Applying Concept-Driven Engineering for Business Process Specifications

Peggy Schmidt
Christian-Albrechts-Universität zu Kiel
pesc@is.informatik.uni-kiel.de

Sebastian Kowski
Christian-Albrechts-Universität zu Kiel
sek@informatik.uni-kiel.de

Marion Behrens
University College Cork
mb20@cs.ucc.ie

Abstract:
This paper presents the principles of concept-driven engineering and the Concept-Manager tool as an implementation of these principles. Concept-Driven Engineering is capable of eleminating inconsistencies and redundancy that occurr within projects, i.e. in the software-development process to increase quality, decrease time to market, and increase flexibility. This method is based on the principle of human communication: concepts that classify objects by their characteristic features. Concepts are e.g. software artefacts, models, meta-models or (sets of) words. The Concept-Manager tool supports creation and organization of concepts and integration of generators, that add a certain syntax to a concept. The evolution of concepts is enabled through version paths and the management of generators over concepts. We demonstrate the practical use of the Concept-Manager tool by organizing BPMN and UML metamodels using the same or related concepts for similar components in order to apply the same syntax of the ASM generator.

1 Introduction

In todays competitive marketplace a critical factor for the success of a software product is the fast development time of software products. It is a highly cooperative and distributed activity involving working groups. A decision for a software supplier is dependend on quality and delivery time for surrendering of software products. To recognize this is extremely important, since software systems have grown more complex and much larger.

A correct construction of complex business objects from the more primitive ones is one of the basic problems during software development process. Variants and similar specifications and their graphics and the produced code are redundant composed. Artefacts are not subscripted into a repository. A descriptor assignment of specification artefacts could support a rapid prototyping for a precocious user feedback and test and validation of the right recording of requirement. We observed that specification artefacts are not executable. Recurrently work items of software production are not automated. That process dissipate development time and decrease the quality.

Concept-Driven Engineering is a software development methodology, which focuses on

defining and organizing concepts. Contrary to creating concrete models as propagated through Model-Driven Engineering (MDE), the usage of concepts provides a basis for any modeling: a common agreement of meanings and characteristics. A growing concept repository is accompanied by higher probability of finding reusable concepts. The possibility of evolution of concepts allows changes at all project stages without causing inconsistencies. The main benefit of concept-driven engineering is the convertibility of concepts by interfacing different semantics (i.e. generators).

We describe the theory of concepts, organization of concepts and semantics, which we adopted from linguistic and biological science, in section 3.1. The ConceptManager tool implements this theory and furthermore provides a basis for a concept repository with multiuser support. It faces the challenges to managing a growing portfolio of products, keeping up with rapidly changing technology and integrating newly requeirements. Our tool supports the reusage of business languages to facilely author and maintain business logic as business needs change. ConceptManager tool mananges the automated model transformation and code generation process to spend less time for specification of artefacts and coding. The architecture of the ConceptManager is illustrated in section 3.2. A concept in our ConceptManager is something understood, and retained in the mind, from experience, reasoning and/or imagination; a generalization (generic, basic form), or abstraction (mental impression), of a particular set of instances or occurrences (specific, though different, recorded manifestations of the concept) [con]. To use common models like BPMN, UML statecharts and so on we have to depose the concepts of these models. A model is pattern of something to be made; any thing of a particular form, shape or construction, intended for imitation; primarily, a small pattern; a form in miniature of something to be made on a larger scale [mod]. Using the ConceptManager tool, we can assign a certain generator to a concept to create a target concept, e.g. a prototype-like executable model, from an existing concept. In software development, if customers explain their concerns to developers or developers present their models to the customers, the problem of using different languages arises. The generation of prototypes is a way to communicate with the user to validate and refine requirements. The generality of concepts-driven engineering becomes evident when we proceed from the level of model instances to concepts of metametamodels.

In recent years model-driven engineering along with the usage of metamodels has become accepted amongst both researchers and practitioners. The number of metamodels available for all kinds of modelling has been increasing. With every metamodel, transformation rules must be written in order to transform model instances and to integrate the new metamodel into the model-based development process. Today metamodels, which have been developed for similar modeling tasks by different groups, use similar concepts. As a result of distributed research and development, these are similar concepts, but not same or related concepts in the context of concept-driven engineering. In section 4 we use the ConceptManager tool to organize different metamodels for modeling of business processes (BPMN and UML state diagram). We illustrate how we can support the generation of transformation rules by reuse of concepts.

2 Related Work

From the Official Blender Model Repository [ble] a large number of blender models, representing objects from the real world, are available to be used in other models with the 3d graphical software. If the user needs a certain model (e.g., a model of a chair), he will find many different kinds (Office Chair, Chair 2, Mies Armchair, etc.). He cannot see in what kind of models (e.g., of rooms) these chairs have been used. On the contrary, if the user is searching for a certain room (e.g., a classroom) he will find a few models of rooms containing chairs and other furniture. But structural information about the room is not provided (e.g., a classroom contains a chalkboard and several desks each with two chairs and, because a classroom is inherited from a conventional room, it contains a door and windows). In our contribution of model management we are focussing on concepts (e.g., *chair*). We provide classification of concepts (e.g., a *classroom* is a *conventional room*) as well as structural information (e.g., a *conventional room* consists of *walls*, *ceiling*, *floor*, *windows* and a *door*).

The number of modeling notations and languages is huge and model transformation plays a key role in Model-Driven Architecture (MDA). Transformation languages and engines have been developed and a nearly unlimited number of transformation models have been created. In the Atlas Transformation Language (ATL) transformations zoo [atl] more than 100 documents containing transformation rules are available for mappings between instances of different metamodels. [IM08] describes the transformation rules for transformation of Feature Models (FM), which represent software product lines, to BPMN. Feature models consist of features and subfeatures hierarchically structured. For relationships between a feature and its subfeatures different concepts (*And*, *Alternative (xor)*, *Or*, *Mandatory* and *Optional*) are specified [Bat05]. Our approach to model transformation is based on the re-use of concepts and component-wise transformation: We assume that different modeling languages/notations make use of the same or related concepts (e.g., *and*, *xor* and *or* in models representing workflows). We will describe in 4 that by reuse of concepts amongst metamodels we will be able to derive transformation rules as well. Typical system specification consists of a number of models for different facets and aspects of the system. Sometimes, systems description is based on a variety of abstraction levels. Thalheim [Tha08] introduce model suites as a set of models with explicit associations among the models, with explicit controllers for maintenance of coherence of the models, with application schemata for their explicit maintenance and evolution, and tracers for establishment of their coherence.

The approach in [FLZ05] describes reference models for business processes. Their analysis of 30 process reference models is based on a framework consisting of criterias such as application domain, used process modeling languages, models size, known evaluations and applications of process reference models. Furthermore, we identify model domains, which have been dealt with, describe similarities and differences between the available process reference models, and point to open research questions. In [Bör07] Boerger proposed a small set of parameterized abstract models for workflow patterns, starting from first principles for sequential and distributed control. The resulting structural classification of those patterns into eight basic categories, four for sequential and four for parallel work-

flows, provides a semantical foundation for a rational evaluation of workflow patterns. A goal of software product lines is the economical assembly of programs in a family of programs. In the paper [BB08], they explore how theorems about program properties may be integrated into feature-based development of software product lines. The approach in [PSC01] deals with the growing number of system units the dependencies between them become vast and tangling. This paper investigates this problem, proposes a more general model (version model) to capture different facets of Aspect Oriented Programming as well as a partial solution towards unit consistency based on versions.

3 Approach of the ConceptManager Tool

3.1 Concepts

In our work *concepts* form the base unit of any model. The thought of "'anything is a concept"' was a guideline throughout the development of the tool that we describe in section 4. In this section we discuss the meaning of the term *concept* and specify its usage in the context of this work.

There exist many different approaches on defining concepts in the fields of cognitive science and philosophy. The term is also commonly used in everyday life. In its most basic meaning a concept (from latin: *concipere* = to conceive, to grasp sth.) can be seen as (1) a plan or a scheme for a project, (2) a first draft of a theory (also called conception) or (3) a mental summary of objects that have a set of common features or properties.

Piaget's hypothesis [Pia52] claims that children acquire their repertoire of concepts in a certain order, starting with basic sensorimotor concepts and gradually progressing from them to more abstract domains. The classical view on concepts assumes that concepts are definable and that they have an intension and an extension. The extension is the set of objects that can be attributed to the concept, the intension is the sum of the features that the objects of a concept have in common. Other definitions of a concept state that a concept has three different aspects or points of view. These are *syntax* (or *symbol*), *semantics* and a *corresponding object* in a user world. The semantics act as a bridge between syntax (or symbol) and a concrete object the symbol refers to, i.e. the semantics of a given symbol determines the referred object. Formal Concept Analysis (FCA) is used to derive an ontology (by means of a concept lattice) from collections of objects and their properties. Concept lattices offer a way to make use of algebraic functions with concepts.

In our view a concept is a general or abstract idea derived or inferred from specific objects or phenomena in the world. Concepts are used to classify objects by their characteristic features or relations to other objects. This view is a little theoretical and must be applied to the concrete instance of the Concept Manager. With our tool we provide a possibility to define concepts in a structured way. We distinguish between classification of concepts and structural descriptions of concepts. The classification represents 'is-a' relationships between the defined concepts and supports only single inheritance. This offers a way to define simple and complex concept hierarchies. The structural description of concepts

provides a means to represent 'part-of' relations and is used to compose a concept out of other concepts. These parts of a concept could be viewed as its features. They are inherited from a concept to all its subconcepts, which again can have additional features (that is additional parts).

Our approach was inspired from mereology. Mereology [mer] is a collection of axiomatic first-order theories dealing with parts and their respective wholes. In contrast to set theory, which takes the setmember relationship as fundamental, the core notion of mereology is the partwhole relationship. Mereology is both an application of predicate logic and a branch of formal ontology.

In this section we outline the theories of relations between concepts. We show how *is-a* inheritance and *part-of* composition with $n : n$ cardinality allows for inheritance of structure and concept substitution. Biological classification, i.e. the taxonomy described in Linnaeus' Systema Naturae, is a method for categorization of organisms. Taxonomic ranks are used to identify each level of the hierarchy. For example the class of 'mammals' is classified in three infraclasses, one of them is the infraclass 'marsupial'. We adopt this approach for the architecture of the ConceptManager tool. We name a concept $\mathcal{C}'$, that, in a classification hierarchy, is one rank below another concept $\mathcal{C}$ ($\mathcal{C}'$ is a $\mathcal{C}$), a *subconcept* of $\mathcal{C}$. We call $\mathcal{C}$ the *superconcept* of $\mathcal{C}'$. If a concept $\mathcal{C}^+$ has a *part-of*/*has-a* relationship to another concept $\mathcal{C}$ we call $\mathcal{C}^+$ a *structural concept* of $\mathcal{C}$ A concept is inheriting all structural elements of its superconcept. For example, if a 'marsupial' has a structural concept 'pouch', then on the 'family' rank the concept 'opossum' inherits the 'pouch'. Unlike inheritance where concepts only have one superconcept, this is a $n : n$ relationship. Structural concepts can as well be classified through having subconcepts and a superconcept. A structural concept can be substituted by any one of its subconcepts. For example, if an 'opossum' has 'feet' as structural concept, and we create 'feet with webs' as subconcept of 'feet' and 'water opossum' as subconcept of 'opossum', then we can substitute the structural concept 'feet' of 'opossum' with its subconcept 'feet with webs'.

Changing a concept is possible whether or not the concept has already been used or referenced. Changes in a concept, that has not been used nor referenced or locked, does not affect any related concepts or users. If the user wants to reuse an existing concept that has been used or referenced, he can do so as long as he does not add any changes to the concept. If a concept $\mathcal{C}$ that has been used or referenced must be changed for reuse, a duplicate concept $\mathcal{C}^*$ has to be created. In order to reconstruct the history of a concept, each duplicate concept $\mathcal{C}^*$ enters a *case-of* relationship with the original concept ($\mathcal{C}^*$ case-of $\mathcal{C}$). A concept can as well be a semantics concept which can be assigned to a concept Single concepts may be linked to different semantics in different contexts. Semantics provide a context for concepts if they are linked to them.

3.2 Architecture

We have developed a tool for collaborative management of concepts, called ConceptManager. The ConceptManager is based on concept structures and allows different users to

create and manage concepts and their structures in a coordinated and convenient way. Despite the creation of simple and complex concepts and concept structures the tool supports the derivation of new concepts from existing ones (including inheritance of the concept structure), a powerful concept search engine, reusability of existing concepts and the definition of constraints on concept structures. Moreover, we have integrated a collaborative management of code generators. This offers a genuine added value through the possibility to generate models, source code, natural language or the like from existing concept structures.

Generators are reusable entities and can be assigned to one or more concepts. Similarly, each concept may have an arbitrary set of generators assigned to it. The assignment of a generator to a concept adds a semantic to this concept. Through the assignment of more than one generator to a single concept can possibly have different semantics in different contexts. A further step is the composition of different "compatible" generators. This allows for user-defined combinations of different semantic realizations of individual concepts.

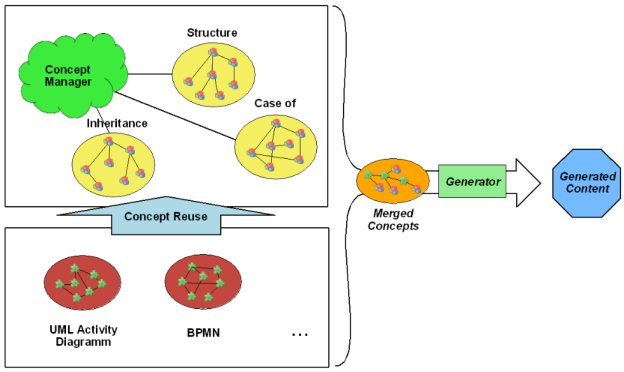

Figure 1: Schematic illustration of concept reuse in ConceptManager

Figure 1 shows a schematic illustration of the reuse of concepts in the ConceptManager. The tool offers various possibilities to manage the relations between concepts (*is-a* (inheritance), *part-of* (structure), *case-of*) already mentioned in section 3.1. The managed concepts can be reused in different contexts by merging them with other concepts, e.g. Business Process concepts. Arbitrary new content can be generated for the given context through the use of generators.

3.3 Operations on Concepts

The ConceptManager provides several functions for managing concepts. These are available via graphical user interfaces. The most basic function is to create new concepts. The user has several options here.

Creating a new and independent concept The user can create a concept which has no

relationships to other concepts. These relationships can still be manually inserted by the user afterwards. Creating a totally new concept can be useful e.g. when a new domain should be represented in the ConceptManager.

Creating a subconcept (*is-a* relationship) of an existing concept The user has the option to create a new concept *B* as a direct subconcept of an already existing concept *A*. This will automatically build a *is-a* relationship between the two concepts, i.e. concept *B* *is-a* *A*, and concept *B* automatically inherits the structure of concept *A*.

Creating a subconcept (*part-of* relationship) of an existing concept The user can directly create a new concept *B* as part of an already existing concept *A*. The concepts are then in a *part-of* relationship, i.e. concept *B* is *part-of* concept *A*. The new concept *B* is thus part of the structure of the concept *A*. Concept *B* is not (yet) in a *is-a* relationship to another concept.

Apart from creating concepts, existing concepts can be changed or deleted. There are also different variants to delete or remove a concept.

Remove from a concept structure A concept can be removed from the structure of another concept. This concept can be a simple concept without further substructures or a complex concept with an arbitrary substructure. Since reuse of concept structures is one main feature of the tool, a concept can be part of various other concepts. In this case, the user is given the possibility to further specify in which structures the concept should be removed.

Remove from concept classification A concept may be deleted from the classification of concepts. If this concept is not to a leaf in the classification hierarchy, the resulting gap in the hierarchy will be automatically closed, i.e. the subconcepts of the removed concept become direct subconcepts of the superconcept of the removed concept.

3.4 Copying concepts

The ConceptManager supports copying of concepts. This action can be triggered via context menu or via drag and drop. We have made a distinction between shallow copy and deep copy.

Shallow copy returns a reference to a concept or a partial structure of a concept that can be inserted as part of another concept. Several concepts then refer to the same partial concept structure. Deep copy creates real copies of the concept or the concepts of the substructure belonging to this concept. The copied concepts can then be inserted elsewhere.

While in the classification of concepts only deep copies are possible, concept structures allow for both shallow copy and deep copy.

3.5 Users, user groups, access rights

In a collaborative management of concepts authentication and authorization of users is necessary. Therefore the ConceptManager provides a mandatory user login. The credentials of a user consist of a user id and an id of an assigned user group. A user can be assigned to several groups and each group can consist of multiple users. In addition, there is a simple access rights management for the ConceptManager, comparable to access rights in Linux or Unix file systems.

Every concept in the ConceptManager owns a three-digit number describing the access rights for this concept. The first digit describes the rights for the logged on user, the second digit describes the rights for the user group of the logged on user, and the third digit describes the rights for all users. Each digit ranges from 0 to 7 and is the sum of the possible rights. The various rights are:

- **0** - general properties (user, user group, permissions, date of creation, etc.) can be displayed angezeigt werden
  - no other operations on the concept are permitted
- **1** - the concept may be read, i.e. its structure can be displayed
  - the concept may be copied
- **2** - the concept may be changed, moved, and extended
- **4** - the concept may be deleted
  - the access rights of the concept may be changed

E.g. the right 731 for a concept means that the current user has all rights to the concept, other members of the same user group may read, copy, modify the concept etc., all other users may only read and copy the concept.

The user groups together with the access rights can be used to restrict the view and work on concepts for certain groups of interest.

4 Usage Scenario: Concepts of Business Process Models

In this section we illustrate how we use the ConceptManager tool to efficiently transform new software artefacts (like new models, graphics, code). A software artefact is itself a concept.

We present how a software process starts with the requirements specification. Instead to write such a requirement specification in natural language, we suppose to write a formal requirement specification with a well-defined sub set of natural language. This formal natural language is deposited in the ConceptManager. From such a formal specification we can derive a specification in natural language, graphics, diagrams, and we are able to transform it to a first prototype (e.g. in a formal language like Abstract State Machines) to test and validate the specification. It is also possible to start with a graphic, resp. a diagram and transform this diagram to a formal specification. The advantages is that the specification, written in a natural language is consistent to the diagrams and may to the

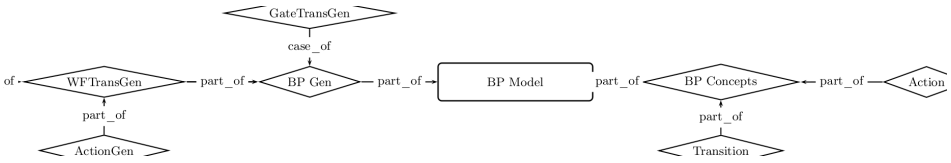

Figure 2: User defined General Business Concepts

implementation. We avoid the redundant declarations of concepts, for instance first as written words into the specification, then into the diagrams. The same holds for variants of all artefacts, like versions of specifications and diagrams. We are able to declare some first automatic quality tests on such a formal specifcation. The assurance of quality and reliability of specifications is essential. For instance Feja and Foetsch developed a visual notation for logical rules at the level of processes and workflows. This enables the business process engineer to use model checking techniques and to produce higher quality business models for subsequent software development [FF08] . We correlate concepts of the process-modelling domain to concepts of the domain of finite state machines. From this general mapping we derive mappings from BPMN models to ASM executable specifications with little effort by reusing existing concepts. We show the generality of this approach by similarly deriving a mapping from state charts to ASM.

In the first step the user has to think about the concepts of his doing. In our running example he has to define general business concepts as we see that current business process approaches are similar. As we see in the related work business process models, resp. notations like BPMN, statecharts and so on are modeled with similar concepts.

In our example at figure 2 the user specified that a business model consists of business concepts and business generators (BP Gen). He defined that business concepts are Actions and Transitions. A business generator needs a workflow transition generator (WFTrans-Gen) and gate generators. A workflow transition generator consists of a guard generator or and an Actions Generator.

Assume that the user works for two different clients. The first client wants that requirement specifications are specified with BPMN and the second client likes state charts. To reduce the development time of the generators which transforms executable code from a modelshe figures out these parts of the transformer which he can reuse in both models. He assigns each generation rule specific code. The result of such a transformer is presented in figure 3. The generator rule is based on the BPMN pattern of Boerger/Thalheim [BT08] To reuse the generation of ASMs we have to define these functions depend on the concrete BPM concept (e.g. BPMN or Statechart): inArcs(**node**), outArcs(**node**). *GuardGen* is an unspecific function. It must be replaced with the special rule of the specific model.

In the next step the user defines the specific business process concepts. A generator is based on the concepts of such concepts. In our example the user defined that BPMN consists Tasks and Gateways. A BPMN diagram consists of Tasks which are derived from Actions and a statechart diagram consists of statechart actions (stchaction). The rule is applied on all actions of a diagram.

```
//Start : WFTransGen(generatedfromWFTransGen)
forall node in Objects with type(node) = activity
  print
    "WorkFlowTransition_node(pi) =
    //End : WFTransGen
    //Start : GuardGen(generatedfromGuardGen)
    if GuardGen then
      if ReadyForExec(node, pi) then
        let t = fringToken(inArcs(node)
          Consume(t, inArcs(node))
          Produce(t, outArcs(node))//End : GuardGen
```

Figure 3: General ASM Generator rule

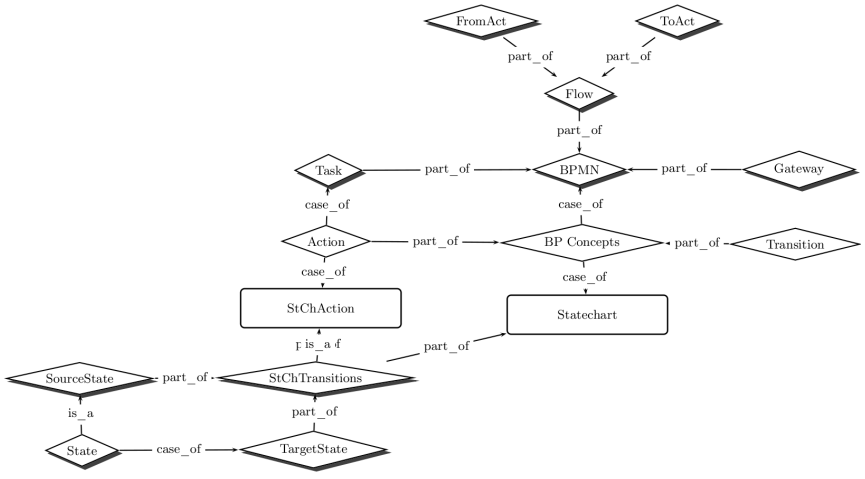

Figure 4: Cut-Out of Specific Business Process Model Concepts

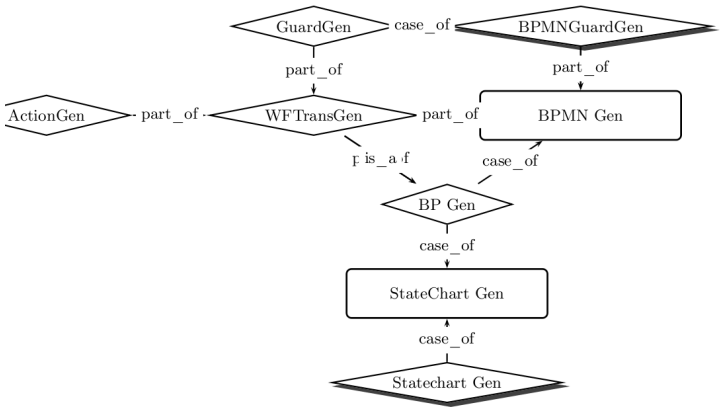

Figure 5: Cut-Out of Specific Business Generator Concepts

forall node in *Action*
  print"WORKFLOWTRANSITION_**node**(pi) =
  if *enabled*($''$+*inArcs*(**node**)"+, pi) **then**
    if *ReadyForExec*(**node**, pi) **then**
      let t = *fringToken*(*inArcs*(**node**)
        par
          Consume(t, *inArcs*(**node**))
          Produce(t, *outArcs*(**node**))$''$

Figure 6: Specific ASM Generator rule for BPMN

The user has to define which concepts of the specific model is mapped on which concept of the general concept. In our example a tasks is derived from a actions. And the user has to enter the parts of the generator which are defined as stubs in the general ASM Generator rule.

Figure 6 presents the specific ASM generator rule for a BPMN concept. We enter for the stub "enabled(inArcs(**node**) , pi)".

Statecharts are defined about concrete state names. Thus we have to recognize this with a specific rule:

In the next step the user can draw a concrete business process, in our example the developer writes a specification with a well-defined subset of natural language, here represented with XML. From this specification he generates the BPMN diagram or he starts with drawing a BPMN diagram of a ordering process (see figure 8). The user has to use a fixed grammar and vocabulary.

```
<IF>
  <object thing="process" nameOfTheThing="order process"/>
  <verb name="executed"/>
   <THEN>
```

```
forall node in Action
  print"WORKFLOWTRANSITION_node(pi) =
  if state(''+statename(node)"+,pi) then
    if ReadyForExec(node,pi) then
      let t = fringToken(inArcs(node)
        par
          Consume(t,inArcs(node))
          Produce(t,outArcs(node))''
```

Figure 7: Specific ASM Generator rule for Statecharts

```
    <object thing="process" name_of_the_thing="verify stock"/>
        <auxiliary_verb name="can be"/><verb name="executed">
</THEN</IF>
```

We do not present the mapping rules between the concepts of the drawing tool (e.g. XMI) and the concepts which the user defined for the generator.

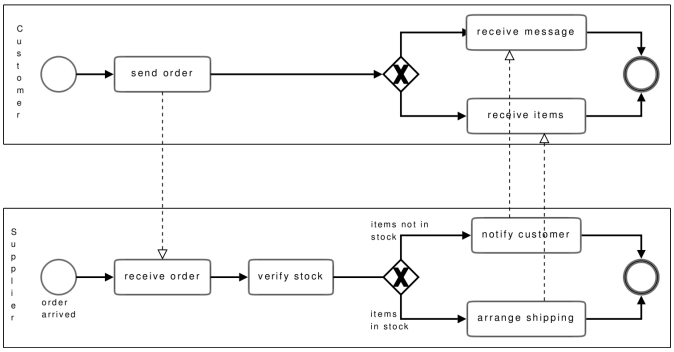

Figure 8: BPMN model for the ordering process

Then we generate the requirements specification for the end user in natural language. "IF the process order process is executed then the process verify stock can be executed."

If we apply our generators on BPMN concepts we would get the ASM code and on Statechart concepts depicted in figure 9

The domain in-dependend model must be instantiated. We call such a instantiated meta model: Merge-Concept as it is a merge between the domain in-dependend model and words from the domain dependend vocabulary like receive order, verify stock. The concepts of the instantiated BPMN model are represented in figure 10.

BPMN :
WORKFLOWTRANSITION_VERIFY_STOCK(pi) =
if *enabled(receive order, pi)* **then**
  if *ReadyForExec*($verify\_stock, pi$) **then**
    let $t = fringToken(transition_2)$
      par
        $Consume(t, transition_2)$
        $Produce(t, transition_3)$

Statechart :
WORKFLOWTRANSITION_VERIFY_STOCK(pi) =
if *state(order received, pi)* **then**
  if *ReadyForExec*($verify\_stock, pi$) **then**
    let $t = fringToken(transition_2)$
      par
        $Consume(t, transition_2)$
        $Produce(t, transition_3)$

Figure 9: Specific ASM Generator rule for BPMN and Statecharts

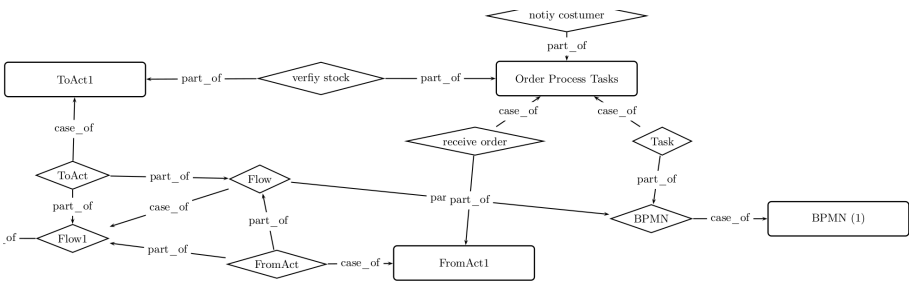

Figure 10: Cut-Out of Specific Business Generator Concepts

5 Future Direction and Conclusion

An important future direction for this work is to use concepts for database transactions and the models of cooperative view update to combine it with business processes [HS07]. For each task of the BPMN diagram we derive a own database schema defined by components and a requested update on one of these components. When developers change a BPMN diagram, resp. the semantics in the form of transformers simultaneously, we need to propagate these changes across all parts of this BPMN diagram to guarantee them consistent. The process of synchronization propagates changes among specifications in different stages to all involved participants. Exchange of models between local platforms is still a challenging issue. The exchange and the synchronization BPMN diagrams of different local copies between local development systems is so far a tough problem. To support the decision making requirement for a valid BPMN diagram [ZKL06] and the ideas of a cooperated drawing of business diagrams we have to implement an cooperated update automaton proposed in [HS07]. Additionally we have to include the approach of Model Engineering [Tha08] as typical system specification consists of a number of models for different facets and aspects of the system. That means we have to figure out how data-centric specifications may be cooperated with BPMN diagrams. To realize that we have to develop transformer which carry out the semantics of BPMN diagrams and the semantics of data-centric specifications.

For each generator for a new concept (resp. a program code, database scripts) we have to define generator profiles. Such profiles are depend upon the used concept, the target language (e.g. ASM, Java, SQL), a user and other constraints. We are able to generate a XML output document on several concepts. For instance we are then able to define XSL-templates to transform concepts. An advantes is that we have a management tool for all defined generators.

```
- Java (target language concept)
  - THEN (concept)
    - <xsl:template match="THEN"> ...</xsl:template>
  - IF (concept)
    - <xsl:template match="IF"> ...</xsl:template>
```

A usage scenario illustrated in this paper shows a way to organize concepts of overlapping meta-models for business process modeling. The main contribution of this scenario is the specification of transformation rules from similar business process models towards executable specifications. We presented how Concept-Driven Engineering uses and benefits from model transformation to provide a re-usage of software artefacts and rapid prototyping. Our approach reduce the development time for software products and increase their quality.

In this paper we presented the Concept-Driven Engineering approach to support the development of business process models, resp. notations. We presented that it is helpful to derive business process models, resp. notations on the basis of common concepts to support the re-usage of concepts. A most important re-usage is the re-usage of transformation rules. On the one hand we can test the underlying semantics of a BPMN diagram (notated as domain independent concepts), on the other hand we can validate the domain dependend concepts on which a business process models, resp. notations are based. To realize a rapid prototyping we explained the idea of the transformer which generated originating from source concepts an executable specification (ASM).

Acknowledgment

The ConceptManager tool presented in this paper has been developed as a part of the "FoPra" (advanced software lab) 2007/08 by the Information Systems Engineering group, Christian-Albrechts-Universität zu Kiel. The project was supervised P.Schmidt, development was conducted by M.Behrens, S.Bolus, S.Knaur, S.Kowski and F.Kramer. The authors would like to acknowledge the fruitful discussions with Bernhard Thalheim.

References

[atl] http://www.eclipse.org/m2m/atl/atlTransformations/.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. 2005. Technical Report 05-14. http://www.cs.utexas.edu/ftp/pub/techreports/tr05-14.pdf.

[BB08] Don S. Batory and Egon Börger. Modularizing Theorems for Software Product Lines: The Jbook Case Study. *J. UCS*, 14(12):2059–2082, 2008.

[ble] http://e2-productions.com/repository/.

[Bör07] Egon Börger. Modeling Workflow Patterns from First Principles. In Christine Parent, Klaus-Dieter Schewe, Veda C. Storey, and Bernhard Thalheim, editors, *ER*, volume 4801 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2007.

[BT08] E. Börger and B. Thalheim. Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach. In *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 24–38, Berlin, Heidelberg, 2008. Springer-Verlag.

[con] http://en.wiktionary.org/wiki/concept.

[FF08] Sven Feja and Daniel Fötsch. Model Checking with Graphical Validation Rules. In *ECBS*, pages 117–125. IEEE Computer Society, 2008.

[FLZ05] Peter Fettke, Peter Loos, and Jörg Zwicker. Business Process Reference Models: Survey and Classification. In *Business Process Management Workshops*, pages 469–483, 2005.

[HS07] S.J. Hegner and P. Schmidt. Update Support for Database Views Via Cooperation. In Y. E. Ioannidis, B. Novikov, and B. Rachev, editors, *ADBIS*, volume 4690 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2007.

[IM08] Antonio Ruiz-Cortés Ildefonso Montero, Joaquín Peña. ATL Transformation: Feature Models for representing runtime variability in BIS to Business Process Model Notation. 2008. http://www.isa.us.es/uploads/tools/fm2bpmn/doc/draft.pdf.

[mer] http://en.wikipedia.org/wiki/Mereology.

[mod] http://1828.sorabji.com/1828/words/m/model.html.

[Pia52] J. Piaget. New York : International Universities Press, c1952., 1952.

[PSC01] Elke Pulvermüller, Andreas Speck, and James Coplien. A Version Model for Aspect Dependency Management. In Jan Bosch, editor, *GCSE*, volume 2186 of *Lecture Notes in Computer Science*, pages 70–79. Springer, 2001.

[Tha08] B. Thalheim. Model suites, 2008.

[ZKL06] O. Zimmermann, J. Koehler, and F. Leymann. The role of architectural decisions in model-driven SOA construction., 2006.