValue="" />
  <attributes name="gender" required="true" defaultValue="" />
 </entities>

 <entities name="Wheel">
  <attributes name="manufacturer" required="true" defaultValue="" />
  <attributes name="type" required="true" defaultValue="" />
 </entities>

 <entities name="Car" displayAttribute="type"
  hasMany="DriverRef:Driver,WheelRef:Wheel">
  <attributes name="type" required="true" defaultValue="" />
  <attributes name="power" required="true" defaultValue="" />
  <references name="DriverRef" type="//@entities.0" modality="1" />
  <references name="WheelRef" type="//@entities.1" modality="4" />
 </entities>
```

```
26
27 </metamodel:Model>
```

Listing 3.4: Example XMI File corresponding to the Prototype Meta Model

3.3.5 Code structure generation

The foundation of code structure generation is the Xpand2 template engine part of the OpenArchitectureWare framework. The Xpand2 template engine requires the intermediate model as input on the one hand, and the Xpand2 template for the transformation on the other hand. The first input was described in the previous Section. The latter part is attached to this deliverable in the Appendix C.3.

The template in the Xpand language as specified in [4] contains the following main body:

```
<<DEFINE grailsModel FOR Model>>
  <<FOREACH entities AS entity>>
    <<EXPAND grailsModelClass(package) FOR entity >>
    <<EXPAND grailsControllerClass(package) FOR entity>>
  <<ENDFOREACH>>
<<ENDDEFINE>>
```

This definition shows that two different types of code structure files are created. First, the sub-template for the Model-Classes is invoked which creates one Model output file for each Entity. Second, the sub-template for the Controller class is invoked, which creates one Controller output file for each Entity in the input model. The motivation for this approach is that Grails expects at least these two files per Entity (Grails terminology: Domain Class).

The full template is attached in Appendix C.3. It contains the sub-templates for the Model-Classes and Controller-Classes. The distinction between attributes and references can be seen in the template of the Model-Classes, and how the individual filenames are created, as they directly reflect the Entity name.

The next Section explains how the components are aligned together to one chain of seamlessly connected transformation steps.

3.3.6 Transformation Workflow

As introduced in the oAW introduction 3.3.1 a workflow engine is used to connect all the steps to one integrated process. The integrated workflow engine is a declarative

configurable generator engine configurable via a simple XML file. This XML file contains the *generator workflow* as it represents all steps necessary to generate any output format out of an input model. Such a generator outflow consists of a number of straightforward generator steps that are executed sequentially. The generator workflow of the code structure generation prototype can be found in Appendix C.2.

The first three steps of the attached generator workflow are used for data cleansing. The fourth step parses the input SBVR data and stores the model as text file in the standardized XMI format. This XMI data in the next step is read in and stored in the workflow pipeline. The workflow pipeline is used to transfer data between the workflow components. After that the Xpand2 generator is invoked which is responsible for the code structure generation. Finally in the last three steps of the generator workflow, the output files (the Grails application) are packaged in a Zip-Archive and ununsed files are removed again.

The advantage of the workflow approach for the code generation is, that the workflow easily can be changed according to the needs of the developers, as there is no hardcoded data flow in the prototype. Components can be added, removed or exchanged in order to adapt the prototype to use different input formats or generate different output formats as needed. The following Section is dedicated to the integration of the code structure generation into the Servent infrastructure.

3.3.7 Servent Service

As the prototype was planned to be open for enhancements - i.e. adding support for different source or target-formats without modifications to the editors, the code generation prototype was designed as a service running in an application container. The additional advantage of this approach is, that the transformations do not have to be tightly integrated into the editor but can be searched during runtime. To be independent of a central point of authority, the DBE Peer-to-Peer infrastructure was chosen as the platform of the transformation service. More information about the P2P infrastructure known as *Servent* can be found in [19].

The interface of the service can be found in Listing 3.5. As can be seen, the input format is SBVR text. The output format is *SBVR2GrailsResult*, which is a wrapper class for the archived Grails application data.

```
1 package at.opaals.sbvrserventservice;
2
3 /**
4  * This is the public interface for the SBVR2GrailsService
```

```
5  * used in the DBE Servent infrastructure.
6  */
7 public interface SBVR2GrailsService {
8
9  /**
10   * Transforms the SBVR string to Grails code.
11   * The Grails code is returned in a zip-file
12   * contained in the returned SBVR2GrailsResult instance.
13   *
14   * @param sbvrDescription SBVR code to be transformed.
15   * @return zip-file marshalled in the SBVR2GrailsResult class.
16   * @see SBVR2GrailsResult
17   * @see SBVR2GrailsServiceImpl
18   */
19  public SBVR2GrailsResult transformSbvrToGrails(String sbvrDescription);
20
21 }
```

Listing 3.5: SBVR Transformation Service Interface

Listing 3.6 shows the invocation of the service using the DBE Servent infrastructure. The service is bound to a proxy which is responsible to forward the interface calls to a service deployed somewhere in the P2P infrastructure. The information flow can be seen in the sequence diagram in Figure 3.10. After the proxy is initialized, the proxy object serves as a wrapper to the Servent infrastructure connection on the one hand, and as a service on the other hand. Using Java Reflection mechanisms, a proxy can be instantiated as a service, in order to *to pretend* to be a class implementing a given interface [30]. When the *transformSbvrToGrails* is invoked on the proxy object, as can be seen in listing 3.6, the request is *marshalled* and sent to the configured Servent, which is responsible to route the service invocation to an appropriate Servent. After the service was executed at a Servent in the P2P network, the response is again marshalled and sent back to the service consumer.

The following Section explains how the principle is used to integrate the transformation service into the Business Modeller, as the editor of choice for SBVR authoring and SBVR related transformations.

```
1  /**
2   * Exemplary invocation of a service deployed in the P2P servent
3   * infrastructure of the DBE project.
4   */
5  URL servent = new URL("http://www.sepiax.org:2728");
6
7  ClientHelper ch = new ClientHelper(servent);
8
9  SBVR2GrailsService ds = (SBVR2GrailsService) ch.getProxy(SBVR2GrailsService.class
       ,new String[] {
10   "at.opaals.sbvrserventservice.SBVR2GrailsService"
```

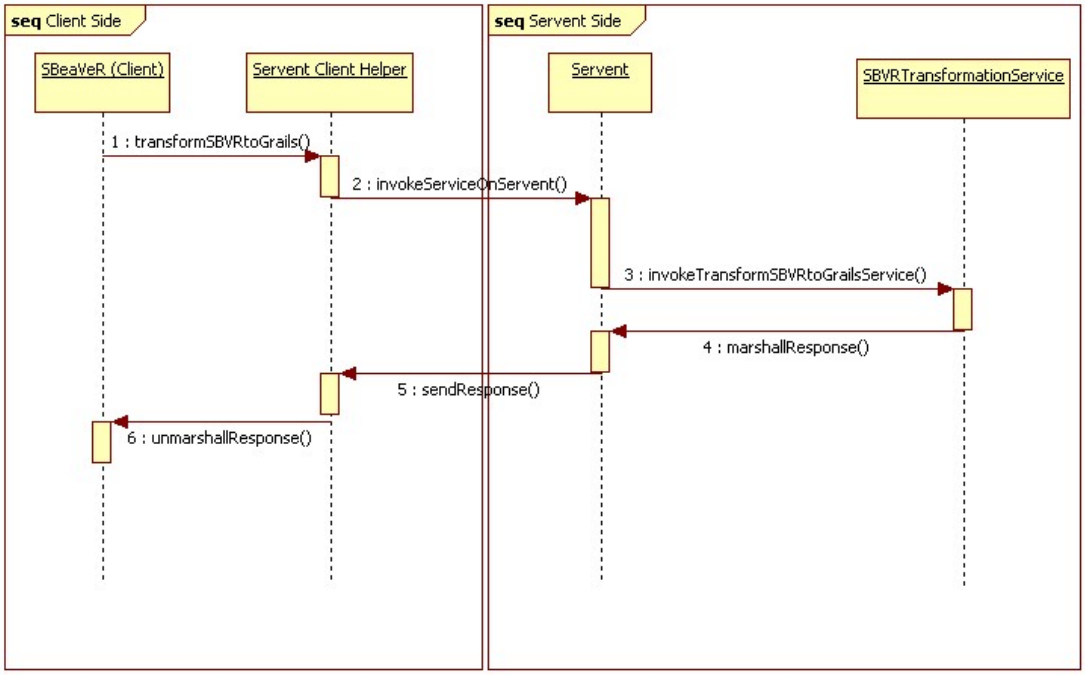


Figure 3.10: Service call using the Servent infrastructure.

```
11      });
12
13    SBVR2GrailsResult res = ds.transformSbvrToGrails(buf.toString());
14    res.unmarshall("c:/work");
```

Listing 3.6: SBVR Transformation Service Invocation

3.3.8 Integration

Currently the transformation services are integrated in the Business Modeller as service calls through the Servent infrastructure. In the case of the Grails transformation service, users can press a button to initiate the transformation workflow. The output of the process is stored into a defined folder on the local storage where the Grails application should be installed. As one of many features of Grails is automatic reloading of the generated classes on runtime, this can even be used to update the business application during runtime.

The next step towards distribution is a dynamic lookup of suitable transformation services at runtime such that the user can choose between a number of different

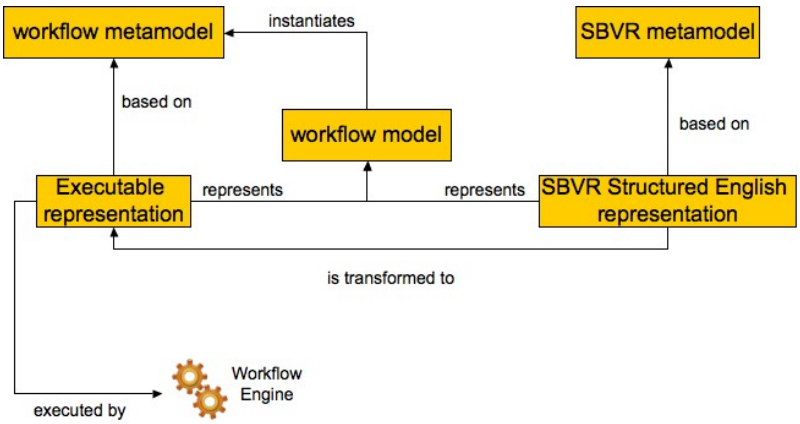


Figure 3.11: The adopted approach automatic workflow generation from SBVR models

offered transformations. This can even be integrated with the SBVR repository described later in this document.

The first part of this chapter was dedicated to the Grails code structure generation prototype, its components and its architecture. The following sections introduce the workflow generation out of SBVR models.

3.4 XPDL workflow generation [UCE]

This Section describes an approach to automatic workflow generation from SBVR models, with a particular focus on executable workflow models, where "executable workflow" means: a workflow implemented through the orchestration of existing software components (e.g. libraries, web services, ...). An executable workflow model is thus a formal representation of the flow of activities needed to achieve an objective, not of the actual implementation of such activities.

Figure 3.11 represents an overview of the approach that has been adopted here for automatic workflow generation from SBVR models.

The aim is to represent a workflow model (that instantiates a given workflow metamodel) through SBVR Structured English (that is based on the SBVR metamodel) and to define an automatic transformation from such representation to an executable one, if possible.

Such approach leads to two main questions:

- how to model a workflow by means of SBVR?
- is it possible to execute such a model?

The first question requires some preliminary considerations on the differences between the metamodels underpinning SBVR and workflow models. In particular it is important to point out that the SBVR metamodel does not provide any support to temporal logic, whereas a workflow model is heavily based on the sequence of activities and on other temporal-logic based concepts. Section 3.4.1 introduces the essential concepts related to a generic workflow leveraging the memetamodel defined by the Workflow Management Coalition (WfMC) and providing their graphical representation by means of BPMN (Business Process Modelling Notation), a graphical notation for modelling processes that has been standardised by the OMG (Object Management Group).

The second question requires the introduction of a particular class of software applications called workflow engine. A workflow engine manages and executes a business process that has been modelled in a specific language. Currently there are several workflow engines, both commercial and open-source, and languages for modelling and executing workflow in such engines. One of the most popular of such languages is the XML Process Definition Language (XPDL) that has been defined as a direct instantiation of the WfMC workflow metamodel. This characteristic makes XPDL, together with its maturity and popularity, our choice as the target executable representation language for transforming workflow models represented in SBVR. Moreover XPDL will be coupled with the Bonita workflow engine in order to actually run and demonstrate the effectiveness of the proposed transformation approach.

3.4.1 Introduction to BPMN and XPDL

The requirement of modelling workflows through SBVR makes it necessary to introduce the main aspects and concepts underpinning a workflow. This will be done describing the essential elements of the workflow metamodel defined by the WfMC (Workflow Management Coalition). This metamodel is the reference one for XPDL that is an xml-like language for workflow modelling and execution. Workflow models represented through SBVR will be transformed to XPDL in order to be executed by a specific workflow engine. Moreover BPMN will be leveraged to graphically represent the essential concepts of the WfMC metamodel.

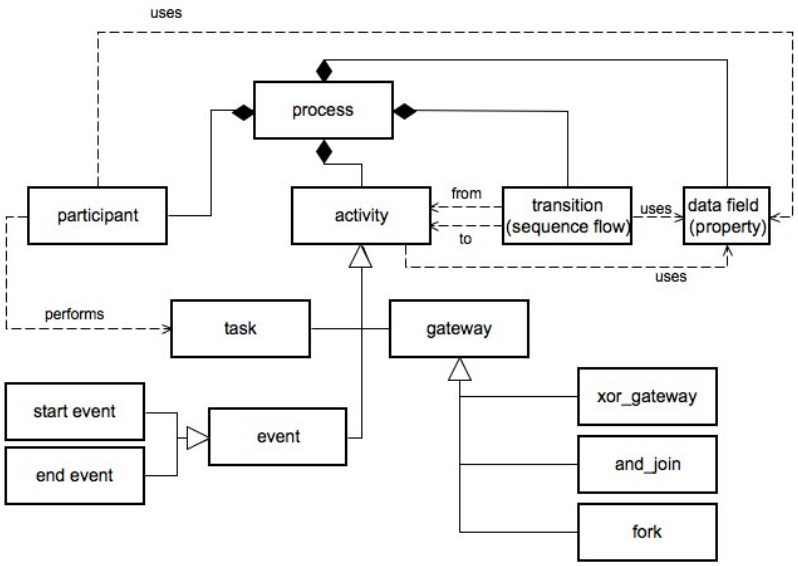


Figure 3.12: The adopted simplified workflow metamodel

Notice that, since the interest of this study is not to provide a complete way to represent workflows through SBVR models, only the essential elements of a workflow will be covered. In particular, an element is considered essential only if it is necessary in order to represent the simple workflow example that is described in 3.4.3.

Figure 3.12 provides an UML Class Diagram representation of the simplified workflow metamodel and Table 3.4.1 describes the subset of its elements that are of interest for this study introducing the correspondent BPMN representation.

In order to serialize and execute a modelled workflow, we used a specific execution language that can be interpreted and transformed into real activities by a workflow engine. Such a language is the XML Process Definition Language (XPDL) that is based on XML and was originally designed to allow the interchange of process models among different systems. XPDL is based on the WfMC metamodel, and in the following a generic XPDL representation of the above introduced workflow concepts is provided.

Task

A generic Task has the following XPDL representation:

```
<Activity xmlns="http://www.wfmc.org/2002/XPDL1.0" Id="" Name="">
 <Implementation xmlns="http://www.wfmc.org/2002/XPDL1.0">
 </Implementation>
```

Name	Description	BPMN representation
Task	A task is an atomic activity that is included within a process. A task is used when the work in the process is not broken down to a finer level of process model detail.	
Sequence Flow	A Sequence Flow is used to show the order that activities will be performed in a process.	
Start Event	An event is something that "happens" during the course of a business process. As the name implies, the Start Event indicates where a particular process will start.	
End Event	As the name implies, the End Event indicates where a particular process will end.	
Partecipant	Participant refers to the resources that can act as the performer of the various activities in a process definition. The participant does not necessarily refer to a human or a single person, but may also identify a set of people of appropriate skill or responsibility, or machine automata resource rather than human.	

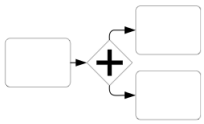
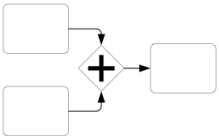
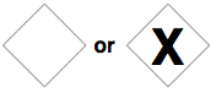
Gateway	A gateway is used to control the divergence and convergence of the flow of a process. Thus, it will determine traditional decisions points, as well as the forking, merging, and joining of paths. The BPMN representation of a gateway is a diamond shape. Internal markers will indicate the type of behaviour control.	
Fork	Fork is a type of gateway that refers to the dividing of a path into two or more parallel paths (also known as an AND- Split). It is a place in the process where activities can be performed concurrently, rather than sequentially.	
Join (AND-Join)	Join is a type of gateway that refers to the combining of two or more parallel paths into one path (also known as an AND- Join or synchronization).	
Exclusive Gateway	An Exclusive Gateway (XOR) restricts the flow such that only one of a set of alternatives may be chosen during runtime.	
Data Field	Data Field defines the data created and used during process execution. The data is made available to activities or applications executed during the process and may be used to pass persistent information or intermediate results between activities and/or for evaluation in conditional expressions. XPDL includes definition of various basic and complex data types, (including date, string, etc.). Activities, invoked applications and/or transition conditions may refer to process relevant data field. There is no BPMN representation for this concept.	

Table 3.2: Description of the essential workflow concepts

```
 <Performer xmlns="http://www.wfmc.org/2002/XPDL1.0">
 </Performer>
 <StartMode xmlns="http://www.wfmc.org/2002/XPDL1.0">
 </StartMode>
 <TransitionRestrictions  xmlns="http://www.wfmc.org/2002/XPDL1.0">
  <TransitionRestriction xmlns="http://www.wfmc.org/2002/XPDL1.0">
   <Join xmlns="http://www.wfmc.org/2002/XPDL1.0" Type="OR"/>
  </TransitionRestriction>
 </TransitionRestrictions>
</Activity>
```

Transition

A generic Transition has the following XPDL representation:

```
<Transition xmlns="http://www.wfmc.org/2002/XPDL1.0" From="" To="" Id="" Name="">
 <Condition xmlns="http://www.wfmc.org/2002/XPDL1.0" Type="CONDITION">
 </Condition>
</Transition>
```

Gateway

A generic Gateway has the following XPDL representation:

```
<Activity xmlns="http://www.wfmc.org/2002/XPDL1.0" Id=\"" Name="">
 <Route xmlns="http://www.wfmc.org/2002/XPDL1.0" />
 <StartMode xmlns="http://www.wfmc.org/2002/XPDL1.0">
  <Automatic xmlns="http://www.wfmc.org/2002/XPDL1.0" />
 </StartMode>
 <TransitionRestrictions xmlns="http://www.wfmc.org/2002/XPDL1.0">
  <TransitionRestriction xmlns="http://www.wfmc.org/2002/XPDL1.0">
   <Join xmlns="http://www.wfmc.org/2002/XPDL1.0" Type=""/>
  </TransitionRestriction>
 </TransitionRestrictions>
</Activity>
```

Note that XPDL does not represent a Fork as a Gateway, thus using the above code. A Fork is represented using only the Transitons from the starting task to the ending tasks.

Participant

A generic Participant has the following XPDL representation:

```
<Participant xmlns="http://www.wfmc.org/2002/XPDL1.0" Id="" Name="" >
 <ParticipantType xmlns="http://www.wfmc.org/2002/XPDL1.0" Type=""/>
 <ExtendedAttributes xmlns="http://www.wfmc.org/2002/XPDL1.0">
  <ExtendedAttribute xmlns="http://www.wfmc.org/2002/XPDL1.0" Name="NewParticipant"
       Value="" />
 </ExtendedAttributes>
</Partecipant>
```

Data Field

A generic Data Field has the following XPDL representation:

```
<DataField xmlns="http://www.wfmc.org/2002/XPDL1.0" Id="" Name="" >
 <DataType xmlns="http://www.wfmc.org/2002/XPDL1.0">
  <BasicType xmlns="http://www.wfmc.org/2002/XPDL1.0" Type="STRING" />
 </DataType>
</DataField>
```

3.4.2 From SBVR to BPMN/XPDL

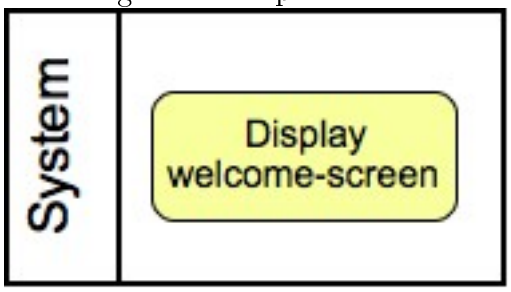
The process of transforming an SBVR representation of a workflow to an equivalent BPMN/XPDL representation is based on a set of rules that define the mapping of specific SBVR Structured English syntactic patterns to correspondent workflow concepts, thus to correspondent BPMN graphical signs and to XPDL serialisations.

Vocabulary

Each binary fact type that has the pattern "`<placeholder1> <verbPhrase> <placeholder2>`" is mapped to a specific BPMN/XPDL task where:

- `<placeholder1>` represents the participant that performs the task;
- `<verbPhrase> <placeholder2>` represents the task.

For example the fact type form "system displays welcome-screen" is transformed to the following BPMN representation:



Moreover `<placeholder2>` is considered as a Data Field.
The correspondent XPDL code is constituted by:

- one tag for representing the tasks "Display welcome-screen";
- one tag for representing the participant "System";

- one tag for representing the data field "welcome-screen".

Each unary fact type that has the pattern "`<placeholder> <verbPhrase>`" is mapped to an Exclusive Gateway (decision point) where:

- `<placeholder>` represents a concept that has a boolean characteristic;
- `<verbPhrase>` represents a boolean characteristic of the concept that is represented by the placeholder.

Moreover a Data Field is created for representing the gateway condition. The name of such data field is obtained by the concatenation of the terms used for expressing the unary fact type.

Note that the current approach requires the characteristic represented by the unary fact type to be boolean, that is to have a binary set of allowed values (typically "true" and "false"). This means that the gateway must have exactly two outgoing paths. The path followed by the flow is determined by the truth-value of the characteristic. For example the fact type form "card is valid" is transformed to the following BPMN representation:

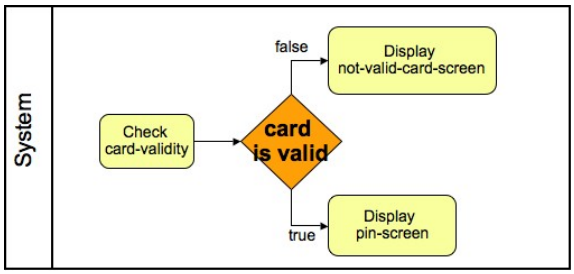
The correspondent XPDL code is constituted by:

- one tag for representing generic gateways;
- one tag for representing the data field "cardisvalid".

Ruleset

Each rule that has the pattern "`if <antecedent1> and <antecedent2> then <consequent>`", where `<antecedent1> and <consequent>` are both binary fact types and where `<antecedent2>` is a unary fact type, is mapped to a flow-path that involves a decision point (Exclusive Gateway). For example the rules "if the system checks the card-validity and the card is valid then the system displays the pin-screen." and "if

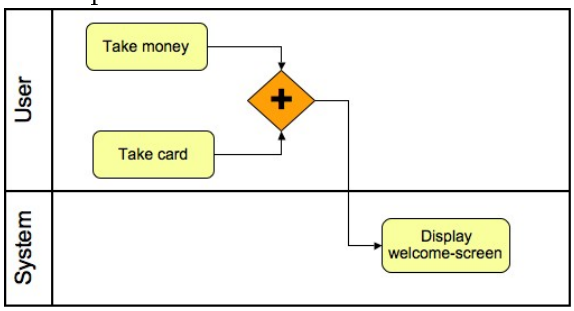
the system checks the card-validity and the card is not valid then the system displays the not-valid-card-screen." are transformed to the following BPMN representation:



The correspondent XPDL code is constituted by:

- three tags for representing the three tasks;
- one tag for representing the participant "System";
- one tag for representing the gateway;
- one tag for representing the transition from task "Check card-validity" to task "Display not-valid-card-screen" with condition value "false";
- one tag for representing the transition from task "Check card-validity" to task "Display pin-screen" with condition value "true";

Each rule that has the pattern "`if <antecedent1> and <antecedent2> ... and <antecedentN> th` with N arbitrary and where each `<antecedent> and the <consequent>` are binary fact types, is mapped to a flow-path that merges N flows into one through a Join (AND-Join). For example the rule "if the user takes money and the user takes the card then the system displays the welcome-screen." is transformed to the following BPMN representation:

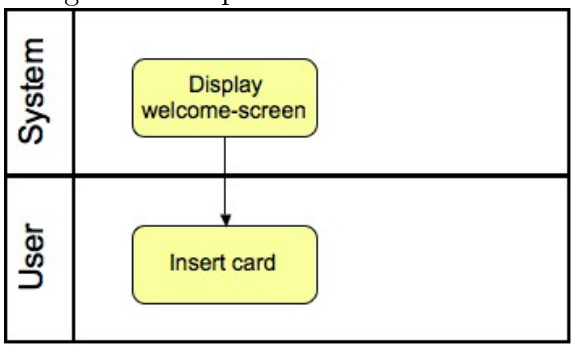


The correspondent XPDL code is constituted by:

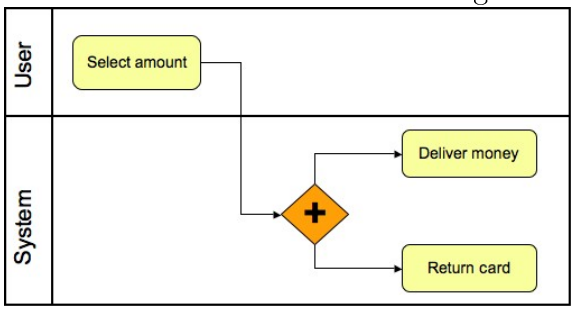
- three tags for representing the three tasks;

- two tags for representing participants "System" and "User";
- one tag for representing the gateway;
- one tag for representing the transition from task "Take money" to the gateway;
- one tag for representing the transition from task "Take card" to the gateway;
- one tag for representing the transition from the gateway to task "Display welcome-screen"

As a consequence, the simple BPMN Sequence Flow derives from the precedent mapping . Indeed each rule that has the pattern "`if <antecedent> then <consequent>`" (N = 1) is mapped to a Sequence Flow. Note that both `<antecedent> and <consequent>` are based on a binary fact type. For example the fact type form "if the system displays the welcome-screen then the user inserts the card" is transformed to the following BPMN representation:



Moreover rules that have the pattern "`if <antecedent> then <consequent>`" and that have the same binary fact type underpinning the `<antecedent>` are mapped to a Fork. For example the rules "if the user selects an amount then the system delivers money." and "if the user selects an amount then the system returns the card." are transformed to the following BPMN representation:



Finally also rules that have the pattern "`if <antecedent> then <consequent1> and <conseque`
... `and <consequentN>`", with N arbitrary and where the `<antecedent> and each <consequent>`

are binary fact types, is mapped to a Fork. For example the rule "if the user selects an amount then the system delivers money and system returns the card." has the same BPMN representation as the above case.

The correspondent XPDL code is constituted by:

- three tags for representing the three tasks;
- two tags for representing participants "System" and "User";
- one tag for representing the transition from task "Select amount" to task "Deliver money";
- one tag for representing the transition from task "Select amount" to task "Return card";

Note that for representing a fork, XPDL does not require to use the gateway tag.

3.4.3 Example: an ATM withdrawal

This Section introduces a simple example of the above introduced approach in real use. The example is based on a simple workflow scenario: a simplified withdrawal from an ATM. Even if the scenario is very simple, it illustrates all the essential concepts introduced in 3.4.1 and defined by the WfMC metamodel.

The process is cyclic and its start/end task is the visualization of a "welcome screen" by the ATM system. In case a user inserts his card in the ATM, the system reacts checking the card validity: if the card is not valid the system displays a warning screen and rolls back to the "welcome screen". It is assumed that the task "do rollback" includes the returning of the card to the user. If the card is valid, the system displays another screen prompting the user to insert his private PIN. After the user provides it, the system check for the PIN validity and, in case of negativity displays a warning message and rolls back to the "welcome screen". Again the roll back task includes the returning of the card to the user. In case the PIN is valid, the system prompts the user to select the amount of money he/she wants to withdraw. For sake of simplicity we don't modelled any check for the availability of such amount in the user's back account. Once the user selects the amount of money, the system delivers money and returns the card to the user. Note that in a real ATM system usually first the card is returned with a timeout of 30 seconds and then money is delivered. We decided to parallelize the two tasks in order to show the use of a fork.

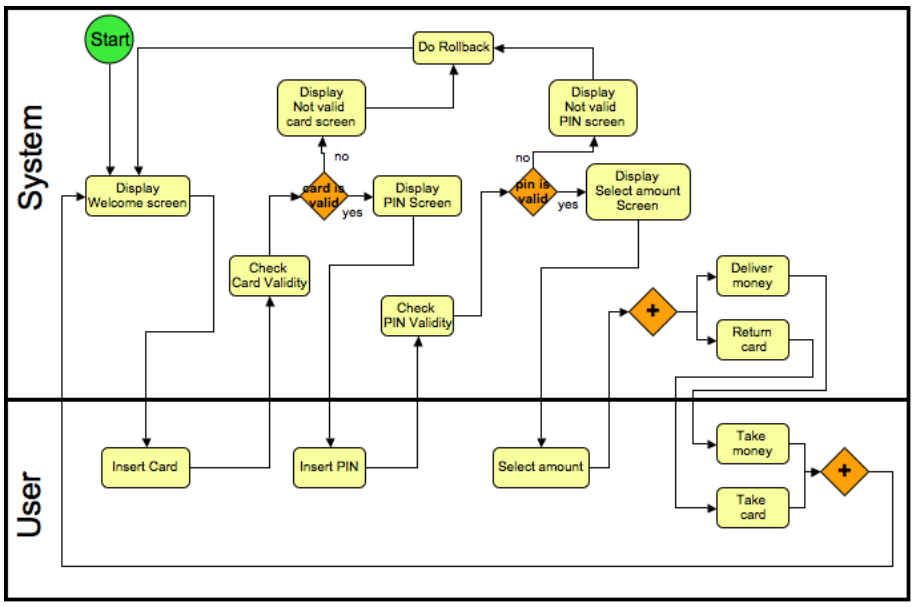


Figure 3.13: BPMN representation of the ATM workflow example

After the user takes both the card and the delivered money, the process ends and the system returns to the initial task "display welcome screen". Figure 3.13 provides a BPMN representation of the process in order to make it more understandable.

According to the transformation rules stated in 3.4.1, the workflow example can be represented by an SBVR model. In Figure 3.14 there is the SBVR Vocabulary of such model.

Moreover it is necessary to provide a ruleset based on the previous vocabulary in order to completely model the workflow example. (Figure 3.15)

The SBVR model described above can be automatically transformed to an XPDL representation.

3.4.4 Required improvements

Like most of the currently available workflow engines, Bonita uses standard XPDL (version 1.0) with some ad-hoc extended attributes to solve some problems that it is not possible to solve with only standard XPDL. Thus, in order to execute on Bonita the automatically generated XPDL code, it is necessary to manually modify it. Moreover some modifications are needed because not all of the required semantics of the XPDL construct can be automatically generated by the proposed approach.

Terms

system
user
welcome-screen
card
card-validity
pin-screen
not-valid-card-screen
pin
pin-validity
not-valid-pin-screen
rollback
select-amount-screen
amount
money

Fact types

system *displays* welcome-screen
user *inserts* card
system *checks* card-validity
card *is valid*
system *displays* not-valid-card-screen
system *does* rollback
system *displays* pin-screen
user *inserts* pin
system *checks* pin-validity
pin *is valid*
system *displays* not-valid-pin-screen
system *displays* select-amount-screen
user *selects* amount
system *delivers* money
system *returns* card
user *takes* card
user *takes* money

Figure 3.14: An SBVR vocabulary for modelling the ATM workflow example

if the system *displays* the welcome-screen then the user *inserts* the card.
if the user *inserts* the card then the system *checks* the card-validity.
if the system *checks* the card-validity and the card *is* not *valid* then the system *displays* the not-valid-card-screen.
if the system *displays* the not-valid-card-screen then the system *does* the rollback.
if the system *does* the rollback then the system *displays* the welcome-screen.
if the system *checks* the card-validity and the card *is valid* then the system *displays* the pin-screen.
if the system *displays* the pin-screen then the user *inserts* the pin.
if the user *inserts* the pin then the system *checks* the pin-validity.
if the system *checks* the pin-validity and the pin *is valid* then the system *displays* the select-amount-screen.
if the system *checks* the pin-validity and the pin *is* not *valid* then the system *displays* the not-valid-pin-screen.
if the system *displays* the not-valid-pin-screen then the system *does* the rollback.
if the system *displays* the select-amount-screen then the user *selects* an amount.
if the user *selects* an amount then the system *delivers* money.
if the user *selects* an amount then the system *returns* the card.
if the system *delivers* money then the user *takes* money.
if the system *returns* the card then the user *takes* the card.
if the user *takes* money and the user *takes* the card then the system *displays* the welcome-screen.

Figure 3.15: An SBVR ruleset for modelling the ATM workflow example

In the following a list of the required manual modifications is provided in order to allow an effective execution on Bonita of the generated XPDL code.

Name and identifier of the process

The proposed approach does not allow the specification a name and an identifier for the modelled workflow although XPDL requires such information. This implies that the attributes "name" and "id" must be manually set in the node "Package" and "WorkflowProcess". For example:

```
<WorkflowProcess Id="ATM" Name="ATM" AccessLevel="PUBLIC" >
```

User Autentication

In the Bonita run-time environment there are many domains for user management. Such domain must be specified explicitly by the XPDL code. In particular the following extended attribute (with information about the user authentication domain) is automatically added at the end of the node "WorkflowProcess".

```
<ExtendedAttributes>
 <ExtendedAttribute Name="initiatorMapper" Value="Ldap" >
  <InitiatorName>jonas-admin</InitiatorName>
 </ExtendedAttribute>
</ExtendedAttributes>
```

Moreover there is the need to specify the type of user (human or system) in the "participant" nodes. Participants of type "HUMAN" are used for tasks that are performed by human beings and that can not be totally automated (they need some kind of interaction with a human user), whereas participants of type "SYSTEM" are used for tasks that are performed by automatic systems and thus are totally automated (they do not need interaction with a human user). This is due to the fact that the proposed approach is not able to understand the type of user of each activity and that default choice is to assume every task is performed by a human. This implies that in every node "participant" that represents an automatic user, the node

```
<ParticipantType xmlns="http://www.wfmc.org/2002/XPDL1.0" Type="HUMAN" />
```

must be replaced by

```
<ParticipantType xmlns="http://www.wfmc.org/2002/XPDL1.0" Type="SYSTEM" />
```

Implementation components

The proposed approach for automatic workflow generation is concerned with the modelling of the flow of the activities needed to achieve an objective and about the orchestration of existing software components that are able to execute such activities. In particular Bonita allows the specification of each component inside the correspondent "activity". In case the activity is performed by the users of type "SYSTEM". We assume that the existing components are Java classes (these classes are called hooks). The information about the hook must be provided by a node "extended attribute" inside the proper node "activity". The hook value is the catenation of the string "*hero.hook.*", the ID of the overall process and the ID of the activity the hook is for.

```
<ExtendedAttributes>
 <ExtendedAttribute Name="hook" Value="hero.hook.processID_activityID" >
  <HookEventName>beforeTerminate</HookEventName>
 </ExtendedAttribute>
</ExtendedAttributes>
```

Moreover it is necessary to modify the node "StartMode" of each activity performed by a user of type "SYSTEM" from "Manual" to "Automatic".

```
<StartMode xmlns="http://www.wfmc.org/2002/XPDL1.0">
 <Automatic xmlns="http://www.wfmc.org/2002/XPDL1.0" />
</StartMode>
```

Cycles

When a workflow is not cyclic, Bonita is able to automatically find the first activity that must be executed since it is assumed that such activity has no input transitions. This approach is not valid in case the first activity is also the final one of a cycle and thus it has at least one input transition. The problem can be solved manually modifying the XPDL code. Indeed Bonita uses two type of transitions: normal transitions and iterations. Iterations are like normal transitions with conditions. To execute cyclic workflows into Bonita it is necessary to replace some transitions with iterations. In particular, for each cycles it is necessary to:

1. Find the first node of cycle;
2. For each transition (with or without condition) that starts from a node in the cycle and arrives to first node of cycle:
   (a) Delete the transition from the XPDL code;

(b) If the transition does not have a condition, put in the starting node of the transition this xml code:

```
<ExtendedAttribute Name="Iteration" Value="true">
 <To>name_of_end_point</To>
</ExtendedAttribute>
```

(c) If the transition has a condition put in the starting node of the transition this xml code:

```
<ExtendedAttribute Name="Iteration" Value="condition">
 <To>name_of_end_point</To>
</ExtendedAttribute>
```

3.5 Transformation of SBVR Business Model to UML Models [IITK]

From the perspective of Model Driven Architecture (MDA), SBVR is at the level where Computation Independent Models (CIMs) of a system are developed. The motivation behind the development of SBVR is to generate business designs which are CIMs in MDA, whereas UML is used to generate the Platform Independent Models (PIMs). This work maps some specifications written as CIMs to some other specifications which are PIMs.

SBVR directly deals with the declarative sentences, propositions but not with the imperative sentences or the sentences which represent some commands. The declarative sentences or propositions are either true or false but a command doesn't have a truth value. This work mainly points out a transformation from declarative form to imperative form.

UML involves around more than 10 concepts which can be used at the different level of abstraction of the system. Depending upon the perspective of the system, a particular UML concept is chosen for example: UML Sequence Diagram is used to represent the requirements of the system where as UML Class Diagram is used to represent the static structure of the system. We are not dealing with all those UML concepts at the moment. We are only interested in the three basic UML concepts Activity Diagram, Sequence Diagram and Class Diagram. It doesn't mean that the analysis of the system will stop here. Petri Selonen et. al., in [31] shows that with the help of these diagrams, other diagrams can be generated like Object Diagram, Collaboration Diagram, and State Chart Diagram etc.

3.5.1 SBVR to UML Activity Diagram

In SBVR the verb makes a relationship between the two terms. These verbs can be of two types, one which implies some action (action verbs) and another which doesn't imply any action (non-action verbs). For example, the verb 'has' and various forms of 'to be' doesn't imply any action. The fact type 'card has pin-number' shows the structure of how card and pin-number are related with each other. The fact types which involve action verbs can be considered as a representation of some activity. For example, the fact type 'atm ejects card' involves the verb 'ejects' which is an action verb. According to the above arguments, this fact type represents the action 'ejection of card'. This argument prompts us to consider the fact types having action verb as the activities for the rest of the discussion.

Categorization of Rules

In SBVR, a business rule is built on top of fact types. Our intension is to draw the activity diagram which represents the execution behaviour of the system. So, we must consider only those rules which represent the execution behaviour. Ignoring the structural rules, we will consider only operative rules. But all the operative rules are not automatable. By using the arguments given in [32], we can further categorize the operative rules according to their IT support as follows:

- **IT support = automated** These are the rule which should be completely handled by IT system without any intervention of human user. We will refer to these rules as Automatable rule.
- **IT support = supported** These are the rules which are only supported by IT system and expect some human intervention through the interface provided by them.

Automating a rule means to enforce a rule through automation. In general, an enforcement policy needs to be specified for each rule, putting an obligation on the system or process as to exactly how, when, and where the system or process is to enforce the rule. That is, there is a rule or set of rules for each automatable rule about how the automatable rule will be enforced by the system or process. This becomes part of the system specification, or process specification. This is often non-trivial, as there are often several options available, and it is a system or process design choice which one to use. Or the enforcement may be complex, involving many

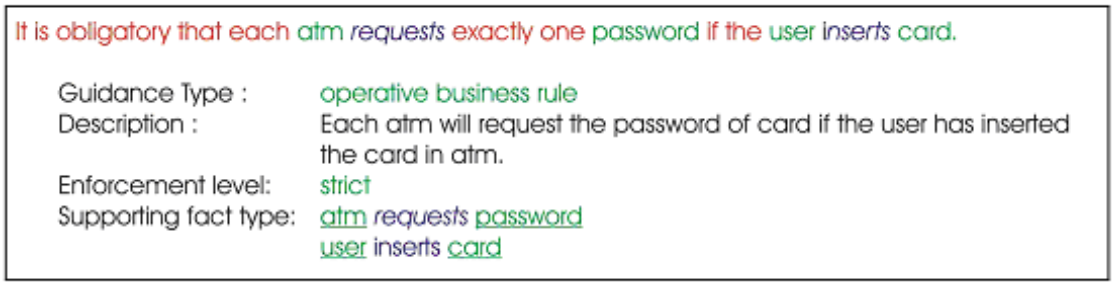


Figure 3.16: A SBVR Rule Signature

steps and coordinated activity to enforce the rule. This information is generally not in the automatable rule itself, but involves other considerations. The enforcement policy can be derived from the enforcement level given in the signature.

SBVR doesn't provide any provision for the distinction between automatable and non-automatable rules. The knowledge about this distinction is somewhere in the mind of rule expert, which needs to be transferred to the sbvr rule signature. A typical rule signature in SBVR is shown in Figure 3.16.

A typical rule in SBVR has the following attributes: 'Guidance Type' states whether it is a structural or operative rule, 'Description' states some related knowledge of the rule, 'Enforcement Level' provides information about how earnestly the business will enforce the rule, 'Supporting fact types' states the fact types which are combined through some conjunctions to compose the rules, and Related fact [33]. The attribute 'Enforcement Level' includes a list of levels [33] which is an example set of levels of enforcement based on Business Motivation Model (BMM) [34]. This list is not normative. We can add whatever levels of enforcement are appropriate for our purpose. Clearly, enforcement by machine is a perfectly good level of enforcement. A rule can have multiple levels of enforcement, e.g. automatable and pre-authorized, which would suggest the system should support pre-authorization work flow to allow an exception to the rule. Manual and pre-authorized would suggest the system would not be required to support pre-authorization of exceptions. Hence, we will include 'automatable' as a category of Enforcement Level that can be assigned to a rule, as shown in the Figure 3.17.

The fact types associated with the automatable rules will be used to form an ordered set of commands. It is a very sensitive and non-trivial process as it must consider both linguistic and logic especially the temporal aspects all of which may

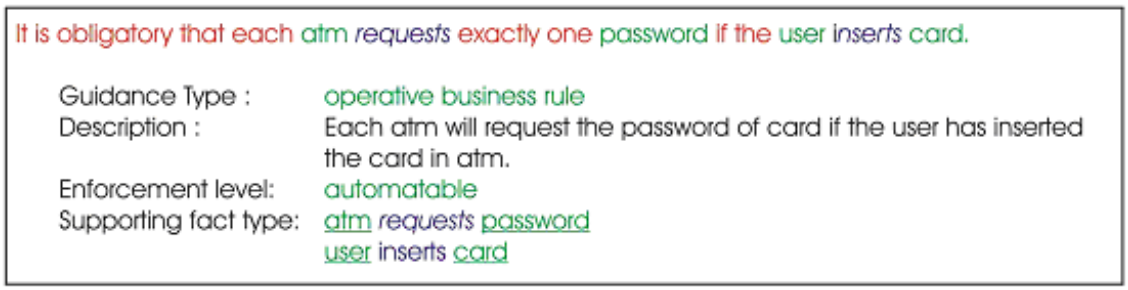


Figure 3.17: A SBVR Rule Signature with additional level of enforcement

not be specified directly in the set of declarative rules. This transformation must be logically consistent with the conceptual schema of SBVR. SBVR doesn't incorporate the temporal logics. So, this work will mainly entertain only 'if-then' rules to find out an ordered set of activities. To deduce the 'if-then-else', we will decompose the else part again in an 'if-then' construct. For example:

```
if p then q else r
```

can be decomposed as

```
if p then q and if !p then r
```

SBVR to UML Activity Diagram Mapping Rules

The work mainly focuses on the mapping of SBVR metamodel to UML Activity Diagram metamodel. The sections below shows how the different components of Activity diagram Metamodel can be captured from SBVR Specification.

Start Node This is the starting point of the activity diagram. It doesn't play a very important role but very important for a better reading of the diagram. We are creating a default start state with the default name "Start".

Fact Types As Activities An activity is the occurrence of an action during the course of execution. In SBVR, these actions are generally represented by the fact types including the action verbs. SBVR also has a provision for giving instances of fact types a type name; it is called 'objectification' [33]. For example, an occurrence of the fact type 'customer places order' might be objectified as a 'order placement'.

An objectification allows us to predicate things about states of affairs, like when they happened, etc. It depends upon the user whether he wants to directly use the fact type as an activity or uses the objectification. The correspondence transformation turns the state of affairs that is the meaning of an instance of an objectification into a command to bring such a state of affairs into existence. Use of the infinitival form of the verb phrase of the underlying fact type for an activity name might be preferred, to reflect the imperative nature of the activity: 'place order', the successful outcome of which is an 'order placement', which is a state of affairs (event) that a customer has placed an order.

Transitions A transition is a set of events, guard conditions and actions which transfers the control from one activity to another activity. An event is the trigger of the transition. Upon triggering the transition, the condition is checked, if the condition holds true, then the corresponding action occurs and brings another activity into existence. It is not necessary that a transition must have trigger. A transition without the trigger is known as triggerless transition. Transitions without guard conditions will occur initially.

A typical business operative rule embeds the above attributes like 'upon event', if the *condition is true*, then *do the action.*

The following format of operative business rules in SBVR *propositional expression 1* [if *propositional expression 2*] are used. Therefore, the mapping from these type of SBVR rules to UML AD will create the triggerless transitions. The propositional expression 2 will help to find out the guard condition and propositional expression 1 will help to find out the action.

Guard Condition Assume an operative business rule

```
It is obligatory that each atm request exactly one password
if a user inserts card
```

The propositional expression 1 in if clause refers to the fact type 'user inserts card'. Depending upon the fact type, we are creating a boolean variable like 'inserts_card'. And the guard condition will become 'insert_card == true'.

Action These are the actions which should be invoked during the transition from one activity to another. For example in the above rule, if a user inserts a card

in the atm, then the atm will do an action 'request password'. What we are doing is taking the propositional expression 1 and finding out the corresponding fact type. Merging the verb and last term of the fact type will create the name of the action like 'request_password()'.

Fork/Join

Fork This is a pseudo-state where one transition is coming and multiple parallel transitions are going out of it. A SBVR rule like 'it is obligatory that each atm print exactly one receipt and each atm eject the card if bank return badAccount-message' represent the enforcement of two activities 'atm print receipt' and 'atm eject card' on the delivery of bad account message from the bank. This situation will generate the fork state. There will be an incoming transition having the guard condition 'return_badAccountmessage == true' and two outgoing parallel transitions pointing toward activities 'atm print receipt' and 'atm eject card'.

Join This is another pseudo-state where multiple parallel transitions are coming and only one transition is outgoing. The generation of this state will be the same as the above except there will be multiple guard conditions and only one outgoing transition.

Decision Node As we are representing all the requirements in if-then rules, therefore the decision nodes don't appear in general.

Partitions The partitions in UML activity diagrams are generally known as swimlanes which basically represents who is doing the activity. The entity doing an activity will be referred as the giver of the activity.

In SBVR the representation of fact type (activity) can be of two types, active form and passive form. In a sentence having an active, transitive verb, the giver of the action of the verb is the subject of the sentence. In English, the 'giver' corresponds to the object filling the role of the first placeholder in the fact type form, e.g. 'customer' in 'customer places order.' If the fact type form is passive, e.g. 'order is placed by customer,' it is the reverse. These mean the same thing or are synonymous forms. Facts in either of these forms would be logically equivalent.

End States This is the end point of the activity diagram. We are creating a default start state with the default name 'End'.

Rule Sequence Engine (RSE)

RSE is an engine used to establish an order between the activities. The engine consists of a data structure used to contain the guard conditions which have been occurred as true and a decision unit to decide the next activity. For example, if we encounter a rule like

```
It is obligatory that each atm request exactly one password
if a user inserts card
```

The fact type corresponding the if clause is 'user inserts card' and corresponding guard condition is insert_card == true. The RSE searches its database for this condition, if the condition exist and holds true, then the activity 'atm request password' corresponding to the 'then clause' will be the next activity and a boolean variable request_password is created and get set to true and inserted into the database. This concept is motivated from OMG's Production Rule Representation (PRR) [35] and RETE Algorithm.

Algorithm

First of all, the vocabulary and rules are parsed and inserted into the SBVR Metamodel given in [33]. Chapter 9 in [33] deduces the logical formulation of a SBVR business rule. The logical formulation of automatable operative business rules will be of our interest at the moment, as we are modeling the activity diagram. Since we are generating the UML AD based on the business artifacts given only in if-then rules, there must be exactly one automatable rule which would not have any if clause, and that will be the starting point of activity diagram. If there is more than one such rule, then there will be two transitions from the start state which is against the UML AD Semantics. It is only possible if those two activities occur in parallel, which will be modeled as fork in AD.

The fact type corresponding to that automatable rule having no 'if clause', will be the first activity. Create a boolean variable as shown in the Section 1.1.3, assign it as true and put into the database of RSE. The detection of further activities will be done with the help of 'implications' given in SBVR Metamodel. The logical

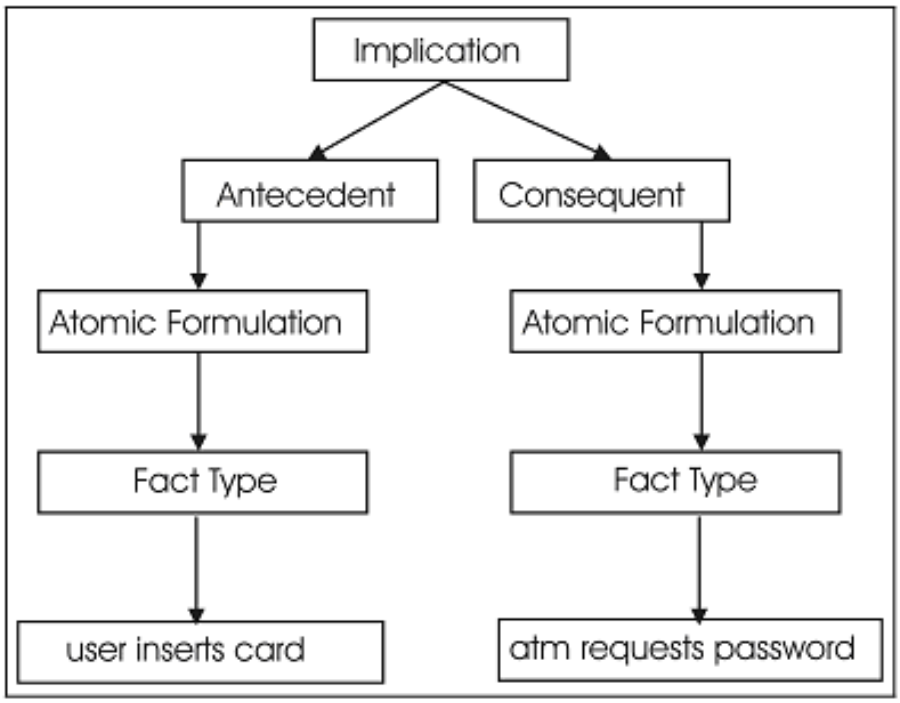


Figure 3.18: Implication: The logical formulation of if-then rule.

formulation of if-then rules is known as Implication. Markus Schacher in [36] writes that a rule structure can be of the following two types:

- Event Condition Action (ECA): This type of rules structure specifies that a particular action is to be carried out when a particular event occurs and a particular condition (guard) is true.
- Condition Action (CA): This type of rule structure specifies that a particular action is to be carried out when a particular condition is true.

We are using the CA rule structure which involves a conjunction of the activity and condition with consequent and antecedent of the implication, respectively. This will become more clear by looking at logical formulation of the rule

```
It is obligatory that each atm request exactly one password
if a user inserts card
```

in the following Figure 3.18.

After deducing the logical formulations, we will look up at all the 'obligation claims' [37] that are 'implications' to find out the relation between the fact types of 'if' construct (antecedent of the implication) and those of 'then' construct (consequent) in an if-then-construct. To find out the next activity, check the database of RSE to know whether the guard condition corresponding to the fact type in antecedent is true or not. If the guard condition holds, find out the fact type corresponding to consequent of the implication and make it as the next activity and also create a corresponding boolean variable with assigning 'true' value and put it into the database of RSE. Also, create a transition from previous activity to the current activity as explained in Section 1.1.2.3. If the guard condition doesn't holds, then search for the next implication. The absence of such an implication will result in the end state.

The flow chart showed in Figure 3.19 present a general algorithm to find out the activity diagrams. The special situation like an activity doesn't have any outgoing transition, fork and join, multiple incoming transitions to the end state are not shown in the flow chart. They can be directly hard coded on top of the basic algorithm shown in the flow chart.

3.5.2 SBVR to UML Sequence Diagram

The initial phase of a software life cycle is the requirement analysis. The requirement analysis includes the analysis of data, events and actions which are important elements in the interface between an IT system and business environment. An environment includes various scenarios each of which can be represented as a UML sequence diagram, which is described in [33]. This Section deals with the transformation of knowledge of a business environment written in SBVR to visual scenarios described as UML Sequence Diagrams.

SBVR to UML Sequence Diagram Mapping Rules

A UML Sequence Diagram (SD) can be represented as a set of messages with the information of their source object, destination object, and the initial message. Every message can have a guard condition to control the execution of event. Messages, MessageEnd, General ordering, event, state invariant and life line are the most important part in UML SD Metamodel. Some of these components can be directly mapped from the UML AD Metamodel. For example: The activities in AD can

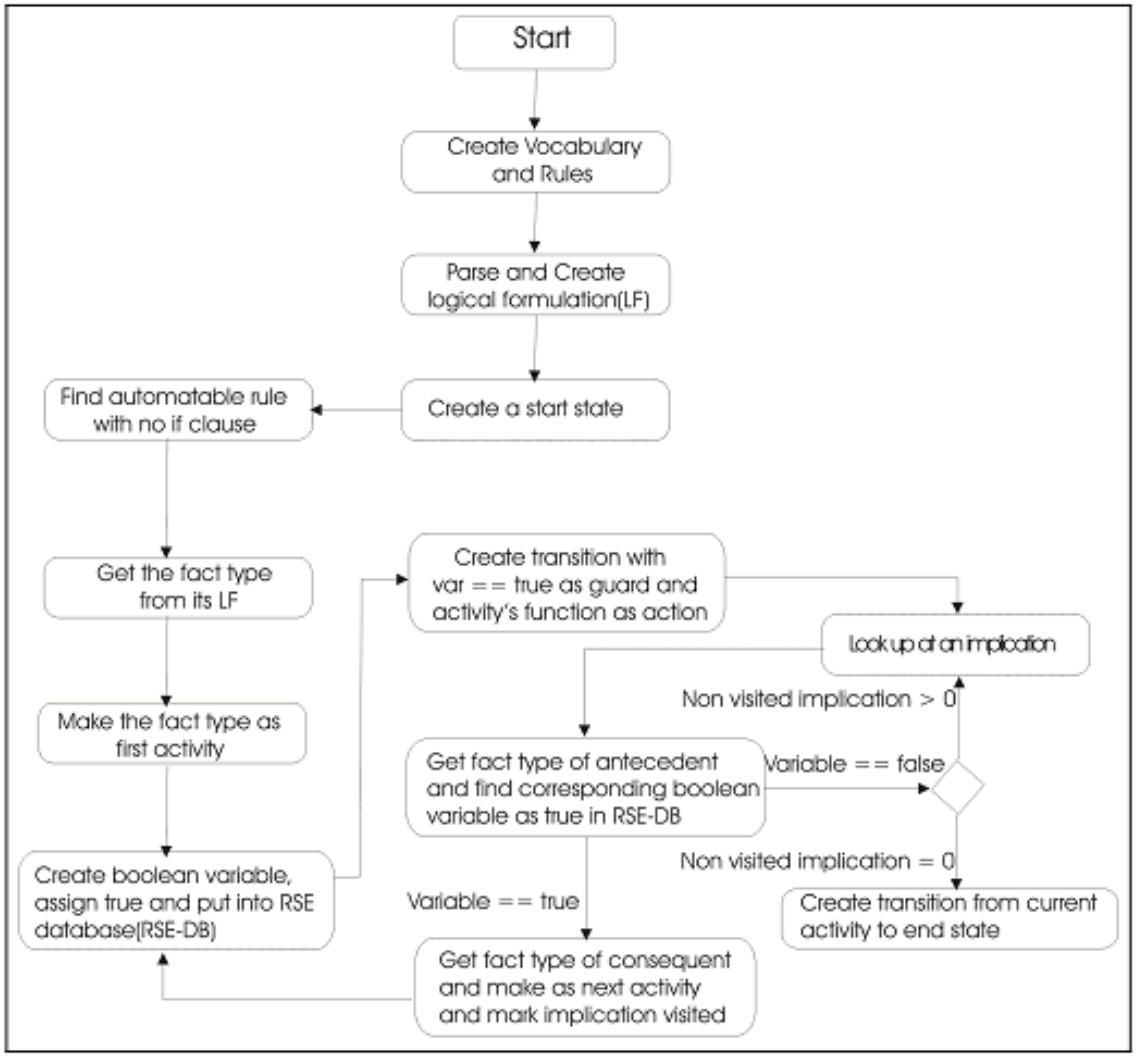


Figure 3.19: Flow Chart of SBVR to UML Activity Diagram Transformation.

be mapped to messages in SD as both of them represent the occurrence of an action. Similarly, the sequence between the activities of SD must be the same as the sequence between activities in AD, as both of them must be consistent with system's behaviour. So, the ordering of activities in AD can be directly mapped to the ordering of messages in SD. Below is a more detailed discussion of UML SD components.

Message A sequence diagram shows the interaction between the objects through messages. These messages are responsible for the occurrence of events and actions in the sequence diagram. Due to its logical similarity with the activities of AD, they have been given the same name as of the activities. For example, name of the activity 'atm ejects card' in the AD will be the message in SD. The message toward the life line of an object shows the event on the object while the messages away from the object are considered as the actions of the object.

Message End Message ends are the locations in life line of objects between which a message is transferred. Logically, it represents what can occur at the ends of message. At one end, it can receive the events and at other end, it can send the events. It is not necessary that the message ends should be located at different objects; it may be on the same object (life line) which will be the case of self message. Sending and receiving of the events is optional.

State Invariant An object in its life time passes through several states. These states are the different configuration of variables of the object. For example, the object 'atm' has the variables 'card' and 'password' of type string and 'card=null and password=null' represents a state of 'atm'. The state change of the object occurs due to the action and events of the object. For example, the action 'user inserts card' will change the value of card to 'card=c' and results in a new state (new configuration) of atm 'card=c and password=null'.

The invariants to the state can be attached in many ways as in plain English or Object Constraint Language (OCL) [38]. It depends upon the user and his requirements, how to attach them.

Life Line (Object) We can assume the object of an activity as the 'giver' of an activity. The semantics of 'giver' is same as the 'subject' in an English sentence having an active verb. The subject is the noun who performs the action of the verb.,

In SBVR's structured English, a sentence having an active, transitive verb, the giver of the action of the verb is the object filling the role of the first placeholder [33] in the fact type form e.g. 'customer' in 'customer places order'. If the fact type form is passive e.g. 'order is placed by customer', then the 'giver, would be the object filling the role of last placeholder. The above two sentences have the same meaning and must be presented as the synonym of each other by a Business Analyst (BA) during the development of Vocabulary. However, it is preferable to use the active form wherever possible. The above two fact type forms would have the same logical formulation or they are logically equivalent.

General Ordering The general ordering in UML SD Metamodel is a partial ordering between the two messages. So, we will look up only at the two activities in UML AD generated above and map their ordering to the ordering of messages.

Algorithm

The generation of UML SD involves the collaboration of both UML AD and SBVR Metamodel. The flow chart for this transformation is shown in Figure 3.20. First of all, the algorithm tries to find out the messages of the SD. According to [31] only activities of the AD can be transformed to messages of the SD. Hence, we will set all the activities of AD as the messages of SD. The first activity in AD will be mapped to the initial message of SD. And the action corresponding to this initial message will be the same as the action corresponding to that activity in AD. This action will become the send event of the destination message end. The other end of the message will not have any associated event. The next thing is to find out the life lines for both source and destination life lines.

The name of the messages is the name of activities which in turn is the fact types, as discussed in Section 3.5.1. In SBVR metamodel, each fact type has some roles which are situated at some placeholder. The source life line of a message will be the term at first placeholder; if the fact type is in active form else it will be the term at last placeholder. To find out the destination life line, we will look up at the next activity in AD. We will get the next activity from the general ordering. The source of the next activity will be the destination life line of current message. This whole process is repeated until we wouldn't traverse all the activities in AD and reach the end state from all the possible paths.

There may be some consecutive activities in AD whose sources are the same.

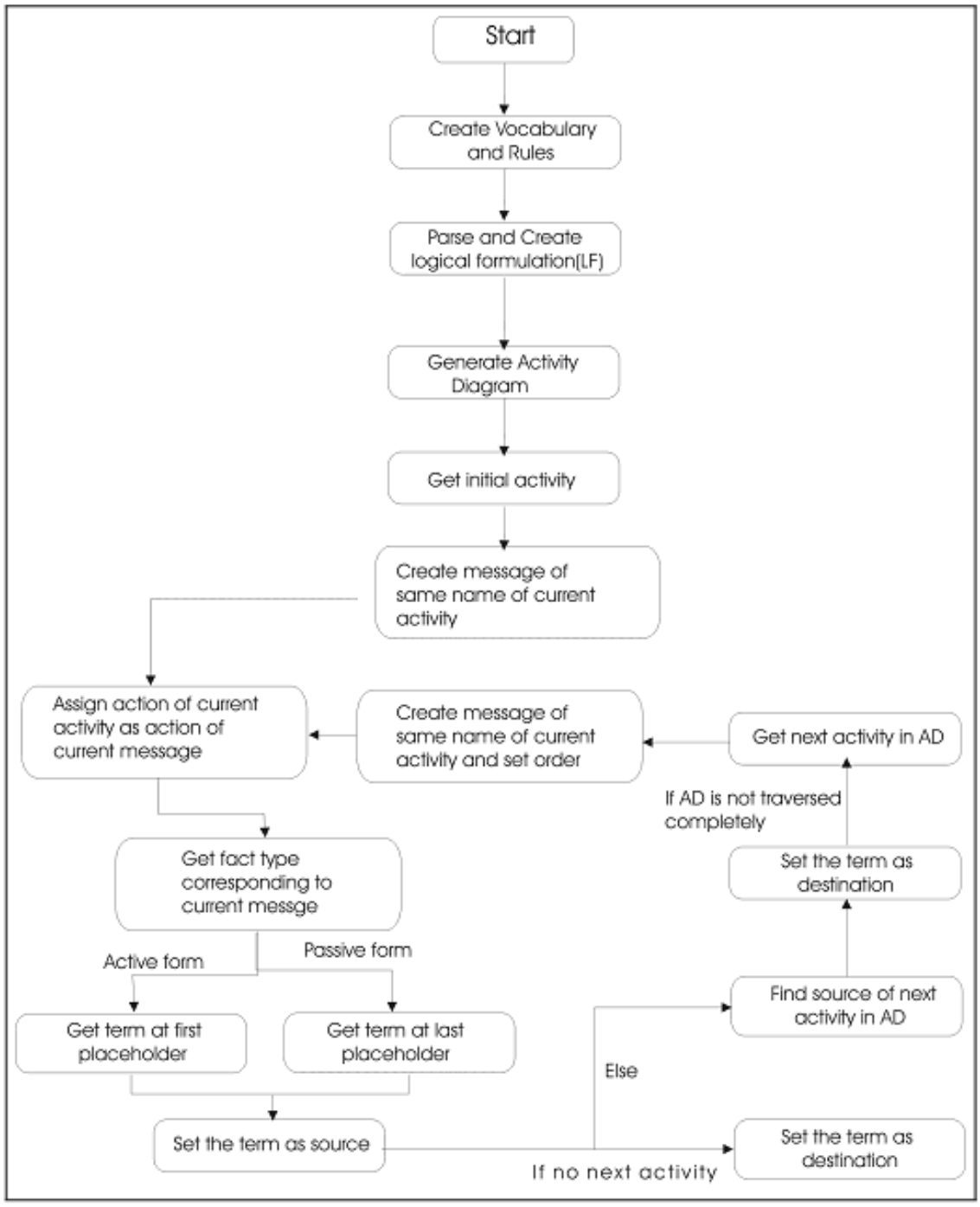


Figure 3.20: Flow Chart of SBVR to UML Sequence Diagram Transformation.

For example the consecutive activities 'atm print receipt' and 'atm ejects card' have the same source 'atm'. In this case, the message corresponding to the activity 'atm print receipt' will be a self message. A self message is a message whose source and destination are the same. Due to this self message, the active object after the occurrence of this message will remain 'atm'. The next message 'atm eject card' will be sent from the object 'atm'.

It may also happen that the activity next to the current activity doesn't exist in an AD. For instance, the activity immediate previous to the end state. This activity doesn't have any next activity. In this case, we may have to compromise as there is no information for the next object. If we see the messages, they are transferring the control to the next destination object. And if there is no information about, who will be the next active object to take over the control, we have to keep the control to the current object. Due to which, the destination life line for this message will be the same as source life line. This is a limitation of this approach and will be recovered in future work. In this approach, we are missing the detection of actors. As our intention is to map the SBVR Metamodel to UML SD metamodel not the UML SD syntax, this is not important at the moment as UML SD Metamodel doesn't include any entity like actors.

Conflict Resolution

The conflict resolution technique is developed by Jon et.,al.,[39]. He suggested that a sequence diagram may have some ambiguities which can be resolved automatically. According to Jon, there are two kinds of constraints of the SD, one is the constraint expressed by the OCL specification and second one is the constraint on the ordering of messages. The conflict between these two constraints means the intended behaviour and the present behaviour of the system is not same. The conflicts can be removed by checking the pre and post condition of messages. For example, pre1 and post1 are the pre and post conditions of message m1 and pre2 and post2 are of message m2. In the sequence diagram, m2 occurs next to the m1. The conflict arises when post1 is not equal to the pre2. The reason may be the domain expert has written the constraints incorrectly or the generated sequence of the messages is incorrect. These conflicts can be resolved automatically by using 'Unification' and 'Frame Axioms' given in [39].

3.5.3 SBVR to UML Class Diagram

In the previous sections, we have discussed the methods of finding the user requirements and determining what objects would be needed to fulfill them. Now, we will discuss the system's static structure design which can be easily visualised by the UML Class Diagram (CD) [40] and how it can be realised from SBVR specification.

3.5.4 SBVR to UML Class Diagram Mapping Rules

The SBVR business rules are declarative in nature. The business vocabulary is used to create the structure of system and business rules put constraints. The structure of the system mainly involves the business objects and their attributes like 'atm' is an object which is a SBVR term and 'atm has card-reader' is a SBVR fact type which shows that the object 'atm' has an attribute 'card-reader'. The business rules determine the correct value of the properties that an object will have and methods of deriving the information needed by the class. Appendix H in SBVR Specification [33] gives a mapping from the SBVR Vocabulary and Rules to the CD. Some of the important rules of the mapping are described here.

Classes

- Class Name: The primary term for a concept that is not a role, individual concept and fact type is shown as class (rectangle). E.g 'Country' would be a class.
- Instance Name: The Name [37] given to an individual concept is shown as an instance specification. The name is followed by a colon and then by the term for its general concept [37]. E.g. The Name 'SBI' is an instance of the class 'bank'.

Attributes For the binary fact types [37] using 'has' as the conjunction between the terms, the second term will be represented as an attribute of the class. For example, in the binary fact type 'atm has card-reader', the class 'atm' will have an attribute 'card-reader' which is of type 'machine'. Ordinarily a unary fact type is transformed into a UML Boolean attribute. E.g in the unary fact type 'atm is open', the class 'atm' will have a boolean attribute 'is_open'.

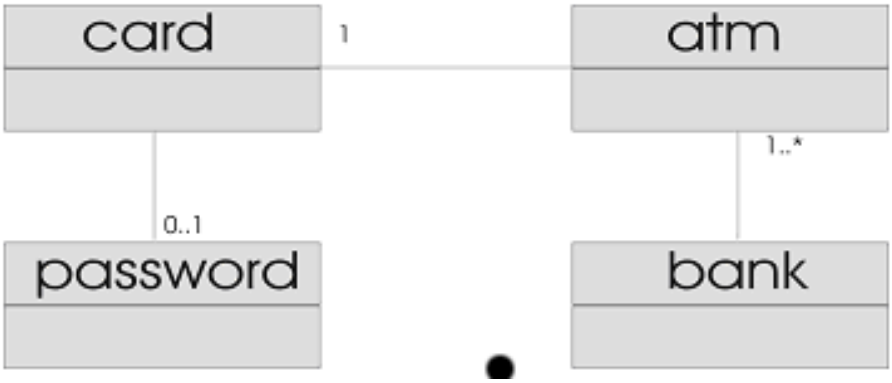


Figure 3.21: Class Structure with their multiplicities.

Functions The above analysis helps us to construct the CD but does not give any information about functions of classes. Before we draw the class diagram, we should know which function will belong to which class. To find out this information, we will follow the same approach as we did in generating the SD in Section 1.2. In a SD, one object interacts with another by sending it a message. For instance, the atm will send the message 'atm requests password' to the 'user' object. This message prompts 'user' to enter the password of card. In the sequence diagram, this message is associated with an action 'request_password()', as we have already shown in Section 3.5.1. Jon Whittle et al., in his study [39] has given some hints about the transformation of SD to CD. By referring [39], we can conclude that the action corresponding to the messages will become a function for the source object in the SD that is 'atm'. The work is not able to determine the return type and arguments of these functions at the moment. We will recommend that the user sets these entities manually. This problem will be handled in the future.

Multiplicity Multiplicity is not referred to properly in [37]. So it is referred from [36]. Quantification in SBVR MetaModel is used to define the multiplicities between the classes. E.g. (see Figure 3.21).

```
It is necessary that each card uses at most one password.
It is necessary that each atm accepts exactly one card.
It is necessary that each bank provides at least one atm.
```

3.5.5 Limitations and Future Work

The SBVR specification is quite large and fairly complex. Additionally, very little published work is available. So, capturing all the semantics of SBVR is quite complex. This work is not able to capture modal formulations [33] permissibility of action) of the rules. Rather than capturing the modality of rules in UML models, we are assuming that all the activities have the highest severity of action. There are some more limitations of our approach for instance we are not able to find out the parameters of the functions of classes and return type of functions. In sequence diagram, we are not able to determine the target object of last message. To construct the AD, we are taking only those rules in account which are in 'if-then' construct where as automatable rules can be found in some more constructs. The SD developed by this technology doesn't give the annotated SD [39]. The future work involves the minimization of these limitations and also includes the following tasks:

- Petri Selonen et al., in [31] shows which UML diagrams are completely interchangeable and which are only supported. The generation of other UML concepts from SBVR like collaboration diagram, state chart diagram etc. will be in priority. This transformation among UML diagrams helps us to minimize the inconsistencies before converting these PIMs to PSMs.
- Generation of PSMs and code base.
- Enabling to work with other languages apart from English.
- Emphasize on capturing of the rules in OCL [38] but we may shift to some other platform as OCL doesn't support the modality of rules

3.5.6 Related Work

This work brings us to a very important outcome which allows the business people to work independently of IT to build up their system prototype. Originally, the SBVR was developed only to communicate or transport the business semantics between the two communities which may be of the same or different language. This research adds one more application area to the SBVR enabling it to generate PIM Models too. The following studies are relevant:

Linehan [37] explores the work to specify the semantics and rules in SBVR as extension of business models that are automatically translated to PIMs which in

turn get converted to PSMs. This technology is known as Model Driven Business Transformation (MDBT). Here, the PIM model include the UML Class diagram, State Chart and Use Case Diagram. But the paper doesn't explicitly specify the algorithms and there is no such information of how to find the function of the classes too.

Schacher [32] explores the work to view the Business rules in the perspective of SBVR and CASSANDRA/xUML [36]. This paper develops an environment completely based on the CASSANDRA platform to create executable UML Models (also called xUML Models) from the Business Rules. It also provides a rule-set to transform SBVR Vocabulary and Rules into Class Diagram, Use Case Diagram based on xUML platform. This paper also shows how the business activities represented in BPMN can be transformed to xUML Notation.

Sorensen et al., in his study [41] shows the lack of ontology in the SBVR Meta-Model and also shows how the ontology integration into SBVR could improve the future releases of this standard. The paper describes that a pattern can be thought of as an arrangement of objects. To create a pattern, there must be a similarity and contrast between arrangements of objects as measured by the proximity metric. This measurement will decide which object should be included or excluded.

SBeaVer [9] is an open source SBVR tool created by Maurizio De Tommasi and Pierpaolo Cira at the University of Lecce in Italy, in a project funded by the DBE[5] project. This tool runs as a plug-in for the Eclipse platform which enables the user to create, validate and verify the business vocabulary and rules.

3.5.7 Conclusion

This work enlightens the transformation of the business rules into PIM models. It starts with an introduction about the definition of SBVR and its metamodel and its transformation to UML models. Hans-Erik Erikssonet et al., in [42] and Markus Schacher in [32] shows the support of UML for the Business Models, which in turn gives an intuition for the business rule transformation to UML Models. These models play very crucial role in further development and also provide a bridge between business requirements and IT system. This bridge enables the BA to control the system development.

Software models can be easily represented as UML Diagrams. UML Diagrams are a visual aid to modelling software graphically. In this work, we have only discussed

[5]http://www.digital-ecosystem.org/

the UML activity diagram, sequence diagram and class diagram. We have also studied how they are interrelated with each other and how other UML diagrams can be generated from AD, SD and CD. BPMN and UML are presently known as one of the best platforms to represent a workflow diagrams. Several companies have adopted these notations according to their needs.

SBVR is declarative in nature and needs to be transformed into the imperative form, to show the behaviour embedded in the SBVR vocabulary and rules. It includes the categorization of automatable and non-automatable rules. We are using only automatable rules in order to get a fully automated workflow. A mapping set from SBVR construct to UML Activity diagram has been presented which includes the ordering of rules using RSE.

A UML Sequence diagram is another aspect in which the system requirements are represented with the help of messages between the objects. The names of activities in AD have been considered as the name of messages and giver of the action embedded in fact types are considered as the source objects. The source object of next message will be the target object of current message. This whole process may introduce some conflicts. Section 3.5.2 briefly discusses how to remove them.

SBVR specification already discusses a mapping from SBVR to UML Class diagram but that mapping set has some deficiencies. What those deficiencies are and an approach to overcome them have been discussed briefly.

The main idea of this work is to separate the business rules from IT and the actual code, by converting the business rules and business vocabulary into AD, SD and CD, which again can be transformed to other UML diagrams [31] to check any inconsistency between intended behaviour and actual behaviour of the system [39], [43].

4 Conclusion and Outlook

This deliverable provides a wide range of concepts related to communication, interaction and expressiveness in Digital Ecosystems. Language is the fundamental basis for every kind of ecosystem where individuals live together. Although this deliverable focuses on a more technical viewpoint regarding expression and language, it gives a considerably good overview of the code-structure and workflow generation perspectives which can be found in a Digital Ecosystem. As SBVR started out as the descriptive language of choice in the DBE project, it was now analysed in more detail from different perspectives. Figure 4.1 provides a quick overview about the different perspectives to SBVR:

1. Linguistics perspective
2. Workflow perspective
3. Code generation perspective
4. Execution perspective
5. DBE and BML perspective
6. SME perspective

While the latter two perspectives could not be studied due to the lack of time, the first four were examined in OPAALS Workpackage 2. The goal of this heterogeneous analysis was to find out where SBVR can help as a descriptive language in a Digital Ecosystem and where other approaches have to be found. The following Sections provide not only a summary of the individual points of view, but also conclusions as a consequence of one and a half years of work in this very interdisciplinary workpackage. Finally this deliverable concludes with an outlook to the second phase of OPAALS.

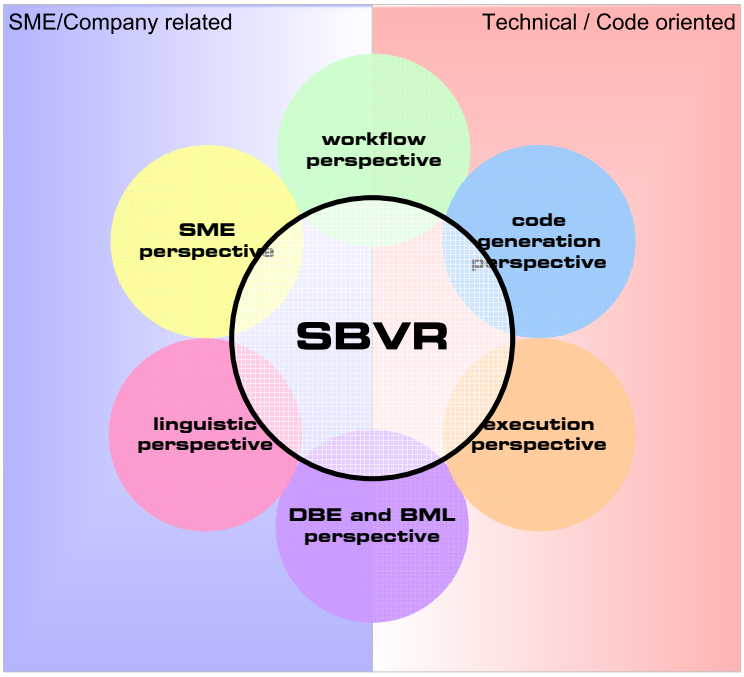


Figure 4.1: Six different perspectives to Workpackage 2.

4.1 Linguistics perspective

The foundation for natural language processing with respect to code structure generation was discussed in Chapter 2. As a major conclusion of this section an algorithm for transforming specifications authored in natural language was introduced, which has not been fully implemented, yet. The practicability of this algorithm therefore has to be examined to prove the correctness of the outcome. Natural language has several advantages compared to SBVR in terms of usability for example. As the *Digital* requires more structure and unambiguity for generating executable XPDL workflows, the *Ecosystem* needs to be human understandable and as natural as it can be. Therefore, a way has to be found to combine NL and SBVR or at least find ways to easily convert one language into the other. A first step for this is given in this deliverable.

4.2 Code (structure) generation perspective

As a step towards formulating software structure and even behavior in natural language, experiments with SBVR as a meta-model was outlined. Possible applications

of SBVR were discussed in Chapter 3, where the practicability of user interface generation on the one hand and the workflow extraction out of business process information was shown in a prototypical manner.

The structures of the business entities were implemented in the Grails prototype. Together with the Grails framework this code structure resulted in a database backed application. Although the transformation from SBVR specs to Grails code already was implemented using the P2P Servent Infrastructure, we decided to implement a *distributed runtime* for the generated applications too. This means that the Grails runtime, together with a web container, was bundled into a Servent service, in order to remotely deploy a generated application somewhere in the peer-to-peer ecosystem.

Furthermore, model transformations to code structure were introduced in Chapter 3 in a distributed way, through the usage of Servent technology [19] as the foundation of the prototypes. One step forward will be to not only to do the generative steps in a distributed way but also to store and retrieve the processed information using a distributed repository. Possible details of such a repository were introduced in Section B.2. Future work as regards the repository need to be considered carefully as all implementations have to be aligned with the outcome of Task T10.15 which is responsible for a distributed repository for the Open Knowledge Space.

4.3 Workflow perspective

The workflow generation examples in Section 3.4 and Section 3.5 show the capabilities and shortcomings of SBVR for describing and consequently executing workflows. From these first implementations follows that the transformation of SBVR statements is possible in principle but seems to make practical sense in just a few cases. There are two very promising approaches for workflow generation out of SBVR introduced in this deliverable. Firstly the transformation to XPDL and BPMN and secondly the transformation to UML Activity diagramms. As BPMN is technically seen very close to BPEL, the toolset can be utilised for producing BPEL as well in future. The XPDL approach for workflows will be followed up in Task T2.1 in Phase II of the OPAALS project. Additionally, considering one workflow as such, the individual workflow parts could be enriched with a dynamic lookup and automatic or semi-automatic service composition. That could be worked out with a possible applicability of the REST paradigm for dynamic service composition (Section B.1).

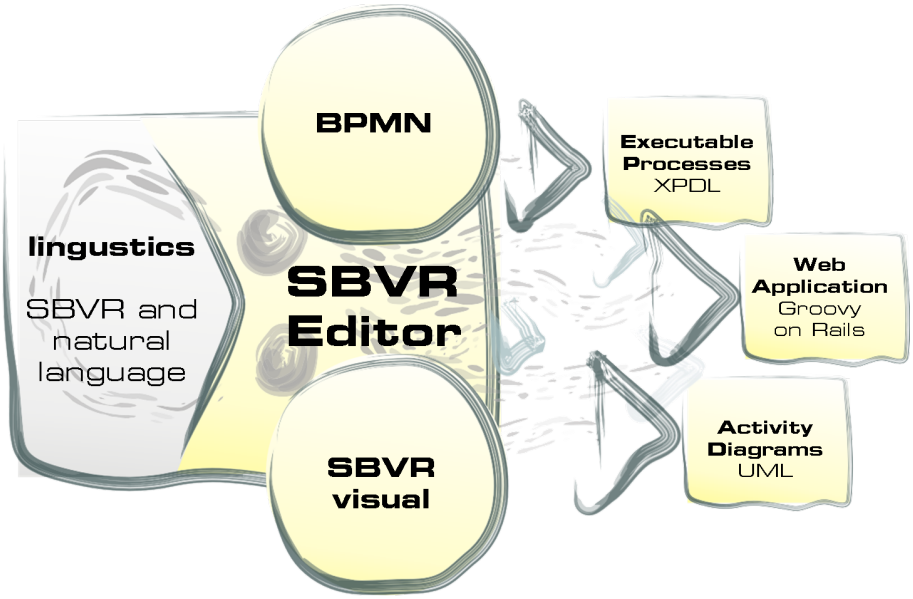


Figure 4.2: SEPIAX basic project structure.

4.4 Execution perspective

In principle, the execution of a set of SBVR specifications can not be done without additional human interaction. As shown in the previous sections, the transformation can result in a simple web-application or a workflow which can be executed, given that there are predefined components with compatible interfaces. The main issue here is also the lack of open source tools for SBVR, which is a very new OMG standard. In phase II work is planned to extend the simple examples of automatic workflow generation and execution as well as in direction of dynamic service composition.

4.5 Future development and SEPIAX

Deliverable 2.2 was intended to give a broad and multidisciplinary view on the usage of SBVR within a Digital Ecosystem. The interdisciplinary and multicultural nature of the SBVR working group was as challenging as it was interesting. Moreover, the interaction within the group and the building of weekly chat sessions as well as the convergence of the vocabulary used during the interaction of partners is also a major outcome of this work package.

The different perspectives and the analysis and prototypical implementation of various use cases led to a more focused work plan for phase II. Beside the extension of the use cases in the area of XPDL and the extension of the editor and parser components, a repository as well as a web-based user interface are planned in order to involve the DE community in the feedback and evaluation process of the upcoming tools.

In order to support the technological catalysts, tools that can be used to describe business processes in the SBVR formalism are currently under development. Some of these tools are embedded in the up-coming Semantic Ecosystems for Process Identification, Automation and eXtraction (SEPIAX) project. The goal of SEPIAX is to provide a combination of tools to create, manage and use business models in a Digital Ecosystem.

Figure 4.2 summarizes the efforts and components of SEPIAX so far. Starting from the SBVR Editor (see SBeaVeR in Appendix A) the three transformation services for 1) Executable Processes in XPDL, 2) Web Applications with Groovy on Rails and 3) UML activity diagrams will be integrated. The first two of them are already integrated and work with the SBVR editor as servENT services. The source code of the UML activity diagram service is also in the SEPIAX repository but bases on another editor component at the moment. For the visualisation of SBVR and the workflows we can think of two visualisation components. One for visualizing XPDL workflows with the Business Process Modelling Notation (BPMN) and one for a visualisation based editing of SBVR statements. These components are not planned or implemented yet. Of a high importance is the connection to linguistics in the SEPIAX activities. For the web-based editor component for phase II for example, it is also a extension of the collaboration with WP6 and the linguistic group planned.

Bibliography

[1] A. Farghaly. *Handbook for Language Engineers*. CSLI Publications, Stanford, 2003.

[2] V. Fromkin, R. Rodman, and N. Hyams. *An Introduction to Language*. Thomson, Boston, 2003.

[3] D. Jurafsky and J. H. Martin. *Speech and Language Processing: Introduction to Natural Language Processing*. Prentice Hall, 2001.

[4] D. Efftinge, P. Friese, A. Haase, C. Kadura, B. Kolb, D. Moroff, K. Thoms, and M. Voelter. *openArchitectureWare User Guide*. 2007. `http://www.eclipse.org/gmt/oaw/doc/4.2/openArchitectureWare-42-reference.pdf`. Last accessed on 12/04/2007.

[5] Grails Community. Groovy on Rails Framework. Webpage. `http://grails.codehaus.org`. Last accessed on 11/14/2007.

[6] Fachhochschule Salzburg GmbH. *Semantic Ecosystem for Process Identification, Automation and eXtraction (Homepage)*, 2007. `http://www.sepiax.org`.

[7] Fachhochschule Salzburg GmbH. *Semantic Ecosystem for Process Identification, Automation and eXtraction*, 2007. `http://sourceforge.net/projects/sepiax`.

[8] Fachhochschule Salzburg GmbH and Languages and Models community. *Languages and Models. Chat logs of the SBVR related chats in OPAALS*, 2007. `http://wiki.opaals.org/WP2_Automatic_Code_Generation_From_Models/Language_and_Models`.

[9] M. De Tommasi and P. Cira. *SBeaVeR*, 2006.

[10] S. Duncan. *The MIT Encyclopedia of the Cognitive Sciences*, chapter Language and Communication, pages 438–440. MIT Press, 2001.

[11] G. Chierchia. *The MIT Encyclopedia of the Cognitive Sciences*, chapter Linguistics and language, pages xci–cix. MIT Press, 2001.

[12] B. Katz, G. Borchardt, and S. Felshin. Syntactic and Semantic Decomposition Strategies for Question Answering from Multiple Resources. In *Proceedings of the AAAI 2005 Workshop on Inference for Textual Question Answering*, pages 35–41, Pittsburgh, July 2005.

[13] B. Katz and B. Levin. Exploiting Lexical Regularities in Designing Natural Language Systems. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING '88)*, 1988.

[14] A. D. Friederici. The time course of syntactic activation during language processing: a model based on neuropsychological and neurophysiologial data. *Brain and Language*, 50:259–281, 1995.

[15] A. D. Friederici. Towards a neural basis of auditory sentence processing. *Trends in Cognitive Science*, 6(2):78–84, 2002.

[16] D. Klein and D. Christopher. Accurate Unlexicalized Parsing. In *Proceedings of the 41st Meeting of the Association for Computational Linguistic*, pages 423–430, 2003.

[17] R. Eder, T. Kurz, T. J. Heistracher, V. Bayon, M. Russo, and A. Filieri. D2.1 - Design of Software Generation Prototype. OPAALS Project, October 2007.

[18] OMG. *MDA Guide*, June 2003. Version 1.0.1, `http://www.omg.org/docs/omg/03-06-01.pdf`. Last accessed on 03/02/2007.

[19] DBE Consortium. Swallow - DBE Servent. Webpage. `http://swallow.sourceforge.net/`. Last accessed on 11/14/2007.

[20] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[21] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*, January 2006. Version 2.0, `http://www.omg.org/docs/formal/06-01-01.pdf`. Last accessed on 04/03/2007.

[22] OMG. *Unified Modelling Language Specification*, 2003.

[23] Groovy Community. Groovy - An agile dynamic language for the Java Platform. Webpage. `http://groovy.codehaus.org/`. Last accessed on 11/14/2007.

[24] Java Community Process. JSR 241: The Groovy Programming Language. Webpage, 2004. `http://jcp.org/en/jsr/detail?id=241`. Last accessed on 11/14/2007.

[25] Apache Software Foundation. Apache Maven Project. Webpage. `http://maven.apache.org/`. Last accessed on 12/11/2007.

[26] Eclipse Foundation. About the Eclipse Foundation. Webpage. `http://www.eclipse.org/org/`. Last accessed on 11/14/2007.

[27] Inc. Object Technology International. *Eclipse Platform Technical Overview*. 2003. `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`. Last accessed on 12/04/2007.

[28] Eclipse Foundation. Eclipse Modeling Framework Project. Webpage. `http://www.eclipse.org/modeling/emf/`. Last accessed on 11/14/2007.

[29] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[30] Sun Microsystems. Trail: The Reflection API. Tutorial. `http://java.sun.com/docs/books/tutorial/reflect/index.html`. Last accessed on 11/14/2007.

[31] P. Selonen and K. Koskimies. Transformation between uml diagrams. *Journal Of Database Management*, 14:37, 2003.

[32] M. Schacher. *Moving from Zachman Row 2 to Zachman Row 3*, 2006.

[33] OMG. *Semantics of Business Vocabulary and Business Rules Specification (SBVR)*, March 2006. Interim Convenience Document, `http://www.omg.org/docs/dtc/06-03-02.pdf`. Last accessed on 04/05/2007.

[34] OMG. *Business Motivation Model Specification*, 2006.

[35] OMG. *Production Rule Representation Specification*, 2003.

[36] Know Gravity. *CASSANDRA/xUML*, December 2007.

[37] M. H. Linehan. Semantics in model-driven business design. In *Models/UML Conference*, 2001.

[38] OMG. *UML 2.0 OCL Specification*, 2003.

[39] J. Whittle and J. Schumann. Generating statechart designs from scenarios. pages 314–323, 2000.

[40] OMG. *UML MetaModel*, 2005.

[41] D. Sorensen, A. Pastiak, A. Mitra, and A. Gupta. Integrating ontology into sbvr. *Eller College of Management Working Paper*, No. 1033-06, 2006.

[42] H.-K. Eriksson and M. Penker. *Business modeling with UML*. John Wiley and Sons, Inc, 2000.

[43] A. Gupta and A. Raj. Strengthening method contracts for objects. In *Asia Pacific Software Engineering Conference*, 2006.

[44] D. Hay and K.A. Healy. Defining business rules-what are they really. Technical report, The Guide Business Rules Project, 1996. Last accessed: 12/17/2007.

[45] R. G. Ross. *The Business Rules Manifesto*. Business Rules Group, 2003.

[46] C.J. Date. *What Not How: The Business Rules Approach to Application Development*. Addison-Wesley Professional, 2000.

[47] D. Thomas and D.H. Hansson. *Agile Web Development with Rails*. Raleigh, Pragmatic Bookshelf, 2006.

[48] S.J. Mellor and M.J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, 2002.

[49] B. Lucas and C. F. Wiecha. Collage: A Declarative Programming Model for Compositional Development and Evolution of Cross-Organizational Applications. In *W3C Workshop on Declarative Models of Distributed Web Applications*, 2007.

[50] S. Hendryx. *Model-Driven Architecture and the Semantics of Business Vocabulary and Business Rules*. Hendryx & Associates, 2005.

[51] Google. Google Data APIs (Beta) Developer's Guide. Technical report. Last Accessed: 25/10/2007.

[52] Amazon. Amazon Simple Storage Service (Amazon S3). Technical report, 2007. Last Accessed: 25/10/2007.

[53] P. Castro. Overview: Microsoft Codename Astoria. Technical report, Microsoft Corporation, 2007. Last Accessed: 25/10/2007.

[54] Microsoft Corporation. Web Structured, SchemaŠd & Searchable (Web3S). Technical report, 2007. Last Accessed: 25/10/2007.

[55] J. Gregorio and B. de Hora. The Atom Publishing Protocol. Technical report, The Internet Engineering Task Force, 2007. Last Accessed: 25/10/2007.

[56] Y. Goland, E. Whitehead, A. Faizi, and D. Jensen. RFC 2518: HTTP Extensions for Distributed Authoring – WEBDAV. Technical report, Internet Society (ISOC), 1999.

[57] R.T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.

[58] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, Inc, 2007.

[59] R.Fielding, J.Gettys, J.Mogul, H.Frystyk, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol–HTTP/1.1. Technical report, Internet Society (ISOC), 1999.

[60] U. Kuster, M. Stern, and B. Konig-Ries. A Classification of Issues and Approaches in Automatic Service Composition. In *Intl. Workshop WESC*, volume 5, 2005.

[61] M.J. Hadley. Web Application Description Language (WADL). Last accessed 12/19/2007.

[62] R. Chinnici, J. Moreau, C. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Technical report, W3C, 2007. Last accessed: 12/19/2007.

[63] D. Obasanjo. Google Base Data API vs. Astoria: Two Approaches to SQL-like Queries in a RESTful Protocol. Technical report, 2007. Last Accessed: 25/10/2007.

[64] J. Yang, M.P. Papazoglou, and W.J. van den Heuve. Tackling the challenges of service composition in e-marketplaces. In *In Proceedings: Research Issues in Data Engineering: Engineering E-Commerce/E-Business Systems, Twelfth International Workshop*, 2002.

A Introduction to SBeaVeR [UCE]

The SBeaVeR is an integrated tool that allows business analysts to create high level business models through the SBVR Structured English formalism.
It is a text-based tool that allows the modeller to create a description of a generic business model by typing structured sentences and business rules through an easy to use graphical interface. The editor helps the modeller in the process of creating a business model in a computation-independent fashion, avoiding technical modelling formalisms typically based on the object oriented modelling paradigm as used by IT system designers and technical people.
This section focuses on the functional aspects of the tool and shows its main use cases, as in figure A.1.

A.1 Use case: "Create Vocabulary"

In this scenario the user defines the SBVR vocabulary needed to describe the elements of a business model and their semantic relationships.

A.1.1 Define term

The following use case defines the creation of a vocabulary entry with specific focus on the definition of a term representing a concept involved in the domain of the modelled business (e.g. system, user , employee, bank account, order). The user can put in the editing area each entry. Each entry includes the term itself and optionally other information, according to the SBVR meta-model, labelled by captions like *Definition*, *Source*, *General Concept*, etc.

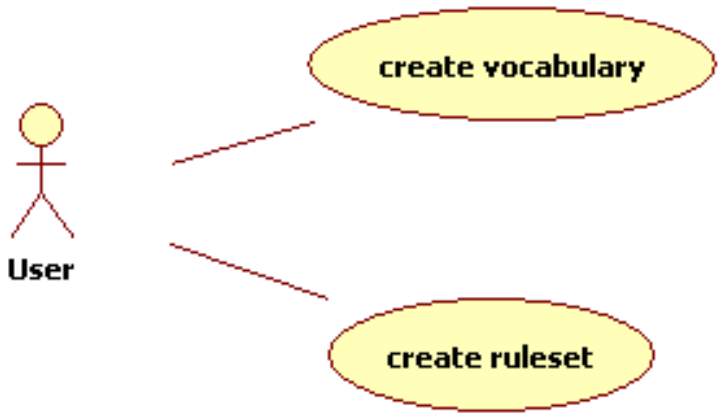


Figure A.1: Main use case of the SBeaVeR.

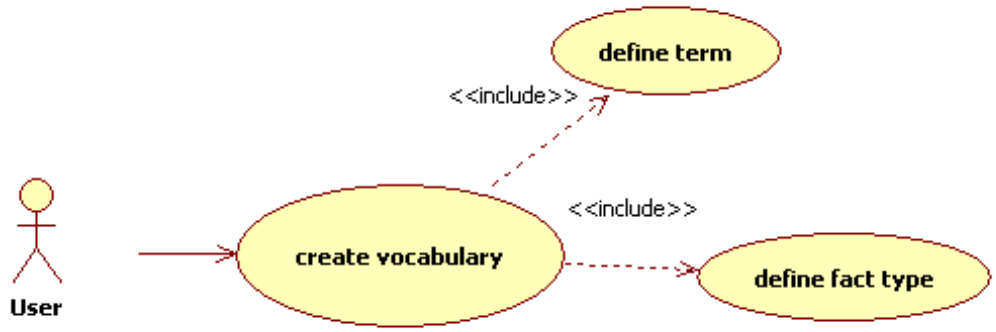


Figure A.2: Vocabulary Definition Use Case.

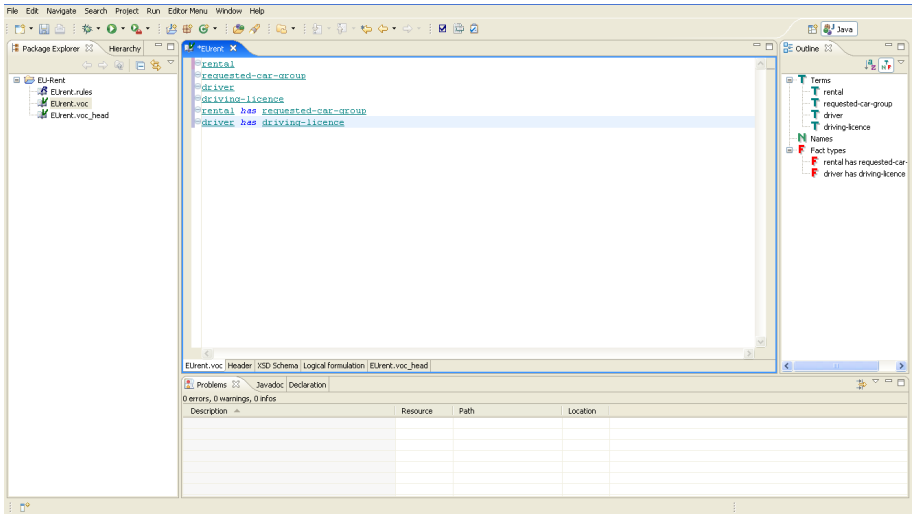


Figure A.3: The vocabulary editor of the SBeaVeR

A.1.2 Define Fact type

In this use case the user defines the fact types needed to identify semantic relationships (binary fact types) among and characteristics (unary fact types) of the terms already defined (e.g.user **inserts** card, system **verifies** card, card **is valid**). The user can put in the editing area each entry. Each entry includes the fact type itself and optionally other information, according to the SBVR meta-model, labelled by captions *Definition*, *Source*, *General Concept*, etc.

A.2 Use case: "Create Ruleset"

In this scenario the user defines a rule-set based on a formerly described business vocabulary. The user writes the rules in the input area an the text is dynamically formatted accordingly to the SBVR Structured English styles(e.g. `it is necessary that each` rental car **has** `at least one` rental driver.).

A.3 SBeaVeR for Workflow

In order to implement automatic workflow generation from SBVR models, the SBVR editor SBeaVeR has been leveraged and enhanced. In particular one additional use case has been designed (Figure A.5) and developed. Such use case includes the

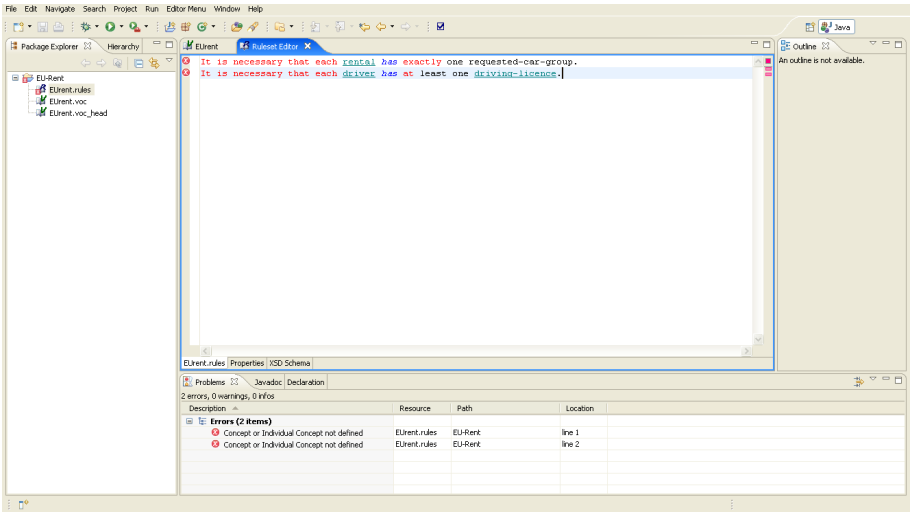


Figure A.4: The ruleset editor of the SBeaVeR

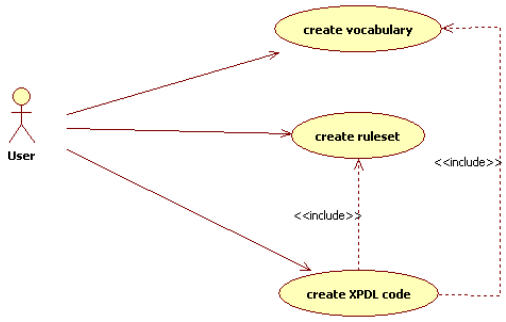


Figure A.5: Main use case of the SBeaVeR enhanced for XPDL generation

existing ones for SBVR vocabulary and ruleset creation and editing, providing an additional feature for automatic transformation of the model.

Such use case has been implemented through a tab in the graphical user interface for ruleset editing. Indeed, after the definition of an SBVR model (vocabulary + ruleset) of a workflow, the enhanced SBeaVeR provides to the user the corresponding XPDL representation in a new tab (see Figure A.6).

The java classes that implement the transformation have been both embedded in the SBeaVeR code for stand-alone use, and packaged as a remote service to be deployed in the DBE Servent for easier future integration in a Digital Business Ecosystem. Such service exposes a method called getXPDL that accepts as input

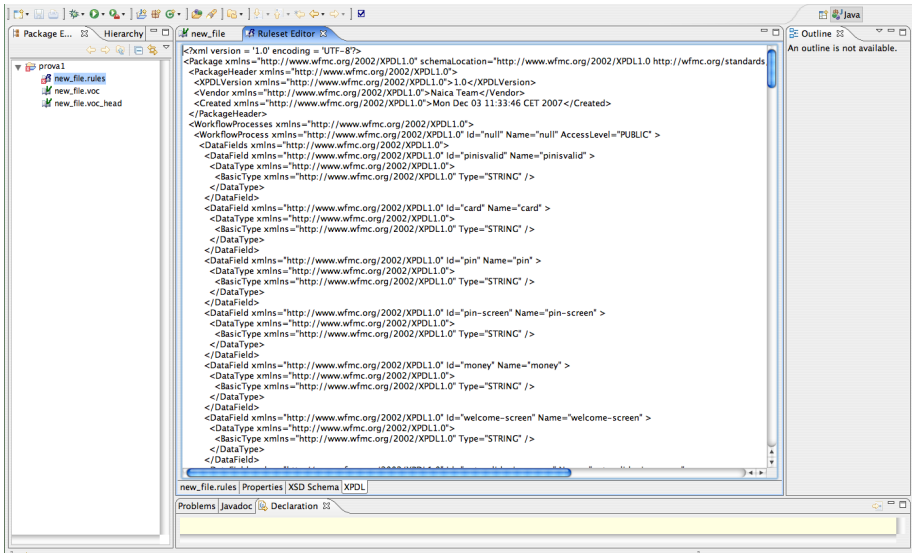


Figure A.6: The tab added to the SBeaVeR for visualizing the generated XPDL

two parameters of type String (the first for the vocabulary and second for the ruleset) and that returns as output a String (the XPDL representation). In both cases the transformation is provided by the following components:

- a parser for vocabulary recognition and interpretation and for consequent instantiation of java classes for in-memory representation of the workflow;
- a parser for ruleset recognition and interpretation and for consequent instantiation of java classes for in-memory representation of the workflow;
- 6 classes for in memory representation of the workflow: Activity, Gateway, Performer, Property, Sequence, WFProcess. Each one of such classes has a method called getXPDL() that provides the XPDL code correspondent to its workflow element.
- A class called Transformer that orchestrates the parsers and the six workflow classes in order to obtain the complete XPDL representation.

B SBVR related initiatives

B.1 Declarative service composition with SBVR [UniS]

The mainstream approach to the creation of information systems for businesses consists of using an imperative programming language to create the entire information system. This includes persistence, the user interface and the various processes that are to be used within the business as well as various algorithms that may be of use to the business. Over time, abstractions have arisen that allow programmers to avoid a lot of repetitive code. Such innovations include standardised persistence frameworks, user interface libraries and recently, business process modelling tools that allow business analysts to specify their business processes and even use those exact process specifications as part of the running system. All these developments however, are incremental improvements that do not address a basic source of problems for information systems. This can be traced to their inherently procedural nature. The specifications for their operation, the actual reasons why specific design decisions have been taken, are not formally specified but instead may exist in design documents of varying quality. Many times decisions are taken by programmers as the need arises, without these decisions being documented or being consistent across the information systems. In general, an imperative approach has no facilities to maintain and enforce business logic consistently across the system.

The approach to information systems advocated in this section attempts to remedy this situation by using a declarative approach to the creation of information systems. This approach should allow the business users to formally specify the business logic at a high level, and have these specifications used directly to produce a working information system. This would allow for the business owners to have complete control over the behaviour of the system and additional decisions will have to be deferred back to them. Additionally, any modifications to the system can should only happen at the specification level and therefore be consistently enforced

throughout the system.

B.1.1 Background

Declarative and Imperative Programming

At the root of the differences between the current and the proposed approach lies the difference between the declarative and the imperative paradigms. In order to understand their differences, it is important to separate the 'what' and the 'how' of a solution to a computing problem. The 'what' refers to the properties that a solution must possess whereas the 'how' refers to the steps followed to achieve the required solution. Declarative programming focuses on specifying the 'what' and depending on a software platform to decide which is the most appropriate way of reaching the goal. An example of a declarative language is the well-known SQL which specifies properties of data but not the way to retrieve it, which is left to the database implementation. The example of SQL as a declarative language and the database as a platform is in many ways related to the approach presented here and will be explored further in later sections.

Imperative programming focuses on the 'how', bypassing the need to define the properties of the required solution since the user can guarantee the desired properties by directly controlling the algorithm. Java is considered an imperative language, although in later versions, declarative elements have appeared either in the language itself or in libraries designed for it.

Evaluation of the imperative paradigm as applied to information systems

Having established that imperative programming is directly related to prescribing the steps required to reach a certain goal, it is justified to characterise process-oriented systems as imperative. In this light, we can examine these systems and draw conclusions about possible limitations of the approach.

Under the current approach, business processes are created allow the actor to execute a request while verifying that the execution of this request will not violate the implicit business logic. While a very large number of use cases could be legally executed based on this business logic, each process focuses on a single use case.

Therefore a business process designer must decide which use cases must have processes designed for them. Usually at this stage, a choice is made between simple use cases which could then be used to carry out more complicated workflows by the

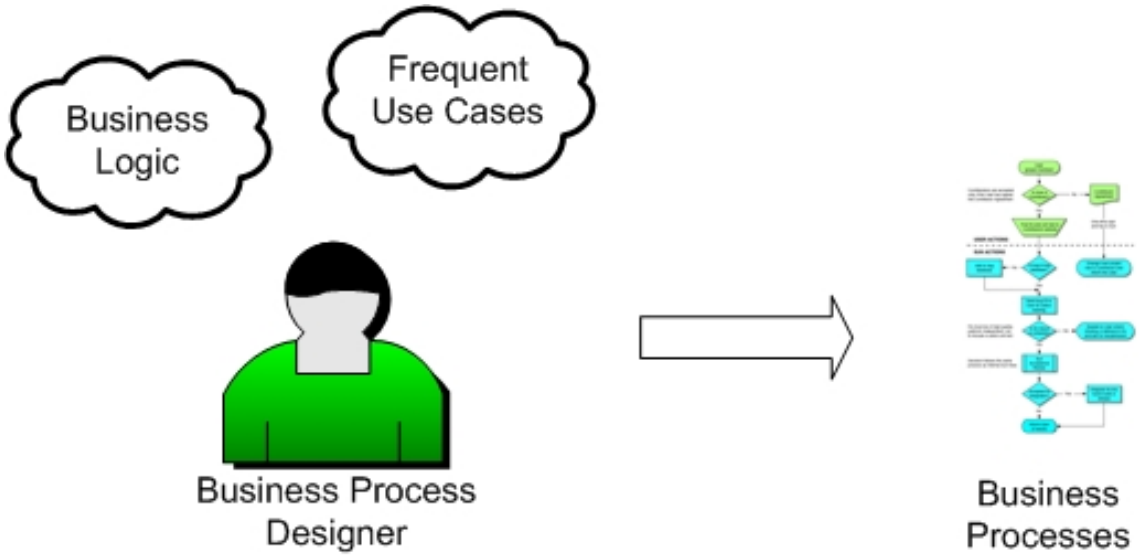


Figure B.1: Business Process design.

user, and more complex use cases that create larger processes, to automate particular recurring workflows within a business. The fact that the designer must select the use cases to create processes for implies that some less frequent use cases are left out. This 'long tail' of individually rare use cases however, when examined as a whole, can account for a significant amount of the system's usage. Additionally, some rare use cases may become more frequent as the business environment changes. In this case the process designer must add new processes, ones that again satisfy the current business logic. However, what usually happens is that the users create ad-hoc combinations of simpler processes to accomplish their new tasks, often repeating steps and performing tasks that belong to the information system, therefore operating with reduced efficiency.

The process is designed in the way that allows it to achieve its user's objectives while maintaining business logic. In order to achieve this, the logic must be hard-coded within the structure of the process. Even if we assume that the process designer creates processes that are perfect in efficiency and correct in business logic enforcement, which is no small task, two more problems become apparent: If a part of the business logic changes, all processes must be examined and the ones affected must be modified. This can be a difficult, error-prone task since the effects of a business logic change are not always apparent. Additionally, possible omissions in altering process logic to correspond to the changes causes inconsistencies in the

system and may even lead to corruption of data, in the case that two or more processes enforce different logic on the same data.

Introduction to Business rules

Before continuing, it would be useful to clarify the usage of the terms 'business logic' and 'process logic'. While business logic refers to the underlying reasons for the behaviour of an information system, process logic refers to the actual sequencing of actions in response to a specific request. It can therefore be said that the business logic determines, or is compiled into the process logic. However it must be noted that the business logic cannot easily be extracted from the process logic. While there are certainly business reasons that determine the control and data flow of a process, it cannot always be determined what those business reasons are simply because different business logic can result in identical processes. In terms of Model-Driven Architecture (MDA), business logic can be said to represent the Computation Independent Model (CIM) or Business Model while process logic more closely relates to the Platform Independent Model.

Business Rules Approach

Currently the term 'business rules' is used by two distinct approaches. One is commonly called Production Rules and focuses to reactions a system should have on certain events or when certain conditions are met. The other approach to business rules has been termed Logic-Based rules or simply 'Business Rules Approach' and is the approach that is described and used in this section. According to this approach, business rules can be considered to be atomic elements of the business logic. One definition of business rules that can be used to better clarify the concept is the following:

> [a] business rule is a specific, formal statement of a single term, fact, derivation, or constraint on the business [44].

Central to the business rules approach is the Business Rules Manifesto [45], written by Ron Ross, widely regarded as 'the father of Business Rules'. This manifesto is also included in the SBVR Specification [33], signalling the intent to harmonize SBVR's use with the principles of the Business Rules Approach. The following excerpts from the Business Rules Manifesto are relevant to this discussion:

> **Article 2. Separate From Processes, Not Contained In Them**
> 2.1. Rules are explicit constraints on behaviour and/or provide support to behaviour.
> 2.2. Rules are not process and not procedure. They should not be contained in either of these.
> 2.3. Rules apply across processes and procedures. There should be one cohesive body of rules, enforced consistently across all relevant areas of business activity.
> ...
> **Article 4. Declarative, Not Procedural**
> 4.1. Rules should be expressed declaratively in natural-language sentences for the business audience.
> ...
> 4.3. A set of statements is declarative only if the set has no implicit sequencing.

From the quote above, the relationship between business rules and business processes is made clear. Additionally, it can be seen that the Business Rules Approach prescribes declarative usage of business rules.

Another very important work is the book "What not How, the Business Rules Approach to Application Development" [46], written by database pioneer C. J. Date, with a foreword by Ron Ross. describes a system that exposes the data model and completes it through rules such as constraints and inferences. The system described is devoid of processes only to be managed through a generic user interface and perhaps a querying language. The approach presented is a natural extension of the relational model that is the cornerstone of most modern databases. The aim is to eliminate coding and represent business logic solely through declarative business rules. The book however does not discuss integration with other components or services and talks only briefly about the user interface. Nevertheless, many useful concepts are illustrated as well as the general vision of fully declarative information systems. In the book, a very useful classification of business rules is also presented:

- Rule

  Constraint

  State Constraint

  Transition Constraint

Stimulus/Response

Derivation

Computation

Inference

The approach presented in this section is in many regards based on the work presented by C. J. Date.

Other Relevant Work

During the last few years, a number of other approaches to the creation of information systems have appeared, some of which have similarities with the work presented here.

The currently most popular web application framework is Ruby on Rails[47]. This framework contains a number of features that are interesting from the viewpoint of a declarative information system. One such feature is the declarative domain model that it uses. A ruby on rails application is based on a domain model that the programmers define and which is used to auto-generate the database structure. In this way the programmers don't have to interact with the database at all. Another interesting feature is what is called 'scaffolding'. Based on the domain model, forms are created that allow manipulation of the data in the database through a web browser. By combining the two features together, a very basic web application can be built entirely through the domain model, without getting into imperative code. Finally, Ruby on Rails, with its latest version 2.0 has integrated full support for REST. At the same time, support for WS-* web services has been removed from the built-in libraries. These three features can be said to have equivalents in the declarative information systems approach presented here.

Another interesting Effort is executable UML (xUML)[48]. xUML uses a subset of the UML notation in order to allow for unambiguous specifications of information systems. A platform then uses these specifications to produce a working application. In this regard xUML is similar to the proposed approach. The weakness of xUML is in its need for imperative code, called Action Language, which it requires in order to handle more advanced scenarios. This is an aspect that is not desirable and may present obstacles to business users.

Also of interest is a project by IBM research called Collage [49]. This project is interesting in its focus on declaratively creating distributed applications. Its usage

of the Recursive MVC pattern and RDF as a unifying data model are also features that can be used in a possible implementation.

Finally, another approach to the problem of simplifying information system development, coming from the field of object-orientated programming is the 'Naked Objects' pattern and associated framework. The pattern was proposed by Richard Pawson in a series of publications including his doctoral thesis in 2003. The naked object pattern is based on three pillars.

Behavioral completeness All objects in the domain model are to be modelled in a way that encapsulates all relevant behavior. According to the naked objects pattern, this is the only necessary code required to model an information system.

Expressive Systems The naked objects framework uses the code of the domain objects in order to generate the entire information system, including the persistence layer and user interface. The interface generated however is not like the usual process oriented interfaces but rather gives the user direct access to the objects (hence the title of the pattern) and their associated functionality.

Users as Problem-Solvers, not Process-Followers Giving the users access to the objects of the domain model rather than predefined processes requires the newly empowered users to learn a new way of interacting with the system. They must access the objects in order to modify the information they need which means they must first understand the domain model. This increased learning curve has been recognised by the creators of the pattern who recognise it is not the most suitable interface for every possible information system.

The last two pillars of the Naked objects, as connected to the user experience when dealing with an information system are wholly compatible with the idea of an information system without explicit processes. The view of the Naked Objects pattern on the user interface issue has been used in conjunction with the Business Rules Approach in order to create a vision for a completely declarative information system that can offer an innovative experience by leveraging the explicit business logic to empower the users even at the user interface level.

Usage of SBVR

The language that has been chosen as the expressive medium for the business rules is OMG standard Semantics of Business Vocabulary and Business Rules (SBVR

[33]). As defined by [50], "SBVR provides a way to capture specifications in natural language and represent them in formal logic so they can be machine-processed". This allows users to be able to verify the requested service composition by directly reading the structured natural language used by SBVR which can then be parsed and executed by a machine.

Since SBVR is a way to capture specifications, there is no technical barrier to following a specific programming style. Specifically, one could conceivably use it to specify the structure of a process therefore conforming to an imperative programming style. While SBVR can be used in an imperative style, the full benefits of the Business Rules Approach that spawned it can only be attained when it is used declaratively.

Distributed computing architectural style

When examining the technologies available for interconnecting applications over a network as large as the internet, the architectural style termed REpresentational State Transfer (REST) is seen to have an advantage over the traditional web service style as implemented by the WS-* stack. Essentially the WS-* stack represents mainly RPC-style interaction tunneled through the HTTP protocol as a transport. Despite the effort invested in WS-* by the large middleware vendors and standards organizations, the APIs of major web applications from Google[51], Amazon[52] and many others are built using REST. Even WS-* heavyweights such as Microsoft are working on projects that utilize REST [53], [54]. Also recently a number of standards have been approved that are compliant with and based on the REST style. [55], [56].

Contrary to what may be assumed, REST is not a new development. It was first identified in 1999 by Roy Fielding in his PhD dissertation [57] as a term to describe the architectural style used by the web. Consequently, HTTP 1.1 was released to better align the web with the principles of REST. While many have since championed REST as a competing web service paradigm to the WS-* stack, it has only recently begun to be more seriously considered with the publication of works such as [58] and the apparent lack of expected adoption for WS-* technologies outside the corporate firewall.

The main concept of REST is the resource as a document that is identified by a URI, a uniform resource identifier. Resources are to be accessed by a universal interface of well defined methods that should be resource-agnostic and therefore have

the same, standard effect on all resources. In the case of HTTP 1.1, the methods include GET, PUT, POST and DELETE [59]. Other protocols such as WebDAV define methods such as LOCK and UNLOCK, suitable for transactions on resources. This Spartan interface is in contrast with the WS-* approach of defining a multitude of unique methods, one for each use case to be executed.

In the context of declarative information systems, the separation between verbs (methods) and nouns (resources) encourages a more declarative style of expression. Additionally, the client is concerned with resources rather than the providers of the services or orchestrating procedure calls. REST also aids more loose-coupled service composition since service providers can be alternated based on resource provided rather than interface compatibility.

B.1.2 Declarative Information Systems

Bringing together the Business Rules Approach as presented by Ron Ross with the implementation principles of C. J. Date's work and the interface of naked objects, it is possible to imagine an information system that is created declaratively. This system will consist of the following elements:

Generic Platform

The platform is the implementation of the engine that makes sure that the business logic is enforced, the state is kept consistent and the user can reach his or her goals. The platform is not customised for every different information system that it is used in but remains in its original form, using the business logic to adapt to different business contexts. This provides the advantage of easily updating the platform when a new release is available. This would not be possible with an imperative system where the code is heavily customised or written from scratch for each instance of an information system it is used in.

Business Logic

The business logic consists of all the information required by the platform to present the behaviour of a complete information system. It is desirable that all the logic is expressed as SBVR in both vocabulary and rules. However it is possible that representations other than SBVR Structured English may be more suitable for specific types of business logic.

State

The state represents the information that is in the system at any given time. The structure of this information is determined by the business logic; however the state itself is independent from the business logic. It can be thought that while the business logic may declare the existence of the term car and the various properties that belong to a car, the actual instances of cars that belong to a business are part of the state and not part of the business logic. It is crucial to make this distinction as this separates the information that may be stored in a database from the information that will be stored as business logic.

Interface

The interface can be different depending on the nature of the client. It a computer is acting as a client, the interface of the system should basically be a REST API that exposes the relevant resources over an HTTP connection. Alternatively, a human should be presented with an interface that allows the same expressivity as the API. Essentially this is based on Naked Objects' concept of 'users as problem solvers, not process followers', informed by the constraints of the REST architectural style. Therefore the interface should present the user with the available resources and the ability to browse or modify each of them as the user chooses. It is possible however that the pure API-like interface is not appropriate for all scenarios, especially when use of a system is transitive and the user will only want to make very specific interactions with the system. In this case, there should exist the capability of layering a 'traditional' user interface on top of the API so as to make the system accessible to all users. With the additional interface layer, although it becomes less expressive from the perspective of the user, the system should be indistinguishable from a traditional information system.

The meta-process

An Elementary Business Process can be thought of as a set of activities performed by or on behalf of an actor within a single session through which the actor may reach certain objectives while maintaining business logic. These objectives can consist of reading the system state and/or altering it. An elementary business process can consist of the following stages:

1. *Authentication* - Generally, authentication must take place before access to

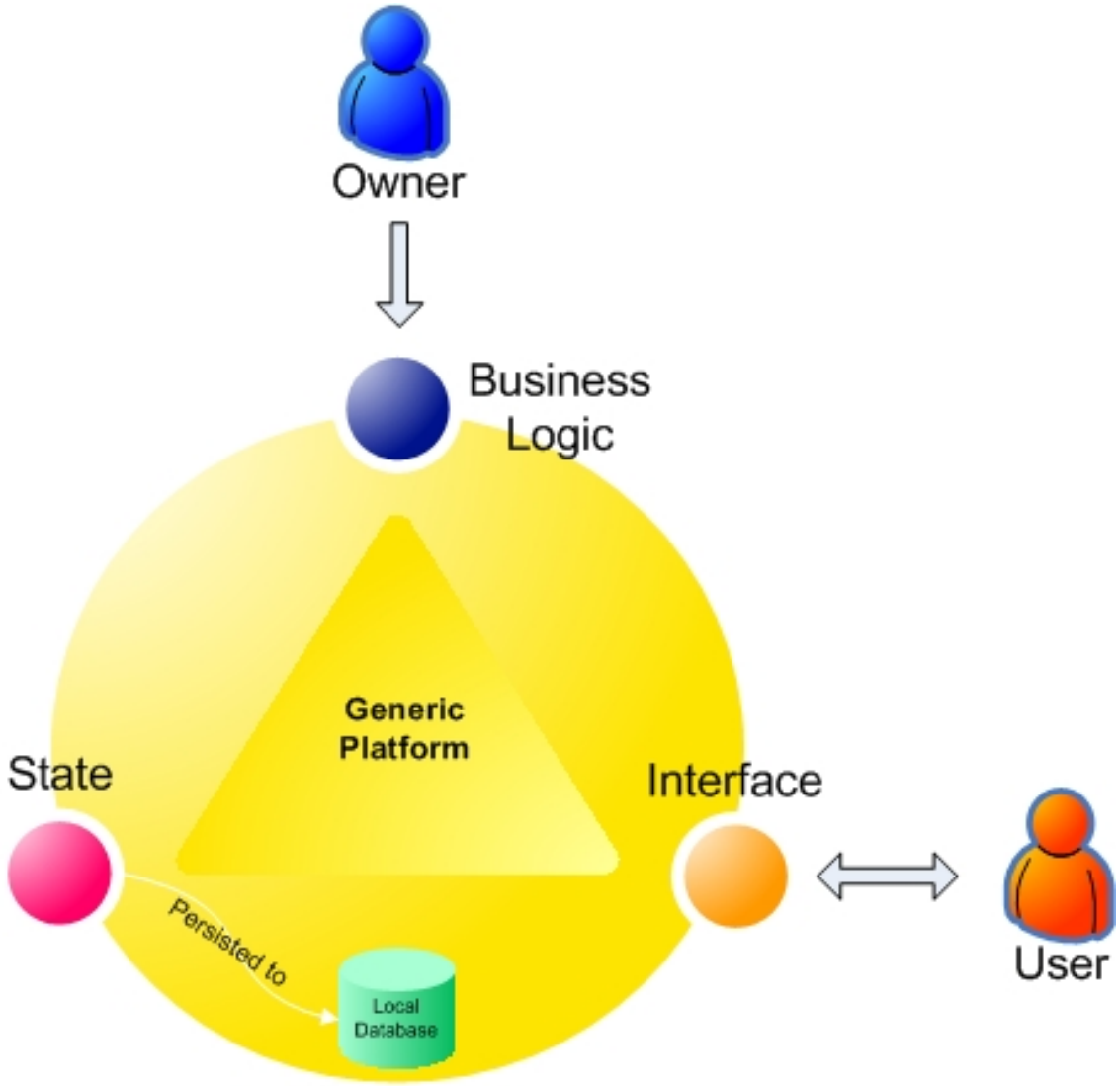


Figure B.2: Illustration of a declarative information system.

a restricted system resource is required. Therefore it is conceivable that a system allows the user to proceed with requirements extraction (stage 2) before requesting authentication (as is a typical e-commerce pattern) or even does not require authentication at all if the process can complete without access to restricted resources.

2. *Requirements extraction* - The user is led through a series of steps which are aimed at forming an action request (query) that is consistent with the business logic and satisfies the user's objectives. Once a compliant query has been reached, the actor may have a choice to further refine it or confirm its execution.

3. *Query execution* -The system attempts to read or alter its state accordingly and returns the results to the actor. If there are more than one changes needed to the system state, they must be executed in a transactional manner to ensure atomicity and isolation.

4. *Results and Rollback* - Especially in processes where the aim was to extract information, a stage dedicated to presenting the requested information is necessary. However any process might need such a stage, even simply to notify the user that the transaction has been completed. Additionally, depending on the system and process, the user may be presented with a menu of options for what she would like to do next and/or an option to end the session by logging out. An option for logging out is dependent on whether authentication was required or not.

A more complex business process, such as those involving multiple actors and sessions, may be decomposed into a set of simple business processes each involving a single actor and session.

The stages presented in fact establish a meta-process, one that can be used to guide the interaction of the system with the user. It is through this meta-process that the business rules will be applied therefore presenting the user with the same behaviour as an optimally designed process-oriented system would, while maintaining the advantages of the business rules approach.

Additionally, the meta-process itself could modify its behaviour according to rules to optimize the behaviour of the system for various businesses (e.g. late authorization, custom authorization logic, resource usage vs. improved user guidance).

Elements of Business Logic

An interesting issue is, in a world without processes, what information is needed to model an information system and how will this information be represented? The information currently identified is as follows:

1. *Data model* - The data model or domain model is an integral part of any information system and this does not change. Where processes used to guarantee the validity of the data contained within it, now rules will perform the same task. In fact, it is conceivable that the data model itself can be represented as a combination of business terminology and rules.

2. *Rich typing system* - Since we plan on explicitly representing the business logic, a more accurate description of the data model may be necessary through the use of a rich typing system, on a higher level than database primitives such as integer, string etc. The typing system may come with predefined types (e-mail, phone number etc.) but there must be the ability for the business to create new types and limit the data that can be entered into them. For example many businesses have unique product id schemes that must be clearly represented if the system is to check the input for consistency. An interesting aspect is whether regular expressions, which are declarative in nature, can be expressed through business rules. If this is possible, then business people will be able to express new types in structured natural language.

3. *Application Logic* - An extension to the data model, and perhaps inseparable from it, is the business logic that guarantees that changes made to the system state are consistent with the intent of the owners.

4. *Computations* - Also a type of business logic but different in nature, computations or derivations state what the value of a given variable is in relation to other variables. Computations can start simple but can also be very complex.

5. *Access Rights* - In order to reveal different parts of the system to different users, we need to define the resources each user has access to.

6. *User Requests* - A user must have means of requesting information or change to the system state.

7. *Presentation* - An information system that is automatically generated must have a way to communicate with the user and respond to her requests. In the

real world, presentation layers can be very complex and often display arbitrary procedurality to achieve an aesthetic effect or usability objective. Our system must be able to accommodate such use cases.

Expressing the Business Logic

We must now determine the language that will be used to express the required information in. Since our goal is to be machine- and human-readable at the same time, an obvious candidate is SBVR. It must then be examined to what extent SBVR can express the information needs identified in the previous section.

1. *Data model* - SBVR can express the data model adequately be defining terms, properties, categories and relations. It is in fact conceivable that a vocabulary such as the one described in section E.2.2.1.4 of the SBVR Specification (Car Specifications) can be directly converted into a relational database model and forms for management of the resulting tables.

2. *Rich Typing System* - SBVR can by nature express constraints on ranges of values. This capability can be used to express complex data types. A simple example is the 'fuel level' term in the EU-Rent case study which can be directly converted to an enumeration. More research is required to find how regular expressions and other complex concepts such as numerical ranges can be expressed in SBVR.

3. *Application Logic* - This area is what SBVR has been explicitly designed for so it is safe to assume that it can competently represent it. Examples can be found all over the SBVR specification, especially in section E.2.2.2

4. *Computations* - While SBVR has been created for business logic, computations appear to pose a challenge for it. It is believed that by adding a vocabulary for mathematical expressions, SBVR will be able to express such constructs.

5. *Access Rights* - When the business logic has been set up, we need to define the parts to which a user can have access. In order to achieve this, the users and/or their roles have to be modelled as well. While it is conceivable that this information can be expressed in SBVR, it must be examined if this use is compatible with the business rules approach.

6. *User Requests* - User requests are a particularly complicated part of the expressive needs of an information system. In a procedural system, requests rarely ever get formalized and when they do, they usually stay within the internal representation of the system. Expressing user requests in a formal manner allows us to open the system to interoperability with other systems and free the user from the necessity of using structured processes. However user requests are fundamentally different from business logic in the sense that the require action rather than simply describing relations between concepts. This limitation can be overcome by thinking of the user's request as a description of work to be undertaken by an active agent such as the information system therefore allowing us to express it using a passive business rules and SBVR in particular

7. *Presentation* - Two approaches are possible for handling presentation. The first one is relying on templates (in a language such as Apache Velocity or JSP) that can be possibly be modified to be compatible with a rule-oriented system. The alternative is using rules to auto-generate a user interface. Since presentation is inherently procedural and very often subjective, the system can only go up to a point in auto-generating it. A system cannot express all possible user interfaces since aesthetics is not formalized. For this reason both approaches have their use and should be combined to achieve the desired result of unlimited interaction with the user (through auto-generated interfaces) and a highly specified user interface for usual or recommended paths. In this way a traditional user interface can be layered on top of a rule-oriented system if needed. This of course sacrifices flexibility for the user however there are specific scenarios where users simply want to execute specific tasks rather than have full access to the system's logic. These systems are ones where users spend very little time on and may have little use for the additional power supplied by a flexible interface. Alternatively, the approaches can be used in conjunction to allow the more experienced users to achieve more complex tasks while providing new users with a short learning curve.

B.1.3 Service composition

Having laid the foundations for the declarative creation of information systems, another aspect that is vital to the approach presented in the context of a Digital

Ecosystem is composing the resources exposed by these systems in order to provide higher-level resources, according to the user's specifications.

Service composition is the process of combining atomic services in order to achieve a composite goal. Service composition is separated into manual composition and automated composition. In manual composition, the services are positioned in a workflow by the user whereas in automated composition the user defines the goal of the service composition and depends on a software tool to define the most suitable way to achieve this goal. According to [60], automated composition can also be subdivided into three categories.

The first is 'Fulfilling Preconditions' in which pre-existing services are combined in UNIX pipeline fashion to fulfill the requirements of a service that does not exist in atomic form. For example a .doc to .pdf converter and a .pdf printer can be combined to emulate a .doc printer. The second is 'Generating Multiple Effects' in which a number of services should be executed to produce separate but interrelated outcomes. A typical example of this type of composition is the travel scenario where flight and hotel can be booked independently but must be coordinated for their result to be useful.Finally, a type of composition called 'Dealing with Missing Knowledge' is defined, where for instance a list of metro stations may be combined with a list of hotel addresses to identify hotels near metro stations. In this type of composition additional information sources are queried and the results are combined with the results from a service provider to satisfy more complicated queries.While these types of service composition can be independent, there are problem domains which require a combination of approach. In this regard, the three types can be considered building blocks for a complete service composition system.

Since we are operating within the context of a business ecosystem, the initial focus is on 'Generating Multiple Effects' type of service composition that can handle use cases similar to the travel scenario. 'Fulfilling Preconditions' and 'Dealing with Missing Knowledge' types of composition can be then added incrementally. To achieve service composition, Declarative Programming, SBVR and REST are combined to provide the foundations. SBVR is used declaratively to state a user's requirements and also describe the available services. REST is used as the interface that a service exposes to the network and as a basis for the querying mechanism. It can be said that REST is used as a machine-to-machine integration avenue, while SBVR is layered on top of it as a human-to-human communication medium. SBVR's formal logic underpinnings combined with the noun-verb separation of REST make

such a pairing feasible.

B.1.4 Implementation Principles

Service Description

While RESTful web services have had uptake in practice, a suitable description language has not yet been standardized. A notable effort is Web Application Description Language [61] developed by SUN which is purpose built for describing restful web services but not yet mature. An alternative is WSDL 2.0 [62] which has been extended to express RESTful services but it's WS-* rooted complexity is a significant drawback. Additionally, expressivity provided by these languages is limited to variable types. Additional implementation information is expressed as text to be read by a developer. Leveraging SBVR as the basis of a service description language should provide sufficient capabilities for integration without human intervention while reducing the complexity level presented by these languages. Additionally SBVR can provide integration advantages when used as a language to describe both services and requests on these services. An additional benefit of using SBVR is that if the service is implemented using a declarative information systems approach as described, a subset of the business logic that is used to create the information system itself can be used to

Repository

A service repository in a RESTful architecture has not yet been considered in literature. This seems reasonable considering the poor uptake of UDDI and the usage patterns associated with web services on the public internet. It is however useful to consider it within the context of a Digital Business Ecosystem. What is needed from the repository is to have semantics for all the resource types used within the ecosystem and the ability for service providers to register as providers of a specific resource. Any provider should be able to create a new resource type however it is expected that market forces will lead to demand-driven standardization of resource types. In fact it should be more feasible for a business owner to register as a provider of the competition's resource than it is to implement the same service interface as would be required in the WS-* approach.

Requirements Expression

Since our focus is on the 'Generating Multiple Effects' type of service composition, we should use SBVR as a means of expressing the desired effects and correlations between them. Within a RESTful architecture this translates to resources and constraints on their attributes. Therefore a two step process is needed. First the user selects the desired resources from the repository from which an SBVR vocabulary is generated. Then, based on this vocabulary, rules are written to express what combinations of these resources are acceptable with both absolute and relative constraints. Absolute constraints refer to properties of the combination itself whereas relative constraints refer to properties of a given combination of resources in comparison to all other available combinations.

Combination Generation

According to the vocabulary and rules that express the request, the combination generation algorithm should query the relevant providers and determine the combinations that satisfy the absolute constraints. Methods of RESTful querying are already implemented in Microsoft Astoria and Google's GData [63] which can serve as a reference. Determining the appropriate combinations requires identifying a suitable algorithm. Backtracking may fit the requirements; however there is probably a more efficient way to search the solution space for matches. It is important to note here that bounded and unbounded resources should be treated at different phases due to their different nature. At first, range queries are made to all bounded resources once and viable combinations of the results generated. Afterwards, the unbounded resources are queried repeatedly, once for each generated combination of bounded resources. At the end of this process, a list of combinations exists.

Combination Evaluation

After the suitable combinations are determined, they can be evaluated either manually or automatically. Automated evaluation depends on relative constraints entered during requirements expression whereas manual evaluation returns the combinations to the user for arbitrary ordering. These two approaches can be combined by using automated evaluation as a step before manual evaluation therefore easing the user's work. However, if a composition is to be exposed as a service itself, manual evaluation is not feasible.

Transaction Generation

Once the combinations to be executed have been determined, they should be combined into a transaction tree to be executed by the transaction model. The existence of more than one suitable combination is necessary due to the fact that the resources have not been locked since querying. Locking resources in this fashion would place great strain on providers and lead to system-wide delays.This can be achieved by merging the combinations into a single transaction tree. The transaction tree can be represented with the notation presented in [64]. While the easiest way to do this is to utilize a serial alternative coordinator at the top level and connect to it the different combinations as separate sub-trees, this foregoes a lot of potential for optimization. Ideally, the transaction tree should utilize similarities between the combinations to create more optimal transaction trees that contain fewer leaf nodes and therefore avoid unnecessary service invocations.

B.1.5 Service Composition within an Information System

Having presented the vision on standalone declarative information systems and standalone service composition, based on the same technologies and principles, the next challenge is to integrate service composition as part of an information system. In order to achieve this, a number of elements of the two approaches must be redefined in order to allow them to be understood as a single approach. Specifically, the state of a declarative information system, can now be seen as spreading across the entire network to all the resources that the system, and therefore its users, have access to. Additionally, the service composition requirements that we expected the user to define for service composition to take place, can be thought of as part of the business logic of the system. In this way, it can be thought that the service composition is defined by the business owner at a generic level and is instantiated by the user for a specific scenario. In this case, the user would only need to enter the instance data for the instantiation providing a very similar experience to systems that perform service composition using imperative code, while maintaining the flexibility and direct control by the owner that the declarative paradigm provides.

It must be noted that while the system can connect to many other systems through the network, it is not obligatory that the other systems are implemented in the same way. The only thing that is required is that the systems expose a RESTful interface and a description in SBVR. Therefore, the information systems

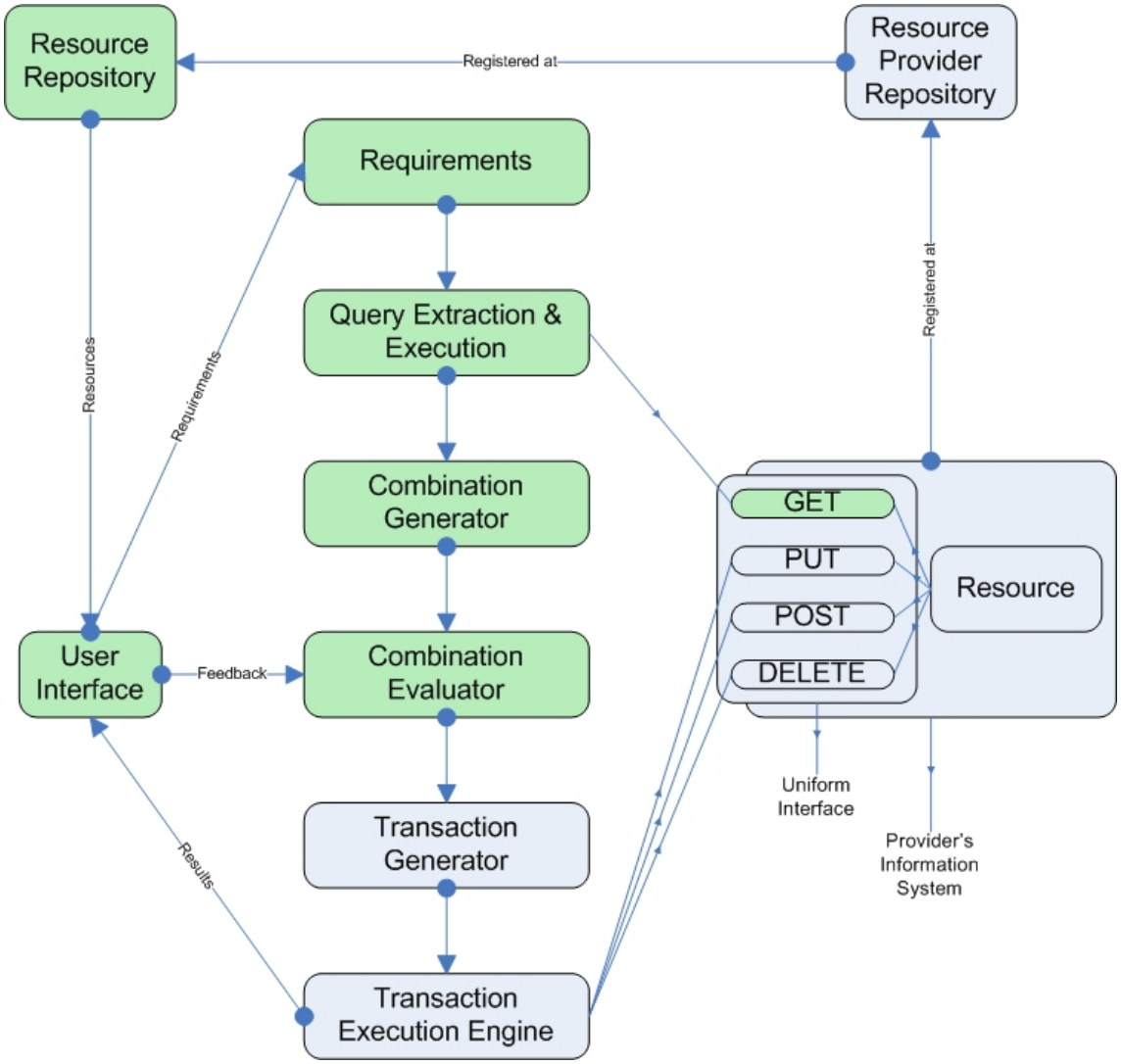


Figure B.3: Service Composition Workflow.

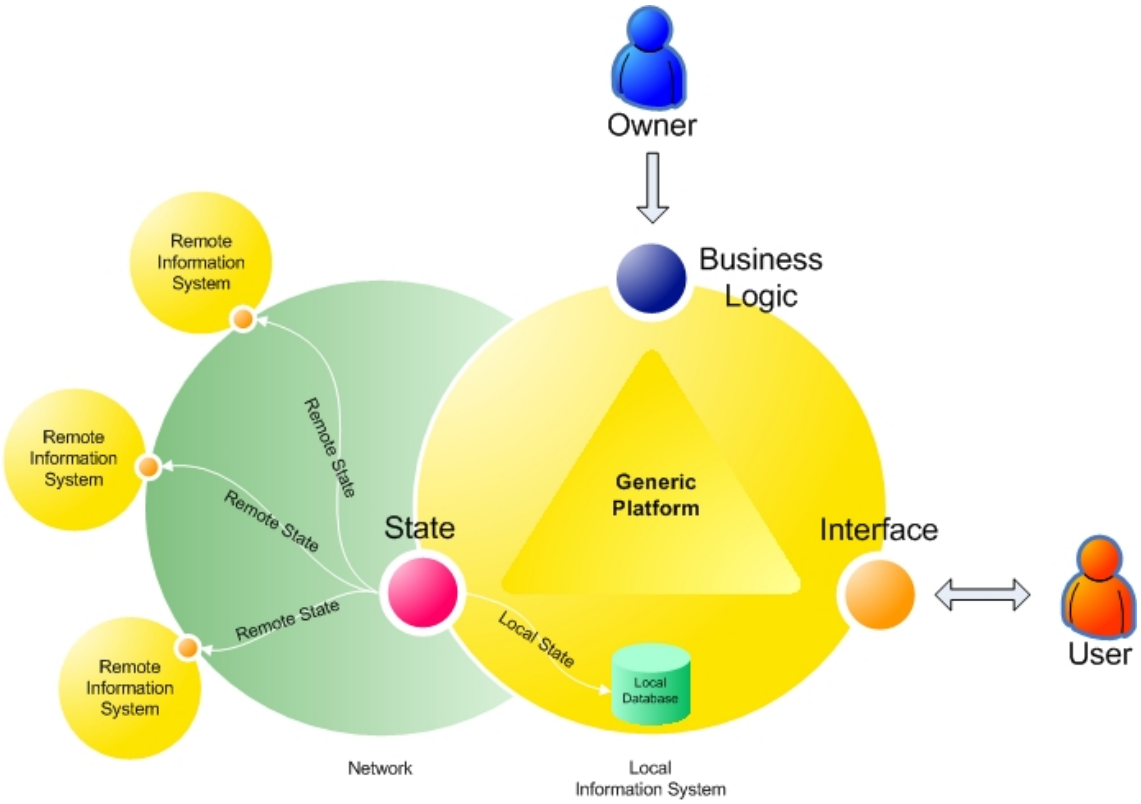


Figure B.4: The complete picture.

themselves can be implemented in any possible way, including legacy systems that utilize adapters to participate in the Digital Ecosystem or even entirely manual systems with a thin computing layer that defers all the processing to humans.

Another point that must be noted is that while the work on service composition focuses on retrieving providers through a repository, it is possible that an information system will need to connect to a specific remote provider, for example an ATM needing to connect to the information system of the bank that owns it. For this, the business logic must allow systems to specify specific providers therefore circumventing the need to use a registry, if the owner of the information system requires it.

B.1.6 Discussion

The criticisms of a system such as the one presented are mainly focused on two aspects. First of all, the performance that such a system can yield when compared to an analogous imperative system is an area of concern. Here we must again look to the performance issues of every new level of abstraction introduced in computer programming. Higher level languages have always had to deal with this criticism; however successive implementations have sometimes performed better than the original language. This is due to the fact that optimizations can be applied at the level of the platform and be utilized by all the systems that utilize this platform. Therefore it is worth the time to optimize every detail of the system, whereas this would not always the case with an imperative system. A very telling example is the paradigm of the relational database. While initially criticised for lack of performance, today it is very rare to see someone preferring to create custom code to access data structures. This is because database implementations have evolved to the point where their performance better than what an ordinary programmer could write for most cases, making the alternative rarely worth the effort.

Another criticism of depending on a platform is the fear for lack of flexibility. In fact this may be relevant to the need to learn a new method of programming therefore creating the impression that the more unfamiliar paradigm is less capable. Another reason may be the need to modify the business logic at a top level for any changes to take place in the system's behaviour. This may seem cumbersome in some cases where only small change is required. However it is always in the long term interest of the business to have the requirements of the system formally specified and therefore even if more effort is required in some cases, this effort is outweighed

by the benefits of maintainability and enforcement guarantee that a system based on the Business Rules Approach can provide.

Advantages of the approach presented include improved time to completion of a project and reduction in the probability for software errors assuming the underlying platform is reasonably well tested. This also implies a drastic reduction in cost of the system or equivalently an increase in the capabilities of the system that can be created for the same expenditure.

The separation between business logic and platform can yield another two advantages. First of all, the platform can be updated when a new version is available without altering the business logic. This allows the business to utilize development done upstream from the providers of the platform. Additionally, if a business wants to effect architectural change in the information system, for example moving to a client-server or web-based paradigm the platform can support it therefore providing reusability for the business logic. The alternative would have been an almost complete rewrite of the system.

Finally, with the usage of SBVR Structured English notation or similar, it can be said that business users can be much closer to the decisions taken for the system. The owner can at least audit the business logic that has been used for the system if not alter it. This is a very significant advantage for business owners who either have to trust a programmer or software vendor with implementation of the business logic or have to delay or forego implementation of an information system entirely due to this added complexity.

It is also very important to discuss the relationship of this approach to automated code generation. While the approach produces systems with behaviour equivalent to systems with generated code, the approach to executing the models may or may not involve code generation. This depends on the implementation of the generic platform. For example run-time interpretation of the business logic or just-in-time compilation may prove to be as efficient. In general, the code of the system can be said to be the actual business logic entered by the owner. Its compilation for reasons of performance or others is not vital to the approach that this strategy is followed. However it is important to note that the functional result of the approach should be the same as a system that generates complete code, including method bodies, for the range of scenarios that the platform has been built to handle.

B.2 A Domain Model Repository for a Digital Ecosystem [LSE]

A repository (possibly distributed) is envisioned in a Digital Ecosystem to store the collective public conceptual schemas (domain model) of each of the users of the Digital Ecosystem. The metaschema for the conceptual schema is SBVR. The repository is an implementation of the OMG MOF 2.0 specification. The repository can be used as a general purpose knowledge repository. A feature of the conceptual schema representation in the repository is that it is indexed in a concept lattice, which helps authors of domain models to find concepts they need to express themselves, and to add their own concepts as the community vocabulary evolves, while minimizing redundancy.

B.2.1 A Community Repository for Digital Ecosystem Users

A repository (possibly distributed) is envisioned in a Digital Ecosystem to store the collective public conceptual schemas of each of the users of the Digital Ecosystem. The repository is a community collective of the Digital Ecosystem so that any user can potentially use the conceptual schema of another user together with their own conceptual schema to create peer-to-peer applications between them. Of course, users may want to keep certain aspects of their schemas private. Such private schemas are held in a local repository of similar structure, to be shared only with specific other users of the owner's choosing. Each local repository logically includes the community collective repository and any other accessible private repositories needed.

B.2.2 Conceptual Schemas and Domain Models

A conceptual schema is a combination of concepts and facts (with semantic formulations that define them) of what is possible, necessary, permissible, and obligatory in each possible world in the domain of a user of a Digital Ecosystem. A conceptual schema is also known as a domain model. A concept and the term that represents it are distinct in the repository. The facts are stated using a vocabulary that represents the concepts that are involved in the rules using a grammar that includes the vocabulary. A conceptual schema can be said to be fact-oriented. A conceptual schema is not object-oriented, as the term is generally applied to analysis and design

of information systems, although a concept corresponds to a type of an object in the OO paradigm.

B.2.3 A Concept Lattice Organizes the Conceptual Schemas

The concepts of the collective conceptual schema are organized within the repository in a concept lattice. The lattice is a subsumption lattice. The location of each concept in the lattice is based on the meaning of the concept: each concept subsumes the characteristics of the concepts below it in the lattice, and is subsumed by those above it. Each concept has at least an English representation, whose English definition is authoritative in defining the concept and hence its position in the lattice. Other representations of a concept, including synonyms in English or any other language, can also be included. The lattice is an index of concepts, which helps authors of conceptual schemas to find concepts they need to describe their domain, and helps them to avoid redundancy as they add new concepts.

B.2.4 Namespaces Segregate the User's Terminology

Each user has a namespace for the representation of concepts they wish to define and make available in the repository. General namespaces owned by a Digital Ecosystem administration function are also provided for commonly used concepts. A representation may be moved, together with its definition, with agreement of the respective namespace owners, from one namespace to another. A representation may be used by any user of the Digital Ecosystem to sate a fact.

B.2.5 SBVR is the Metaschema for the User's Conceptual Schemas

The metaschema for the conceptual schemas is based on the OMG's Semantics of Business Vocabulary and Business Rules specification (SBVR). The repository itself is an implementation of the OMG's Meta Object Facility (MOF) 2.0 specification. As such, the repository supports fine-grained access, identification, and version control of a schema at the level of model elements.

B.2.6 A General Purpose Repository

As a MOF repository, other MOF-based models, such as UML models - any model that has a MOF metamodel - can also be stored in the repository. This would include application design and deployment models derived from stored SBVR conceptual schemas. By defining a general MOF metamodel for documents, documents of all kinds can also be stored in the repository, each accessible as a file. Document contents can be accessed at the level of granularity defined for it in the MOF metamodel of the document. In this manner, the repository can serve as a general knowledge repository, such as that of the OPAALS OKS. In this manner also, computer program source code and object code, perhaps generated by transformation from stored models, can also be stored in and accessed from the repository. Communication of conceptual schemas in and out of the repository uses the XMI/XML format as specified in SBVR. Communication of other models and documents is similarly based on XMI.

B.2.7 The MOF Family of Specifications

With MOF 2.0, the OMG divided the MOF specification into several components, each of which, with the exception of the Core, are relatively independent, and represent different compliance points. Several of these, perhaps all, are relevant to the OPAALS repository described in this paper. These include eight specifications that are issued or in process in the OMG: MOF 2.0 Core, MOF 2.0 IDL Mapping, MOF 2.0 XMI Mapping, MOF 2.0 Versioning, MOF 2.0 Query/View/Transformations, MOF Models to Text Transformations, MOF 2.0 Facility RFP, MOF 2.0 Semantics RFP. This last one was raised as a result of SBVR to improve the capability of MOF to represent semantics. Most of the issued specifications includes one or more companion XML Schema Definition (XSD) files that can be used in development of conforming applications.

B.2.8 Available Repository Products

Commercial and open source MOF repositories are available. A leading commercial implementation is the Adaptive Metadata Manager http://www.adaptive.com, for which there are plans to support SBVR. The Eclipse Modeling Framework (EMF) includes an open source implementation of part of the MOF specification, http://www.eclipse.org/emf, called Ecore, which is functionally (not structurally)

equivalent to the Essential MOF (Emof) package of MOF 2.0 Core. No known MOF product includes a concept lattice index for SBVR.

B.2.9 Concept Lattice Implementations

Concept lattices are familiar to ontology projects, in some form or other. One prominent example is the Suggested Upper Merged Ontology (SUMO) http://www.ontologyportal.org/. SUMO was developed within the IEEE Standard Upper Ontology Working Group. Another is the open source OpenCyc http://www.opencyc.org/ . Both of the above-mentioned ontologies contain extensive vocabularies. Some of this content might be usable within OPAALS for a starter vocabulary in a general namespace in the repository. These each use their own knowledge representation language. SUMO uses KIF and OpenCyc uses CycL. Any of this content would need to be transformed into SBVR for use in the repository described here.

B.2.10 Example Concept Lattice

Much useful material related to this paper can be found in John Sowa's work. http://www.jfsowa.com/ . See http://www.jfsowa.com/logic/math.htm#Lattice for a formal description of a lattice. Figure B.5, excerpted from John Sowa's work, illustrates a top-level concept lattice. Additional concepts based on this would specialize these general concepts.

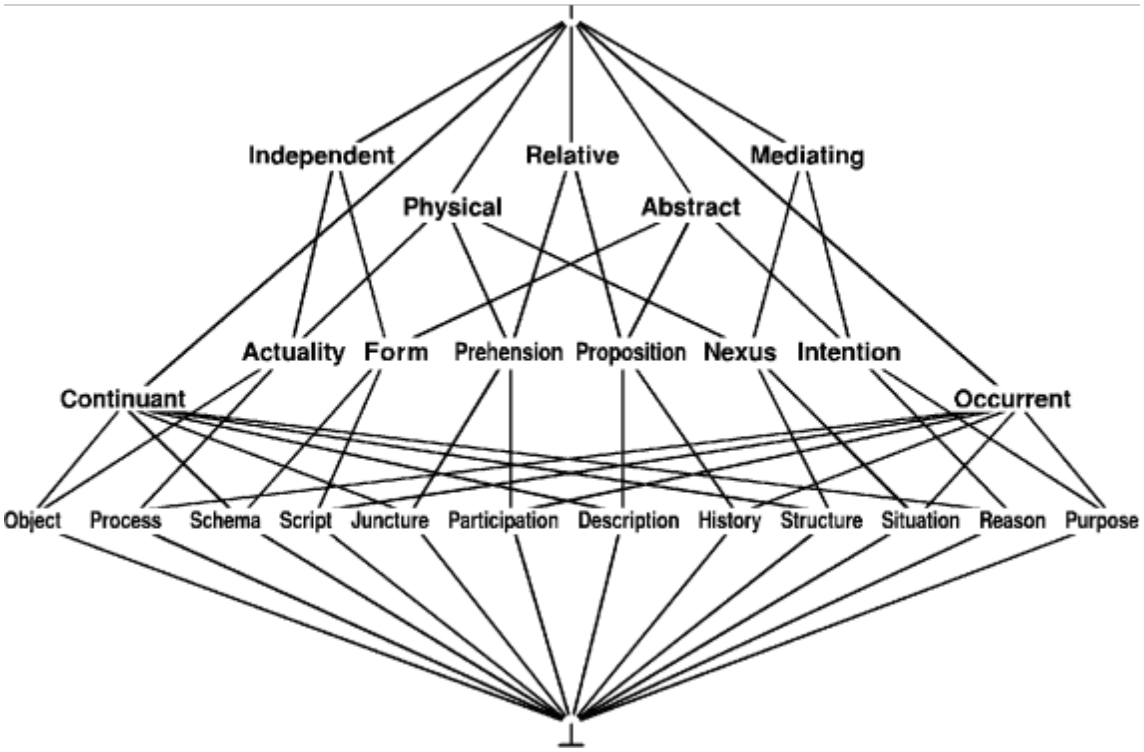


Figure B.5: Top-level concept lattice.

C Sourcecode

```
public static String findVB_N_D(String VP)
  {
    Pattern p =
      Pattern.compile("^\\t\\tVB(P|Z).*?(^\\t\\tVP(.*?))((^\\t\\t[^\\t])|\\z)",
      Pattern.DOTALL|Pattern.MULTILINE);
    Matcher m = p.matcher(VP);
    if(m.find()) {
      System.out.println("findVB_N_D find\n"+m.group(2)+"*");
      String lastVP = find_last_VP(m.group(2).replaceAll("(^|\\n)\\t\\t","\n\t"));
      System.out.println("lastVP\n"+lastVP);

      p = Pattern.compile("\\t\\tVB(N|D|G) .*?\\t+(.*?)$",
          Pattern.DOTALL|Pattern.MULTILINE);
      m = p.matcher(lastVP);
      if(m.find()) {
        System.out.println("findVB_N_D find\n"+m.group(2)+"#");
        return("is "+m.group(2));
      }
      return("Something went wrong (find_last_VP).");
    }

public static String find_last_VP(String VP) {

    Pattern p = Pattern.compile("(^\\t\\tVP .*?)((^\\t\\t[^\\t])|\\z)",
      Pattern.DOTALL|Pattern.MULTILINE);

    Matcher m = p.matcher(VP);
    if(m.find()) {
       return(find_last_VP(m.group(1).replaceAll("(^|\\n)\\t\\t","\n\t")));
    }

    return(VP);
  }
```

Listing C.1: Source code for pseudocde in section 2.3.

```
<workflow>

  <property name="src-gen" value="src-gen" />

  <property name="groovy-app-home" value="c:/work/na" />
  <property name="transformationOutputRoot" value="${groovy-app-home}/tmp" />
```

```
7   <property name="model" value="${groovy-app-home}/model.xmi" />
8   <property name="zipout" value="${groovy-app-home}/tmp.zip" />
9
10  <component class="at.tools.FileCreator">
11     <filename value='${model}' />
12  </component>
13
14  <component class="at.tools.FileDeleter">
15     <delete value='${groovy-app-home}/grails-app' />
16  </component>
17
18  <component class="at.tools.FileDeleter">
19      <delete value='${groovy-app-home}/model.xmi' />
20  </component>
21
22  <component class="at.opaals.sbvr.SBVRGenerator">
23     <inputSlot value="sbvrString" />
24      <destFilename value='${model}' />
25  </component>
26
27  <component class="org.openarchitectureware.emf.XmiReader">
28     <metaModelFile value="metamodel/metamodel.ecore" />
29     <modelFile value="${model}" />
30     <outputSlot value="model" />
31     <firstElementOnly value="true" />
32  </component>
33
34  <component class="org.openarchitectureware.xpand2.Generator">
35     <metaModel
36         class="org.openarchitectureware.type.emf.EmfMetaModel">
37        <metaModelFile value="metamodel/metamodel.ecore" />
38     </metaModel>
39     <expand
40         value="template::Grails::grailsModel_FOR_model" />
41     <outlet path="${groovy-app-home}/grails-app/"></outlet>
42     <outlet name='SRC_CONTROLLER' path='${groovy-app-home}/grails-app/controllers
           ' overwrite='false'/>
43     <outlet name='SRC_MODEL' path='${groovy-app-home}/grails-app/domain'
           overwrite='false'/>
44
45  </component>
46
47  <component class="at.tools.FileZipper">
48     <srcdir value='${groovy-app-home}' />
49     <targetfile value="${zipout}" />
50  </component>
51
52  <component class="at.tools.FileDeleter">
53     <delete value='${groovy-app-home}/grails-app' />
54  </component>
55
56  <component class="at.tools.FileDeleter">
57      <delete value='${groovy-app-home}/model.xmi' />
58  </component>
```

```
59
60 </workflow>
```

Listing C.2: Model Transformation Workflow of the Grails Prototype.

```
1  <<IMPORT metamodel>>
2  <<DEFINE grailsModelClass(String package) FOR Entity>>
3  <<FILE package.replaceAll("\\.","/")+"/"+name+".groovy" SRC_MODEL>>
4  /* package <<package>>; */
5  class <<name>> {
6     <<FOREACH attributes AS attr>>
7        <<IF attr.type.toString().matches("String")>>
8              <<attr.type>> <<attr.name>> <<IF (attr.defaultValue.length >>= 0)>> =
                  "<<attr.defaultValue>>" <<ENDIF->>
9        <<ELSEIF attr.type.toString().matches("Char")>>
10           <<attr.type>> <<attr.name>> <<IF (attr.defaultValue.length >>= 0)>> =
                 '<<attr.defaultValue>>' <<ENDIF->>
11       <<ELSE>>
12           <<attr.type>> <<attr.name>> <<IF (attr.defaultValue.length >>= 0)>> = <<
                attr.defaultValue>> <<ENDIF->>
13       <<ENDIF->>
14    <<ENDFOREACH->>
15    <<FOREACH references AS ref>>
16          <<IF ref.modality == "1">>
17             <<ref.type.name>> <<ref.name>>
18          <<ELSEIF ref.modality != "1">>
19             List <<ref.name>>
20       <<ENDIF->>
21    <<ENDFOREACH->>
22
23    <<IF hasMany.length >>= 1>>
24    static hasMany = [<<hasMany>>]
25    <<ENDIF->>
26
27
28    <<IF displayAttribute != null && displayAttribute.length >> 0  >>
29       public String toString() { <<displayAttribute>>; }
30    <<ENDIF>>
31 }
32 <<ENDFILE>>
33 <<ENDDEFINE>>
34
35 <<DEFINE grailsControllerClass(String package) FOR Entity>>
36 <<FILE package.replaceAll("\\.","/")+"/"+name+"Controller.groovy" SRC_CONTROLLER>>
37 /* package <<package>>; */
38 class <<name>>Controller {
39     def scaffold = <<name>>
40
41 }
42 <<ENDFILE>>
43 <<ENDDEFINE>>
44 <<DEFINE grailsModel FOR Model>>
45    <<FOREACH entities AS entity>>
46       <<EXPAND grailsModelClass(package) FOR entity >>
```

```
47        «EXPAND grailsControllerClass(package) FOR entity»
48    «ENDFOREACH»
49«ENDDEFINE»
```

Listing C.3: XPAND2 Template for the Model Transformation.