INFORMATICA, 2007, Vol. 18, No. 4, 585–602
© 2007 *Institute of Mathematics and Informatics, Vilnius*

Reusing Ontological Knowledge about Business Processes in IS Engineering: Process Configuration Problem

Donatas CIUKSYS
Faculty of Mathematics and Informatics, Vilnius University
Naugarduko 24, LT-03225 Vilnius, Lithuania
e-mail: donatas.ciuksys@mif.vu.lt

Albertas CAPLINSKAS
Institute of Mathematics and Informatics
A. Goštauto 12, LT-01108 Vilnius, Lithuania
e-mail: alcapl@ktl.mii.lt

Received: May 2006

Abstract. Business process engineering is an important part of the advanced enterprise engineering. One of the still open issues is the question how in the enterprise system design to reuse ontological knowledge about business processes. The paper proposes to consider a family of similar business processes as a generic process and to represent knowledge about generic processes in a domain independent way. It describes the main scheme for reuse of such a domain independent knowledge when developing enterprise-wide information systems (IS). The main attention is paid to the process configuration problem. In order to solve this problem, a configurator (human being or machine) must find a set of components that fit together to satisfy the problem specification. An approach based on Description Logics is proposed for this aim. The main contribution of the paper is the proposed process configuration technique.

Key words: business process, knowledge reuse, ontology, description logics.

Introduction

The issues of business process engineering are an important part of advanced enterprise engineering methodology. In particular, Business Process Management (Smith, 2003) and Service-Oriented Architecture (Erl, 2005) are increasingly gaining interest lately. Both approaches facilitate software reuse. However representing the knowledge about families of similar business processes and reusing this knowledge in enterprise engineering projects still remains an open problem. The paper proposes an approach how to deal with this problem. The proposed approach combines together reuse techniques developed in domain engineering (Czarnecki, 2000), knowledge engineering (Chandrasekaran, 1986), and ontology-based systems engineering (Davies, 2006). The main idea behind this approach is that the principle of separation of concerns (also known as "divide-and-conquer" strategy (Damaševičius, 2002)) should be used to separate process knowledge and domain knowledge. Two separate ontologies are proposed to describe this knowledge. As a result process ontology can be reused in different application domains. Similar approach is used in knowledge engineering where task knowledge and domain knowledge have been separated.

Reusing of process knowledge requires the process to be configured taking into account peculiarities of the particular application domain. In order to solve the process configuration problem, a configurator (human being or machine) must find a set of mandatory and optional process parts that fit together to satisfy the problem specification. It is one of the hardest problems in reusing of process knowledge.

The paper proposes a Description Logics (Baader, 2003) based approach to solve the process configuration problem. It is assumed that all process parts (mandatory and optional ones) are described by the process ontology. OWL (Web Ontology Language, Description Logics based ontology specification language) is used to represent the knowledge. OWL knowledge base (KB) consists of three separate parts: TBox, ABox and RBox. In our case, TBox contains definitions of concepts (process parts) and structural relationships (mostly part-of) between them, ABox contains the process parts chosen by process engineer as necessary for the resulting business process, and RBox contains descriptions of the dependencies between optional process parts in the form of so-called DL-safe rules (Motik, 2005). Model-based approach is used to check the consistency of KB. Some semantic reasoner should be used for this aim. The KB is consistent if the reasoner is able to built successfully at least one model of this base. In other words, the ABox should be built that contains all required parts of the resulting process. So, any model of the given KB is regarded as the solution to the process configuration problem. The model building process is iterative one and process engineer is allowed to control this process in an interactive mode. It means that he can to add/remove optional process parts on the fly and re-check KB consistency.

The rest of the paper proceeds as follows. Section 1 discusses the notion of generic business process. Sections 2 and 3 introduce engineering of process domain and process engineering respectively. Reusable upper-level ontologies are proposed in Sections 4–6. Section 7 is dedicated to the analysis of process configuration problem and its possible solution using Description Logics (DL). Finally, paper ends with conclusions.

The paper develops further, refines and improves ideas proposed in (Caplinskas, 2004; Ciuksys, 2006).

1. Notion of a Generic Business Process

A business process is a partially ordered set of linked activities that create value by transforming an input into a more valuable output. Both input and output can be artefacts and/or information and the transformation can be performed by human actors, machines, or both. A generic business process is an abstraction of a family of similar business processes. All members of this family include a set of common core parts (commonalities) and each particular member includes some additional parts (variabilities), which may differ for different members of the family. A generic business process is described by a kind of feature model (Kang, 1990) and by ontology. The feature model can be seen as a view of generic process ontology (Czarnecki, 2006). The generic business process does not include any control knowledge about the sequencing of business activities. Control knowledge is added later, reusing process ontology in a particular application domain.

We suggest that generic processes should be expressed in terms of abstract roles (actors, inputs, outputs, resources, capabilities). Generic processes are used to generate particular processes (i.e., members of family) that then are *located* in chosen application domain. The purpose of generation is to produce the required configuration of the process or, in other words, to decide which variabilities are not relevant to this particular member of family and reject them. After that, roles should be replaced by the entities of application domain in which this member of process family is located. We call this activity *role assignment*.

Thus, the proposed approach provides two main activities: *engineering of process domain* and *process engineering*. The term "*process domain*" is used here to denote a group of particular processes that exhibit similar behaviour and are used to achieve similar goals. Indeed, it is a synonym for the term "generic business process".

2. Engineering of Process Domain

Engineering of process domain is an activity that is analogous to the domain engineering activity in the two life cycles model (Czarnecki, 2000). Similarly as *metaprogramming* is used to manage variability in a domain, develop generic domain software components, and describe generation of customised software component instances (Czarnecki, 2000; Štuikys, 2004), we use *metamodelling* to manage variability in a process domain, model generic processes and describe generation of particular processes. The purpose of process domain engineering is to develop particular process domain. This activity includes three sub-activities referred as analysis, design and implementation of process domain (Fig. 1).

Analysis of process domain provides domain scoping (definition of the boundaries of process family) and discovering commonalities and variabilities among the processes in this domain. The result of analysis is a feature model that describes variabilities and commonalities within business process family. Design of process produces generic business process ontology. It refines terms defined by feature model and adds to ontology epistemic knowledge. It is important to point out that the resulting ontology is based on

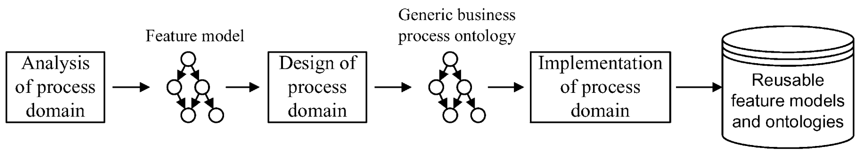

Analysis of process domain
Feature model
Design of process domain
Generic business process ontology
Implementation of process domain
Reusable feature models and ontologies

Fig. 1. Engineering of process domain.

the upper business process ontology defining concepts such as *activity*, *input*, *output*, *resource*, *capability*, etc. required to model any generic business process. Upper business process ontology is described in detail in Section 6.

The purpose of implementation of process domain is to create reusable assets from feature model and process ontology. The process ontology as a reusable asset is represented using Web-Ontology Language (OWL) (W3C, 2004).

3. Process Engineering

Process engineering is an activity that is analogous to the application engineering activity in the two life cycles model (Czarnecki, 2000). Its purpose is to generate a particular business process and to locate it in a chosen application domain. Process engineering starts with two parallel activities – analysis of application domain and configuration of generic business process (Fig. 2). The result of analysis is application domain ontology. This ontology is based on the upper application domain ontology that defines concepts required to model application domains, such as *active entity* (e.g., job position, application system, organisational unit, etc.), *provided capabilities* possessed by active entity, *passive entity*, *state*, etc. Process configuration rejects variable parts of the process that are not relevant for chosen application domain and produces final configuration of the located process. A software tool (configurator) is used to support this activity. The main responsibility of this tool is to prevent violation of dependencies between variants of feature model.

The configured process and application domain ontology are inputs for the next step, role assignment. Business process is still described in terms of roles (actors executing process's activities). Requirements for actors that can pretend to play these roles are expressed in form of *required capabilities*. Application domain ontology defines active entities and their *provided capabilities*. So, both roles and entities are characterised in terms of capabilities. Role assignment is done by matching required capabilities to provided capabilities. If an active entity is too coarse-grained for business process role (provided capabilities subsume required capabilities), this entity must be re-engineered and splitted into several more fine-grained entities (so called *capability specialisation*). If an entity is

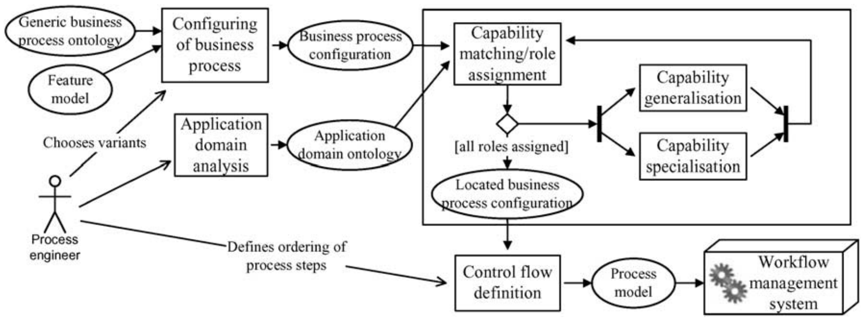

Generic business process ontology
Feature model
Configuring of business process
Business process configuration
Capability matching/role assignment
Capability generalisation
Capability specialisation
[all roles assigned]
Located business process configuration
Chooses variants
Application domain analysis
Application domain ontology
Process engineer
Defines ordering of process steps
Control flow definition
Process model
Workflow management system

Fig. 2. Business process knowledge reuse – process engineering.

too fine-grained (provided capabilities are subsumed by required capabilities), then usually there will be some number of them and, in this case, these entities have to be composed to one, more coarse-grained, composite entity (so called *capability generalisation*). If no active entity candidates to play some role, a new entity must be created, for example, by employing a new person or by developing a new application system (not shown in Fig. 2). Iteration "role assignment – capability specialisation/generalisation" is being repeated until all roles are assigned. The result is located business process. However, the control knowledge is still undefined. Next activity "control flow definition" adds control knowledge that defines execution order of business activities. As a result executable business process model is produced. It is described in WS-BPEL language (OASIS, 2007) and can be executed by some workflow management system. This system orchestrates execution of business process activities and at certain times (as defined in business process model) requests services provided by appropriate active entities. Capabilities provided by application systems are requested through Web Services interfaces. Human service providers are requested through special user interfaces (UI). They are informed about pending activities that must be performed.

4. Upper-Level Ontology

Application domain ontology captures domain knowledge independently of its use. Both application domain ontology and process ontology should be described by some common system of metaconcepts. It means that some higher-level ontology is required. We call this ontology upper-level ontology (Fig. 3).

This ontology introduces generic concepts that are shared by all lower-level ontologies and reflect underlying theory about the nature of enterprise's social reality (discourse of interest). Our upper level ontology has been influenced strongly by Uschold's enterprise ontology (Uschold, 1998). The most important difference between Uschold's and our ontologies is that descriptions of roles in our ontologies allow specifying states that limit

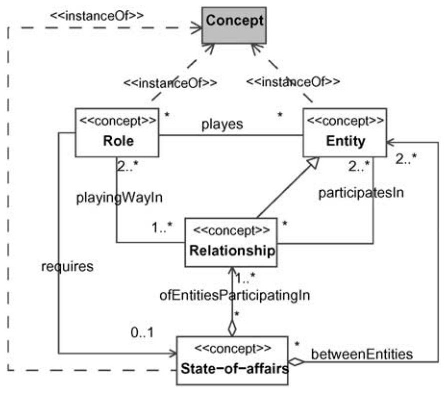


Fig. 3. Upper-level ontology.

possible set of entities that may candidate to play such a roles. Entities allowed to perform such a role must support defined states.

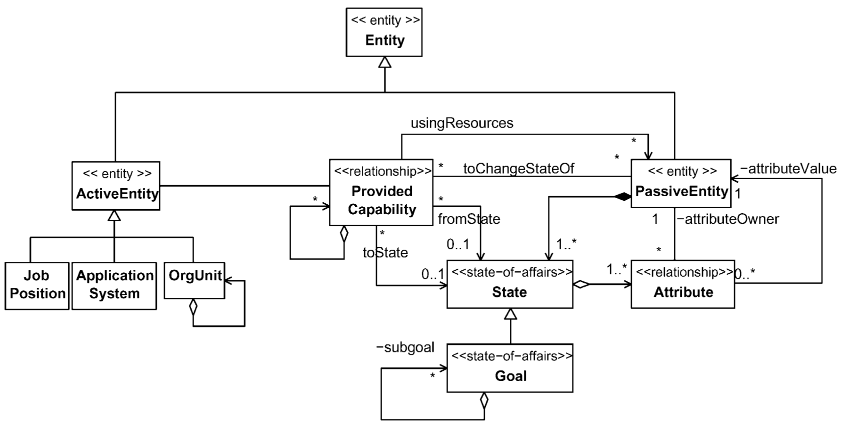
Upper-level ontology (Fig. 3) is two-level ontology. The top level provides the only concept "*Concept*" that is used to define second level concepts "*Entity*", "*Relationship*", "*Role*", and "*State-of-affairs*". These concepts, in turn, are used to define third level concepts in application domain ontology and in process ontology. It means that we follow scheme provided by MOF standard (OMG, 2006). According to this approach, instances of metaconcepts are concepts themselves.

In this paper upper-level ontology and others, described below, are represented using informal UML-like diagrams. Process engineering tool deals with ontologies specified in formal OWL language.

5. Upper Application Domain Ontology

Let us consider now the upper application domain ontology that serves as a basis to define concepts in particular application domain ontologies (Fig. 4).

All concepts defined by this ontology are instances of concepts defined by upper-level ontology. The ontology refines the notion of entity and classifies all entities into: *active entities* and *passive entities* (Fig. 4). They may overlap. Active entities must provide capabilities required to achieve some business goals or subgoals. Business goal is a state of passive entity that candidates to play output role in some business activity. Such an organisation of concepts is introduced in order to facilitate role assignment. Active entities are further subtyped into *job positions*, *application systems*, and *organisational units* ("OrgUnit" in Fig. 4). All of them possess provided capabilities and may candidate to play roles defined by process ontology. Being active entities they can change states of passive entities. For example, a job position may provide writing capability and consequently be able to prepare a document (change its state). Similarly an application system



<< entity >> Entity
usingResources
<< entity >> ActiveEntity
<<relationship>> Provided Capability
toChangeStateOf
<< entity >> PassiveEntity
-attributeValue
-attributeOwner
fromState
toState
0..1
1..*
Job Position
Application System
OrgUnit
<<state-of-affairs>> State
<<relationship>> Attribute
0..*
-subgoal
<<state-of-affairs>> Goal

Fig. 4. Upper application domain ontology.

may provide order-processing capability and be able to change the state of order from unprocessed to processed one.

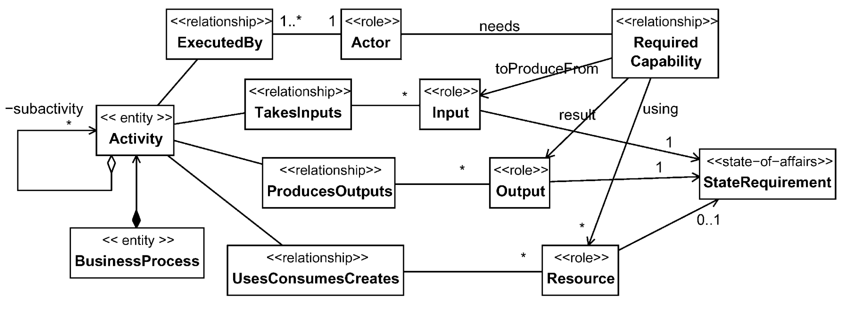
Finally, concept "*goal*" models business goals. They form a hierarchy. It means that we make assumption that an application domain (as a specific area of business) should explicitly state business goals. These goals must be achieved to ensure successful operation of enterprise in this business area.

6. Upper Process Ontology

Up until very recently there was no widely accepted and standardised business process conceptualisation. Common approach was to develop a new business process conceptualisation each time when a new business process related project was started or a new tool was developed. In 2003 OMG consortium announced an initiative that aims to standardise the conceptualisation of business processes and to develop so called Business Process Definition Metamodel (BPDM). The draft that candidates to be the final submission is already prepared (OMG, 2007). It describes following groups of concepts:

1. course model: introduces control flow concepts, such as *transition*, *gateway*, *fork*, *join*, etc.;
2. activity model: introduces structuring concepts, such as *process*, *activity*, *sub-activity*, etc.;
3. interaction protocol model: introduces interaction and data flow concepts, such as *interaction* and *data* (documents) being exchanged with these interactions;
4. event model: introduces concepts, describing events that happen during the course of business process, such as *start*, *finish*, *error*, *abort*, etc.

BPDM defines more than 100 concepts. Our upper process ontology is subset of BPDM. Included are only those concepts, that describe all kinds of roles provided by business processes (Fig. 5). *Actors*, *inputs*, *outputs* and *resources* are all modelled as roles. Domain entities must be assigned to these roles when business process is located in a particular application domain: active entities may candidate to actor roles, passive entities – to input, output and resource roles.



<<relationship>> ExecutedBy
1..*
1
<<role>> Actor
needs
<<relationship>> Required Capability
toProduceFrom
<<relationship>> TakesInputs
*
<<role>> Input
result
using
-subactivity
*
<< entity >> Activity
1
<<state-of-affairs>> StateRequirement
<<relationship>> ProducesOutputs
*
<<role>> Output
1
0..1
<< entity >> BusinessProcess
<<relationship>> UsesConsumesCreates
*
*
<<role>> Resource

Fig. 5. Upper process ontology (process conceptualisation).

These concepts are not sufficient to represent variabilities provided by process feature model. So, concepts such as *variability*, *variant*, *variation point,* etc. must be included into upper process ontology.

Our conceptualisation of variability is based largely on a general model of variability in product families proposed by Becker in (Becker, 2003). Simplified version of this model is shown in (Fig. 6). Let us discuss this model shortly. *Variability* represents a capability to change or adapt system (Gurp, 2001). In our context, process with variabilities can be adopted in various application domains. According to Becker, "a *variation point* is a spot in a software asset where variation will occur" (Jacobson, 1997; Gurp, 2001; Becker, 2003). *Variability* specifies a set of variation points and a set of *variants*, that define the extent of variability (i.e., the higher the number of variants, the higher the degree of system's adaptability). A *variability resolution* is a choice of suitable variant. This choice must be performed for each of variabilities. *Dependencies* constrain choices of variants, for example, choice of one variant may require choice or removal of the other one (e.g., the choice of payment type "Credit card" within e-shop business process may render optional activity "Connect with bank" as required). Final choices of variants are saved in *derivation profile*. Chosen variant must be integrated within system at the variation point. The *implementation mechanisms* specify techniques how to do this. These techniques depend on the kind of variants and on the kind of assets that contain variation points.

By fixing particular variability occurrence context, we can give more specific interpretation to variability concepts. In the case of business processes variabilities may occur in inputs, outputs and activities. However, for the matter of simplicity, we consider the variations that occur only in activities. We will use the term *generic activities* to refer to activities with variabilities. The reason for this simplification is that majority of process variabilities are found namely in activities. Therefore generic activities are the only kind of variation points that we consider or, in other words, the concept *GenericActivity* specialises the concept *Variant* (Fig. 7). Both *Activity* and *GenericActivity* specialise the

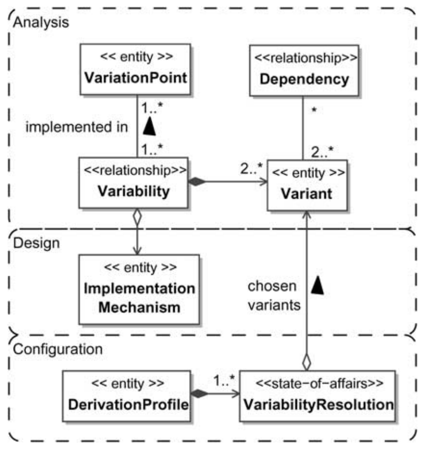


Fig. 6. Variability conceptualisation.

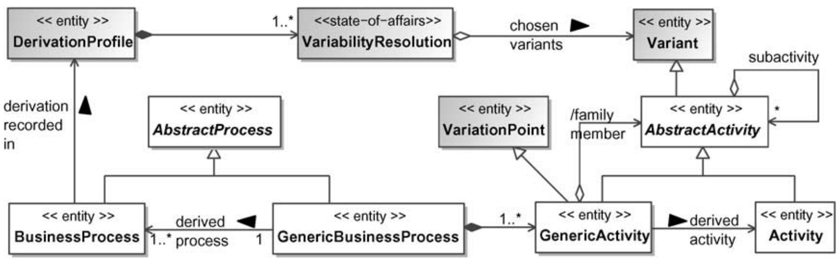


Fig. 7. Process variability conceptualisation.

concept *AbstractActivity*. This allows them to be interchangeable, that is, during process domain analysis process engineer will replace some activities by generic ones, and during process configuration generic activities will be replaced by non-generic activities, which comprise the chosen variant. As a consequence we have only one kind of variants, namely abstract activities, i.e., the concept *AbstractActivity* extends the concept *Variant* (Fig. 7).

Generic activity represents the family of activities. This family may consist of simple as well as generic activities. As a consequence we can build hierarchies of generic activities. We conceptualise this ability as an aggregation association between generic activity and abstract activity (Fig. 7). Thus abstract activity, activity and generic activity instantiate Composite design pattern.

In process domain design step process engineer has to specify how activities will be derived from generic activities (i.e., how variability will be resolved), that is, choose variability *implementation mechanisms*. Puhlmann discusses following variability implementation mechanisms suitable for business processes (Puhlmann, 2005):

1. *Parameterisation* allows introducing the variability to the behaviour of the generic activity by putting parameters in certain places. Then, we can resolve the variability by assigning values to these parameters.
2. *Inheritance* allows for the replacement of generic activity by specialised one. Specialised activity must conform to the interface of the generic activity (at least inputs and outputs must match) and usually provides additional behaviour.
3. *Design Patterns* based on information hiding and inheritance like the Strategy design pattern can be represented in processes using inheritance. For example, Strategy design pattern comprises so called *abstract activity* that defines interface (inputs and outputs) and several implementing activities. One of these activities must be chosen and will replace the abstract activity.
4. *Extensions/Extension Points*. Extension points are places where the process can be extended with additional behaviour. They may be represented by so called *null activities* – activities without behaviour. Extensions (non empty activities) may be chosen to replace the null activity. An extending activity must have a compatible interface in order to be integrable into the process at the corresponding extension point.

The configurator guided by business process analyst should be able to resolve all variabilities (i.e., to choose exactly one variant for each of the variabilities) and integrate chosen variants into process using appropriate variability implementation mechanisms. As a result, a description of fully configured process is produced (i.e., without variabilities) and stored in *derivation profile*.

7. Process Configuration

7.1. *Process Configuration Problem*

In the proposed approach a generic business process should be described by a kind of feature model (Kang, 1990) that, as stated above, is a view of upper process ontology. Fig. 8 shows example of a feature model for ordering process of e-shop. Ordering process in this example includes three mandatory parts (*Basket*, *Transaction* and *Fulfilment*) and one optional part – *Approval*. A generic activity *Basket* has a variation point that specifies two exclusive (XOR) alternatives – the variants *Temporal* and *Persistent* – one of which must be selected. A generic activity *Payment* has three *range* (OR) variants – *PayByBill*, *PayOnDelivery,* and *CreditCard*. Any non empty group of them must be selected as payment type. Besides, three dependencies between variants are defined in this example. Choice of activity *Shipping* requires choosing activity *PrintedInvoice* and choice of activity *ElectronicDelivery* requires choosing activities *CreditCard* and *OnlineDisplay*. These dependencies are examples of *requires* dependencies. In addition, feature models often have *mutually exclusive* dependencies, when choice of one variant forbids choosing other variants.

So, business process configuration should be designed that satisfies all mentioned dependencies. Configuration design problem is a problem where a set of pre-defined components is given and an assembly of selected components should be find that satisfies a set of requirements and obeys a set of constraints (Wielinga, 1997). In order to solve the configuration problem, a configurator (human being or machine) must find a set of components that fit together to satisfy the problem specification. Typically, it means that

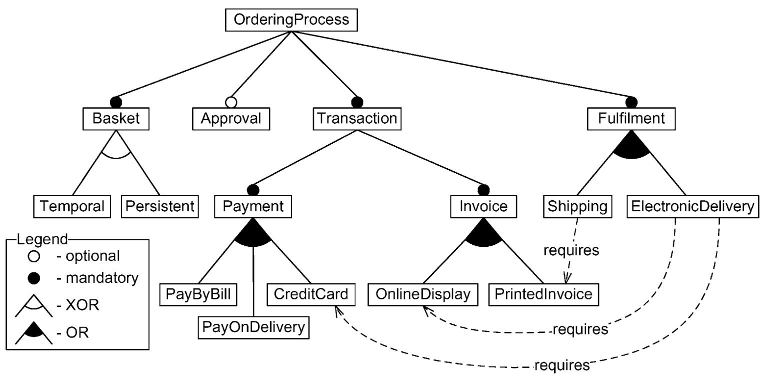


Fig. 8. Example – feature model for ordering process.

the solution will be arrangement of components, which satisfy all constraints provided by the specification.

In our case, treating the activities as components and the dependencies as constraints, we are dealing with the configuration design problem. The aim is to resolve all variabilities and to obtain such configuration of the business process that satisfies dependencies stated by the feature model.

Similarly as in (Czarnecki, 2006), we consider feature model as a view of generic process ontology. It means that feature model generally shows only those activities that have variabilities (i.e., generic activities). We argue that it is purposeful to consider feature model as a view of ontology because in this case the feature model hides unnecessary information and consequently it can be more effectively processed by configurator's software. The main purpose of the configurator is to guide process engineer through selection of variants until all variabilities will be resolved. This tool should also prevent violation of dependencies between variants of feature model.

A number of approaches including constraints, expert systems, model-based reasoning, case-based reasoning, Propose, Critique and Modify (PCM) methods, hierarchical methods, binary decision diagrams (BDD) (Bryant, 1986) and Description Logics (Baader, 2003) have been proposed to solve the configuration problem. These methods are shortly discussed in (Wielinga, 1997) and in (Baader, 2003). We use an approach based on Description Logics, because we define all proposed ontologies using OWL DL.

7.2. *Description Logics and language $\mathcal{SHOIN}(\boldsymbol{D})$*

Description Logics (DL) is a family of concept-based knowledge representation formalisms. DL formalisms provide reasoning apparatus that emphasises the decidability of key reasoning problems. Any DL knowledge base includes two components: the TBox and the ABox. The first one is used to define the hierarchy of concepts of application domain, more exactly, a lattice-like structure of concepts. Concepts denote sets of individuals. The formalism also allows defining roles of concepts. Roles denote binary relationships between individuals. There are *elementary* and *complex* descriptions. Elementary descriptions define *atomic concepts* and *atomic roles*. Complex descriptions are built inductively applying *concept constructors* (union, intersection, negation, etc.). Different DL formalisms (i.e., description languages) differ mainly by the constructors they provide. Any description language allows building complex descriptions of concepts and roles and assigning names to these descriptions (Baader, 2003). It has a model-theoretic semantics. Thus, TBox is used to define intensional knowledge about the application domain.

The ABox is used to define extensional knowledge or, in other words, knowledge that is specific to the individuals of the domain of discourse (Baader, 2003). Extensional knowledge often is referred also as assertional knowledge because the knowledge about the individuals is presented in the form of assertions.

DL allows reasoning about concepts, individuals and assertions. For example, it is possible to determine whether a description is *satisfiable* (i.e., non-contradictory), or whether one description *subsumes* another one. It is possible also to check whether assertions defined in ABox are *consistent* (i.e., whether the set of assertions has a model), and whether a particular individual is an instance of a given concept description. Such reasoning helps to determine whether a knowledge base is meaningful at all. The reasoning also can be used to check subsumption of given concepts (Baader, 2003).

We use $\mathcal{SHOIN}(\mathbf{D})$ description language. This language is at the core of Web Ontology Language OWL DL (Horrocks, 2003). It belongs to the family of so-called $\mathcal{AL}$-*languages*. All languages belonging to this family are some extensions of $\mathcal{AL}$ (attributive language) introduced in (Schmidt-Schauß, 1991). It is a minimal language that is of practical interest. The notation and Tarski-style semantics of $\mathcal{AL}$ is given in Fig. 9 where letters A and B are used for atomic concepts, the letter R for atomic roles, the letters C and D for concept descriptions, and the letter $\mathcal{I}$ for interpretation function (Baader, 2003).

To give an example of what can be expressed in $\mathcal{AL}$, let's assume that *Activity* and *Generic* are atomic concepts. Then *Activity* $\sqcap$ *Generic* and *Activity* $\sqcap$ $\neg$*Generic* are $\mathcal{AL}$-concepts describing, intuitively, generic activities and not generic ones. If, in addition, we suppose that *hasSubactivity* is an atomic role, we can form the concepts *Activity* $\sqcap$ $\exists$*hasSubactivity*.$\top$ and *Activity* $\sqcap$ $\forall$*hasSubactivity*.*Generic*, denoting those activities that have a sub-activity, and those activities all of whose sub-activities are generic, correspondingly. Using the bottom concept, we can also describe those activities without any sub-activities by the concept *Activity* $\sqcap$ $\forall$*hasSubactivity*.$\bot$.

$\mathcal{SHOIN}(\mathbf{D})$ adds to the $\mathcal{AL}$ negation of arbitrary concepts, transitive roles, role hierarchies, nominals, inverse roles, number restrictions, and data types. It is sound, complete and decidable description language. The semantic of this language is described in (Horrocks, 2003).

According to (Tobies, 2001), the inference in $\mathcal{SHOIN}(\mathbf{D})$ (and in OWL DL as well) is of worst-case nondeterministic exponential time (NExpTime) complexity. A related logic, $\mathcal{SHIQ}(\mathbf{D})$ (Horrocks, 2000), distinguished from $\mathcal{SHOIN}(\mathbf{D})$ mainly by not supporting nominals (or named objects), is ExpTime-complete (Tobies, 2001). Inference in $\mathcal{SHIQ}(\mathbf{D})$ logic extended with DL-safe rules still is of deterministic exponential time (Motik, 2005).

$A \mid$	(atomic concept)			
$\top \mid$	(universal concept)	$\top^{\mathcal{I}}$	$=$	$\Delta^{\mathcal{I}}$
$\bot \mid$	(bottom concept)	$\bot^{\mathcal{I}}$	$=$	$\emptyset$
$\neg A \mid$	(atomic negation)	$(\neg A)^{\mathcal{I}}$	$=$	$\Delta^{\mathcal{I}} \backslash A^{\mathcal{I}}$
$C \sqcap D \mid$	(intersection)	$(C \sqcap D)^{\mathcal{I}}$	$=$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$\forall R.C \mid$	(value restriction)	$(\forall R.C)^{\mathcal{I}}$	$=$	$\{a \in \Delta^{\mathcal{I}} \mid \forall b : (a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$
$\exists R.\top$	(limited existential quantification)	$(\exists R.\top)^{\mathcal{I}}$	$=$	$\{a \in \Delta^{\mathcal{I}} \mid \forall b : (a,b) \in R^{\mathcal{I}}\}$

Fig. 9. Notation and Tarski-style semantics of basic Description Logics.

7.3. *Process Configuration Problem Solving: DL-based Approach*

According to (Klein, 1994), a solution of a configuration problem can be defined to be a model of the given DL-style knowledge base. In this case, the configuration space is defined by the TBox that describes both the concept hierarchy and the role hierarchy. Initial ABox includes mandatory variants only. Using this approach, the choice of optional, alternative or range variants entails that ABox is augmented with corresponding individuals and the OWL DL reasoner is asked to check whether ABox is consistent with respect to configuration space. The test succeeds if model of the knowledge base can be built. In the case of inconsistency the choice must be undone. It is important that in our case, the process configuration problem is solved in the interactive mode and that the problem solving process is iterative. It means that the user is not forced to resolve all the variabilities at once. In addition, the user can ask the configurator to resolve unimportant variabilities automatically and in this way minimise the number of required decisions. It is possible because consistency checking requires that a model must be build. In which way unresolved variabilities will be resolved depends on reasoning strategy used by the reasoner.

Let us demonstrate the proposed approach using an example (Fig. 10). The figure shows a part of configuration space for previously described ordering process configuration problem. Only generic activity *Fulfilment* and its two range variants *Shipping* and

$$\begin{aligned}
\text{FulfilmentPart} &\sqsubseteq (\text{Shipping} \sqcup \text{ElectronicDelivery}) \\
&\qquad \sqcap =1\ \text{part\_of} \sqcap \forall\ \text{part\_ of.Fulfilment} \\
\text{Shipping} &\sqsubseteq \neg\text{ElectronicDelivery} \\
\text{ElectronicDelivery} &\sqsubseteq \neg\text{Shipping}
\end{aligned} \tag{1}$$

$$\begin{aligned}
\text{has\_fulfilment\_part} &\sqsubseteq \text{has\_part} \\
\text{has\_shipping} &\sqsubseteq \text{has\_fulfilment\_part} \\
\text{has\_electronic\_delivery} &\sqsubseteq \text{has\_ fulfilment\_part}
\end{aligned} \tag{2}$$

$$\begin{aligned}
\top &\sqsubseteq \forall\ \text{has\_fulfilment\_ part.FulfilmentPart} \\
\top &\sqsubseteq \forall\ \text{has\_shipping.Shipping} \\
\top &\sqsubseteq \forall\ \text{has\_electronic\_ delivery.ElectronicDelivery}
\end{aligned} \tag{3}$$

$$\begin{aligned}
\text{Fulfilment} \sqsubseteq\ & \text{OrderProcessPart} \\
& \sqcap \forall\ \text{has\_part.FulfilmentPart} \\
& \sqcap \geqslant 1\ \text{has\_fulfilment\_part} \\
& \sqcap \leqslant 1\ \text{has\_shipping} \\
& \sqcap \leqslant 1\ \text{has\_electronic\_delivery}
\end{aligned} \tag{4}$$

Fig. 10. Configuration space (TBox) for Fulfilment configuration problem.

ElectronicDelivery are included. Furthermore, let $= 1R$ be an abbreviation for $\geqslant 1R \sqcap \leqslant 1R$. The TBox starts by giving so-called cover axiom for concept *FulfilmentPart*, as well as requiring *FulfilmentPart* to be part of and only of *Fulfilment*. Additional axioms ensure the disjointness of concepts *Shipping* and *ElectronicDelivery* (1). Then TBox introduces role hierarchy, stating that role *has_fulfilment_ part* is sub-role of role *has_part*, and both roles *has_ shipping* and *has_electronic_delivery* are sub-roles of role *has_fulfilment_part* (2). The TBox follows by imposing range restrictions for the roles (3). Finally requirements for concept *Fulfilment* are stated (4). It must be *OrderProcessPart* (not introduced here), all the parts it has must be instances of concept *FulfilmentPart*, and it must have at least one *FulfilmentPart*, at most one *Shipping*, and at most one *ElectronicDelivery*.

In our example the initial ABox is very simple: $A = \{f : Fulfilment\}$. If user would choose alternative variant *Shipping* (and resolves variability present in activity *Fulfilment*), ABox would become: $A = \{f : Fulfilment, s : Shipping, has\_shipping(f, s)\}$. The knowledge base would be tested for consistency and consequently user's choice would be validated.

Our knowledge base still misses dependencies between variants. As we can see in feature model (Fig. 8), choice of variant *Shipping* requires choosing variant *PrintedInvoice*. To capture such a constraint within our knowledge base we need *rules*. One of decidable and often used rule-based formalisms is function-free Horn rules (Motik, 2005). Example of Horn rule is: $hasParent(x, y)\ \&\ hasBrother(y, z) \rightarrow hasUncle(x, z)$. The left side of implication is called the *rule body*, the right side – the *rule head*. Description language OWL-DL was extended with Horn rules in (Horrocks, 2004), but this extension is undecidable (Horrocks, 2004). In (Motik, 2005) a decidable combination of OWL-DL with rules has been proposed, where decidability is obtained by restricting the rules to so-called DL-safe ones. In DL-safe rules, concepts and roles are allowed to occur in both rule bodies and heads as unary and binary predicates, but each variable of a rule is required to be bound only to individuals present in ABox (Motik, 2005). In other words, the above "Uncle" rule will be DL-safe, if reasoning will be performed over the set of ABox individuals only.

Let's add rules to knowledge base of our example. The rule that the choice of variant *ElectronicDelivery* requires choosing variants *OnlineDisplay* and *CreditCard* (Fig. 8) may be written as follows:

$$\begin{aligned} has\_electronic\_delivery(f, ed) \rightarrow\ & has\_online\_display(i, od) \\ & \&\ has\_credit\_card(p, cc) \end{aligned}$$

Here f, ed, i, od, p, cc are individuals of concepts *Fulfilment*, *ElectronicDelivery*, *Invoice*, *OnlineDisplay*, *Payment* and *CreditCard* respectively. Similarly all *requires* dependencies may be encoded as rules. *Mutually-exclusive* dependencies are encoded by adding negation to all roles in rules head.

The presence of rules in the knowledge base means that reasoner must be capable to understand them and execute them as needed.

8. Conclusions

The proposed approach demonstrates that it is possible in the context of enterprise engineering to separate the ontological knowledge about business processes, generalise this knowledge and present it in a reusable form. Business process ontology should be build for this aim. This ontology should be based on the small number of very abstract upper-level concepts. We argue that it is purposeful to use for this aim the meta-ontology of Uschold's enterprise ontology (Uschold, 1998). It is important that Business Process Definition Metamodel (OMG, 2007) also is consistent with this meta-ontology. So, concepts required to model business processes can be borrowed from BPDM. However BPDM does not provide any concepts required to model variabilities of generic business processes and should be augmented in an appropriate way. We argue that a framework of terminology and concepts regarding variability proposed in (Gurp, 2001) can be used for this aim successfully.

In order to reuse ontological knowledge about business processes the configuration of the resulting business process should be designed. Most natural way to deal with configuration design problem stated in terms of ontological knowledge is to use model-based reasoning in Description Logics. Although the complexity of this approach is exponential one, optimisation techniques developed in the recent decade proved to be effective enough to build reasoners that can successfully deal with KB describing tens of thousands of concepts and individuals (Baader, 2003) that is fully sufficient to describe any reasonable business process.

One more problem that should be solved in order to reuse ontological knowledge about business processes is the role assignment problem. The paper only sketches shortly the method of solution of this problem. The details of this method are out of the scope of this paper and will be described in further works.

References

Baader, F., D. Calvanese, D.L. Mcguinness, D. Nardi and P.F. Patel–Schneider (2003). *The Description Logic Handbook*; *Theory, Implementation, and Applications*. Cambridge University Press.

Becker, M. (2003). Towards a general model of variability in product families. In J. van Gurp, J. Bosch (Eds.), *Proceedings of the 1st Workshop on Software Variability Management, Groningen, The Netherlands*. pp. 19–27.

Bryant, R. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, **C-35**(8), 677–691.

Caplinskas, A., and D. Ciuksys (2004). Ontologies, knowledge reuse and domain engineering techniques in information system engineering. In O. Vasilecas, A. Caplinskas, W. Wojtkowski, W.G. Wojtkowski, J. Zupancic, S. Wrycza (Eds.), *Proceedings of the Thirteenth International Conference on Information Systems Development*, Vol. 1. Technika, Vilnius. pp. 264–270.

Chandrasekaran, B. (1986). Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert: Intelligent Systems and their Applications*, **1**(3), 23–30.

Ciuksys, D., and A. Caplinskas (2006). Modelling of reusable business processes: an ontology-based approach. In A.G. Nilsson, R. Gustas, W. Wojtkowski, W.G. Wojtkowski, S. Wrycza, and J. Zupancic (Eds.), *Advances in Information Systems Development*, Vol. 1. Springer. pp. 71–82.

Czarnecki, K., and U. Eisenecker (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional.

Czarnecki, K., C.H.P. Kim and K.T. Kalleberg (2006). Feature models are views on ontologies. In *Proceedings of the 10th International on Software Product Line Conference*. IEEE Computer Society, Washington. pp. 41–51.

Damaševičius, R., and V. Štuikys (2002). Separation of concerns in multi-language specifications. *Informatica*, **13**(3), 255–274.

Davies, J., R. Studer and P. Warren (2006). *Semantic Web Technologies: Trends and Research in Ontology-based Systems*. Wiley.

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River.

Van Gurp, J., J. Bosch, M. Svahnberg (2001). On the notion of variability in software product lines. In R. Kazman, P. Kruchten, C. Verhoef and H. van Vliet (Eds.), *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*. IEEE Computer Society Press. pp. 45–54.

Horrocks, I., and P.F. Patel–Schneider (2004). A proposal for an owl rules language. In *Proceedings of the 13th International Conference on World Wide Web*, ACM, New York. pp. 723–731.

Horrocks, I., P.F. Patel–Schneider and F. van Harmelen (2003). From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, **1**(1), 7–26.

Horrocks, I., U. Sattler and S. Tobies (2000). Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, **8**(3), 239–263.

Jacobson, I., M. Griss, P. Jonsson (1997). *Software Reuse – Architecture, Process and Organisation for Business Success*. ACM Press/Addison-Wesley Publishing Co., New York.

Kang, K., S. Cohen, J. Hess, W. Novak and A. Peterson (1990). Feature-oriented domain analysis (FODA) feasibility study. *Technical Report CMU/SEI-90-TR-21*, SEI, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Klein, R., M. Buchheit and W. Nutt (1994). Configuration as model construction: The constructive problem solving approach. In *Proceedings Artificial Intelligence in Design '94*, Kluwer. pp. 201–218.

Motik B., U. Sattler and R. Studer (2005). Query answering for OWL-DL with rules. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, **3**(1), 41–60.

OASIS (2007). *Web Services Business Process Execution Language v2.0.* `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`

Object Management Group, Inc. (2007). *Business Process Definition MetaModel (BPDM).* `http://www.omg.org/cgi-bin/doc?bmi/2007-03-01`

Object Management Group, Inc. (2006). *Meta Object Facility (MOF) Core Specification*, version 2.0. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`

Puhlmann, F., A. Schnieders, J. Weiland and M. Weske (2005). Variability mechanisms for process models. *PESOA-Report No. 17/2005*. `http://www.pesoa.org/pages/Publications-en.html`

Schmidt-Schauß, M., and G. Smolka (1991). Attributive concept descriptions with complements. *Artificial Intelligence*, **48**(1), 1–26.

Smith, H. and P. Fingar (2003). *Business Process Management (BPM): The Third Wave*. Meghan-Kiffer Press.

Štuikys, V. and R. Damaševičius (2004). Soft IP customisation model based on metaprogramming techniques. *Informatica*, **15**(1), 111–126.

Tobies, S. (2001). Complexity results and practical algorithms for logics in knowledge representation. *PhD Thesis*, LuFG Theoretical Computer Science, RWTH Aachen, Germany.

Uschold, M., M. King, S. Moralee and Y. Zorgios (1998). The enterprise ontology. In M. Uschold and A. Tate (Eds.) *The Knowledge Engineering Review*, Vol. 13. Cambridge University Press, United Kingdom. pp. 31–89.

Wielinga, B. J., and A. Schreiber (1997). Configuration design problem solving. *IEEE Expert, Special issue on AI and design*, **12**(2), 49–56.

World Wide Web Consortium (2004). *OWL Web Ontology Language Guide*. `http://www.w3.org/TR/owl-guide/`

D. Ciuksys is a lecturer at the Vilnius University, Lithuania. His research area encompasses applying domain engineering techniques to business process knowledge reuse problem.

A. Caplinskas is a principal researcher and a head of the Software Engineering Department at the Institute of Informatics and Mathematics, Vilnius, Lithuania. At present possesses the Habit. Doctor's Degree, professor. His main research interests include software engineering, information system engineering, legislative engineering, and knowledge-based systems.

Ontologinių žinių apie verslo procesus pakartotinis naudojimas IS inžinerijoje: proceso konfigūravimo problema

Donatas ČIUKŠYS, Albertas ČAPLINSKAS

Straipsnyje pristatomas verslo procesų žinių pakartotinio naudojimo metodas, grindžiamas dalykinių sričių inžinerija, žinių inžinerija ir ontologijomis grindžiama sistemų inžinerija. Pagrindinė siūlomo metodo idėja yra atskirti verslo proceso ontologiją nuo dalykinės srities ontologijos ir pakartotinai naudoti proceso ontologiją skirtingose dalykinėse srityse. Pasiūloma apibendrinto verslo proceso sąvoka, apibrėžiama kaip panašių verslo procesų šeima. Straipsnyje aptariamas apibendrinto verslo proceso nuleidimas į dalykinę sritį, susidedantis iš dviejų gyvavimo ciklų. Pirmame cikle yra atliekama apibendrinto proceso inžinerija, antrame – konkretaus proceso inžinerija. Pastaroji susideda iš trijų žingsnių: proceso konfigūravimo, dalykinės srities esybių priskyrimo proceso vaidmenims ir valdymo srautų tarp proceso veiklų apibrėžimo. Didžiausias dėmesys straipsnyje skiriamas proceso konfigūravimo problemai, kurią siūloma spręsti pasinaudojant aprašomosiomis logikomis (angl. Description Logics).