Enabling compile time variability modelling and implementation in business processes for Eclipse developers

Master Thesis

Author: Roman Sizonenko
Supervisor: Dr. S.J.B.A. (Stijn) Hoppenbrouwers

January 2014
Abstract

Motivation:
Business process is a complex set of activities, performed by actors or executed fully automatically, that are aimed to achieve a specific business goal. All business processes are unique and contain different elements of variability. Both industry and academia are interested in new approaches in modelling and implementing variability in BP.

Problem statement:
Most of the time business processes are modelled in a notation of a specific BP automation technology provider with variability resolved at runtime using control flow elements. But what if it would be possible to detach modelling process from a specific platform and allow only a relevant business process with resolved variability at compile-time be deployed to a business process management system?

Approach:
To solve the problem, a deeper look has to be taken into fundamentals of software product lines and variability modelling of business processes. Based on these findings a new modelling language that supports variability modelling of business process is introduced.

Results:
The new approached allows developers to model variability in business processes independent from automation platform providers in an Eclipse environment, enjoying the full tool support.

Conclusions:
The basis for a perspective research is prepared in this thesis. As still, there are both technical and conceptual challenges for future development, a new and innovative approach was introduces and put to the test.
# Contents

## INTRODUCTION .................................................................................................................................. 4

## 1 VARIABILITY: FUNDAMENTALS, CONCEPTS, IMPLEMENTATION...................................................... 5

  1.1 SOFTWARE PRODUCT LINE FUNDAMENTALS ...................................................................................... 5
  1.2 VARIABILITY IN PROCESSES AND RELATED SERVICES .................................................................. 7
  1.3 VARIABILITY IMPLEMENTATION PATTERNS IN PROCESSES AND RELATED SERVICES, ................. 8
     1.3.1 Process and Service Patterns Generation ....................................................................................... 8
     1.3.1 Process and Component Service Replacement ............................................................................... 9

## 2 VCONCEPT: LANGUAGE SPECIFICATION ......................................................................................... 10

  2.1 CORE VARIABILITY CONCEPT MODELLING ELEMENTS ................................................................. 11
    2.1.1 Models ........................................................................................................................................ 11
    2.1.2 Types ........................................................................................................................................ 12
    2.1.3 Decision Variables ....................................................................................................................... 16
    2.1.4 Constraints ................................................................................................................................. 18
  2.2 BUSINESS PROCESS CONCEPT MODELLING ELEMENTS ............................................................. 22
    2.2.1 Basic Business Process Modelling Concepts .............................................................................. 23
    2.2.2 Advanced Business Process Modelling Concepts ....................................................................... 27
    2.2.3 Variability Modelling Concepts .................................................................................................. 28
  2.3 BUILT-IN OPERATIONS .................................................................................................................... 30
    2.3.1 Internal Types ............................................................................................................................ 30
    2.3.2 Basic Types ............................................................................................................................... 30
    2.3.3 Enumeration Types ...................................................................................................................... 33
    2.3.4 Collection Types ........................................................................................................................ 34
    2.3.5 Compound Types ....................................................................................................................... 37

## 3 PROOF OF CONCEPT WITH RUNNING EXAMPLE ............................................................................. 38

  3.1 PREREQUISITES FOR RUNNING VCONCEPT IN ECLIPSE ............................................................ 38
  3.2 TOOL SUPPORT FOR VCONCEPT ..................................................................................................... 39
  3.3 RUNNING EXAMPLE OF A BUSINESS PROCESS MANAGEMENT SYSTEM ........................................ 40
  3.4 VARIABILITY MODELLING WITH VCONCEPT ................................................................................. 41
  3.5 VARIABILITY RESOLUTION EXAMPLES WITH BPMS ENGINES .................................................... 44
    3.5.1 Business process model with Activiti ............................................................................................. 44
    3.5.2 Business process model with jBPM .............................................................................................. 46
    3.5.3 Business process model with Spring Web Flow ............................................................................ 48

## CONCLUSIONS.................................................................................................................................. 51

## BIBLIOGRAPHY .................................................................................................................................. 53

A.1 APPENDIX 1 – VCONCEPT SOURCE CODE ...................................................................................... 55
Introduction

At this moment, Business Process Management is a popular topic both in academia and industry. It offers challenges and benefits for researchers and business operators. The idea behind it is that almost all products or services are a result of a sequence of activities, performed by actors or executed automatically. These activities may vary due to specific implementation domains, localization or different market conditions. This is why a business process is always variable.

Modelling and implementing variability is a non-trivial and important task. If done in effective manner it allows to save resources, couple with changing market conditions, adapt for specific clients and domains needs and gain an upper hand with the competitors. Mishandling, on the other hand, might lead to completely opposite results.

Some of the main requirements from the industry towards variability in business processes are:

- **R1**: A possibility to model complex business processes in a simple manner, understandable for developers and analysts;
- **R2**: Provide a relevant independence from business process automation technology providers, but at the same time support their products and allow reconfiguration without serious process and system reengineering;
- **R3**: Bring the full tool support of the development environment;
- **R4**: Allow code and services reuse, opportunity to map developed functionality to specific processes;
- **R5**: Automation of the variability resolution and implementation;

And this is just a top of the iceberg of all requirements and demands by the industry. Trying to going in-depth and covering all this issues would be a solid foundation for a PhD thesis. As this thesis shall aim at fulfilling the requirements for a Master degree only essential concepts are a part of the agenda.

The goal of this thesis is to introduce a way for modelling and implementing variability in business processes (and business process management systems, as these two things cannot be separated).

To achieve this goal several tasks were needed to be resolved:

- Research the fundamentals of software product lines and variability in business processes in particular. This would provide a starting point for understanding what types of variability are there and how it could be modelled and implemented;
- Develop a domain specific language which could support variability modelling in business processes, alongside with variability in business process management system modelling;
- Test introduced concepts with an example that would include a business process in a certain environment. Explicitly demonstrate variability and how it will be resolved.

Thesis itself is structured accordingly to the tasks described above. In Section 1 the theoretical background of variability in business processes is covered. Section 2 is the core part and introduces the language specification. Section 3 is dedicated to the running example that illustrates how a specific business process in a business process management system could be modelled and variability resolved.
1 Variability: fundamentals, concepts, implementation.

The purpose of this section is to provide an overview of major concepts that are relevant to understanding variability modelling and implementation. Variability (and hence variability modelling and implementation) is a classical product line concept.

In Section 1.1 I will provide a short introduction to basic concepts in product line engineering.

In Section 1.2 I will give an overview of variability in processes and related services.

In Section 1.3 I will describe variability implementation patterns in processes and related services.

1.1 Software Product Line Fundamentals

In this section I give an overview on some fundamental terminology in software product line development.\(^1\) First, I discuss how product line engineering differs from traditional software development and how reuse is addressed in product line engineering.

Traditional software development focuses on the development of individual products, typically in a project-based approach, where each project commands its own infrastructure, including all relevant software. In contrast, software product line development addresses the integrated development of a range of similar, but also different products. The products in a product line may support different, individual customers or may address entirely different market segments. Instead of understanding each individual system by itself, software product line engineering looks at the product line as a whole and systematically addresses the differences of the products. Thus, while systems in a product line differ in some characteristics (often referenced as features) to serve the needs of different customers or market segments, the individual systems need also to share a set of common functionality so they can be effectively developed together as a product line.

The distinction between software development for reuse and development with reuse is fundamental in software product line engineering. Development for reuse (domain engineering) provides a basis for the development of individual products in terms of assets designed and realized for reuse. In contrast, development with reuse (application engineering) builds the final products on top of the assets developed for reuse in domain engineering. This distinction is shown as two-life-cycle model in Figure 1.1. The two-life-cycle model consists of a specific software development lifecycle for domain engineering activities and a second lifecycle for application engineering. The latter relies on the reusable assets created in domain engineering.

Domain engineering provides a basis for the development of individual products. This basis, the so-called product line infrastructure (also known as core asset base or product line platform (1)) consists of all assets being relevant to the software development during the whole software development lifecycle. To serve for all products that may be developed in the product line, the assets in the product line infrastructure may contain explicit variability. Variation points in the assets describe the potential locations of impact for individual variability. For example, in software architecture model individual services may be tagged as variability points so that the concrete architecture model for a product contains

\(^1\)This section is largely based on material from (2) Chapter 1. Logical structure and concepts are developed by (2) authors and reused here as a short report. Additional sources are quoted separately.
only selected services. This shows also that the core approach in product line engineering to dealing with reuse relies on configuring individual variations.

Application engineering builds the final products based on the product line infrastructure, which usually contains most of the required functionality. The development of a new product based on an existing product line infrastructure consists of eliciting requirements, categorizing requirements as being part of the product line or product-specific and configuring the variability in the product line infrastructure, i.e. deriving instantiated versions of the assets that exactly adhere to the requirements of the specific product.

Explicit variability in the assets of the product line infrastructure is a key concept in supporting different customers or market segments. Variability management encompasses all activities for systematically addressing the variability throughout software product line engineering, e.g. defining, representing, implementing or evolving variability. In product line engineering three main types of requirements may be distinguished:

- **Commonalities** are (functional or non-functional) characteristics that are common to all products in the product line and implemented as a part of the product line infrastructure.

- **Variability** is a characteristic that may be common to some products, but not to all. Variability must be explicitly modelled, implemented and resolved in a way that allows having it in selected products only.

- **Product-specific** characteristics are part of only one product. Typically, product-specific characteristics arise in order to address the concerns of individual customers or markets. This type of variability will not be realized directly by the product line infrastructure, but the infrastructure must be able to support this type of extension.

While commonalities and variability are provided in domain engineering, product-specific characteristics are exclusively handled in application engineering.

An important question is also: when is the decision made about the specific contents of the variability in the assets? Many different approaches are possible and discussed in literature. 2 main concepts are given below:

- **Compile time**: this means that the variability is realized in terms of compile time modifications. Examples could be pre-processors or mechanisms that are part of the
build-process like static weaving in aspect-oriented programming. Often a more refined distinction is made, that also identifies link time as a separate binding time or distinguishes between compile-time and pre-compile-time. This detailed division is not necessary here as I will not go into details of source code compilation itself.

- **Runtime**: this subsumes all other cases of binding (determining) variability that happens during the execution of the system for which the variability applies like a control flow statement implementation in object-oriented programming.

Mastering a range of products instead of individual products is not just a technical topic, e.g. how to effectively realize different functionality among similar but also different products. In addition to architectural and technical concerns, successful software product line development also needs to address business, process and organization concerns (2).

### 1.2 Variability in Processes and related Services

In my research, I identified two representative approaches to realize variability in the composition of services which is a business process model.

The composition of processes and services at **implementation / compile time** is often supported by implementation techniques that exploit MDD techniques, e.g. model transformation and model element mapping.

In (3) the Business Process Family Model (BPFM) is proposed. This represents a variability-enhanced common business process model as a core asset. Park et al. (4) use this as a basis to derive business process variants. The BPFM maps features of an initial feature model to domain activities (these activities are mapped to domain services in a following step). The BPFM is modelled using Unified Modelling Language (UML) (5) activity diagrams that are expanded with variation points, variation point bindings and variant regions. The derivation of a business process variant is based on a given feature configuration and the selection of the corresponding core domain activities and variants. This is, deriving a UML activity diagram in which all variability is resolved. The automated transformation of an UML 2.0 activity diagram to Business Process Execution Language BPEL (6) is presented in (7), (8), (9). Similar to business process models, Steffen et al. (10) introduce reusable flow graphs within METAFrame environment (11) to model service behaviour and then use the final and consistent flow graph to automatically instantiate new services.

Service composition can also be changed at **runtime** to meet new QoS constraints. Li et al. (12) present a set of algorithms for QoS-driven dynamic reconfiguration of SOA-based systems. In this approach, each Web service is annotated with QoS attributes, e.g. response time or cost. Given a set of available services with their QoS attributes and a new QoS constraint, the presented algorithms will calculate the QoS of the overall system with regard to the process structure. As long as the QoS of the system does not meet the new QoS constraint, one or more services will be replaced to meet the constraint – the overall process structure will not be changed.

SAP provides standard platforms and applications, which can be adapted to the customer’s specific business needs. One provided solution is Business Configuration by scoping and fine-tuning for Business By Design\(^2\), SAPs on-demand platform. Additional to the

platform SAP delivers at one hand a comprehensive catalogue (BAC, Business Adaptation Catalogue) exposing the entire set of solution capabilities, described using non-technical business language. At the other hand, SAP and partners deliver so-called Business Configuration (BC) sets, which contains predefined business configurations. At first the customer has to make selection from the BAC based on his specific business needs (scoping), at second he can overwrite parameters of the predefined configuration (fine-tuning). The results are stored in a configuration workspace. The final configuration becomes active through BC deployment. The process comprises activation of UI components and services as well as writing the configuration to Customizing Tables, which are evaluated at runtime.

1.3 Variability Implementation Patterns in Processes and related Services.

In this section I provide an overview of different variability implementation techniques found in literature, which are relevant to service oriented development. Overview is mainly based on a literature review, but also takes into account practical experience with variability modelling and implementation.

The structure of patterns is inspired by standard pattern catalogues like (13), however, it was refined based on the specific context of the work, namely variability modelling and implementation.

1.3.1 Process and Service Composition Generation

Name: Process and service composition generation
Purpose:

Description: Support variability of service composition by generating a specific business process (in BPEL (6)) based on feature configuration and the selection of corresponding domain activities of a Business Process Family Model (BPFM). A BPFM specifies common and variable services in the same way a Feature Model specifies common and variable features in traditional product line engineering.

Variability Object: Service composition (and processes)
Form of Variation: Alternative, Optional
Binding time: Compile time, permanent

Context:

Environment Contexts: The existing implementation requires Web Service technology powered by BPMN/BPEL.

Solution:

Key idea: Generation of specific business processes (in BPEL) from a generic BPFM including common and variable services. The BPFM is modelled using an extension of UML activity diagrams or BPMN including variation points, variation point bindings and variants regions (introduced by this approach). The generation of a business process variants from the BPFM is based on a given feature configuration and the selection of the corresponding core domain activities and variants (used to implement the selected features). This is, deriving a UML activity diagram in which
all variability is resolved. The automated transformation of an UML 2.0 activity diagram (5) to BPEL is presented in (7), (8), (9).

**Technology Background:** Transformation component is required to automatically transform a business process variant derived from the BPFM (UML activity diagram) into a business process defined in BPEL.

**Variability Approach:** Generation based
**Variability Granularity and Selection:** Service

**Dependency Management Support:** Rules and dependencies among services are managed on the level of the feature model by specifying relations and dependencies among features using the FODA methodology (14).

**Further Aspects:**

**Source:** (4), (3) and the conceptually equal Business Process Line (BPL) approach proposed in (15)

### 1.3.1 Process and Component Service Replacement

**Name:** Process and component service replacement

**Purpose:**
**Description:** Support variability of service composition by replacing one or more component services of a business process. A component service provides functionality that is required to achieve the overall goal of the business process. Variability exists in the way that component services are replaced to meet varying QoS constraints.

**Variability Object:** Service composition (and processes)
**Form of Variation:** Alternative
**Binding time:** Compiletime, runtime, volatile

**Context:**
**Environment Contexts:** The existing implementation requires Web Service technology powered by BPMN/BPEL.

**Assumption on Systems:** Activiti, jBPM, Spring Web Flow

**Solution:**
**Key idea:** Triggered by a QoS constraint violation, each component service of the business process is analysed to identify one or more component services that have to be replaced in order to satisfy the QoS constraint (this is based on a set of QoS attributes, their values and calculation algorithms). The replacement of component services does not affect the overall structure of the business process.

**Technology Background:** -
**Variability Approach:** Generation based
**Variability Granularity and Selection:** Service

**Dependency Management Support:** Rules and dependencies among services only exist in the sense of QoS attributes and constraints on the overall QoS value of the business process. No specific variability modelling approach is given.

**Further Aspects:**
**Source:** (12)
2 vConcept: Language Specification

In this section, I will describe the concepts of the vConcept – domain specific language for variability modelling with emphasis on business processes and related service. I will distinguish between a core variability modelling and a business process variability modelling. This distinction facilitates ease of use for the most standard issues in variability modelling as it does not complicate the use of this language for users who do not need the more advanced and specialized features. The concepts of the modelling language are based on the results of the theoretical study in Section 1. In this section, I discussed different levels of expressiveness for basic variability modelling. The core variability modelling language concepts are extended with modelling concepts specific for business process modelling that I identified as prerequisites to effective and efficient variability implementation in business process management systems in particular.

The basic concepts of the vConcept are related to approaches like the Text-based Variability Language (TVL) (16), the Class Feature Relationships (Clafer) (17), the Compositional Variability Management framework (CVM) (18), etc. However, I decided to develop a different approach, based on decision modelling concepts, in order to appropriately address the software product line requirements identified in Section 1.

I will introduce a textual specification to describe the vConcept concepts. This will help to give a precise representation of the modelling concepts. The syntax, I use in this section was developed as a basis for representing these concepts. My presentation of the vConcept-syntax draws upon typical concepts used in programming languages, in particular Java, and other modelling languages such as TVL (16), Clafer (17), the Object Constraint Language (OCL) (19), or the UML (5). The dependency management concepts of the vConcept mostly rely on the concepts of the OCL. I will adapt these concepts as needed to provide additional operations required by vConcept specific modelling elements, e.g. match and substitute operations for decision variables of type string.

I will use the following styles and elements throughout this section to illustrate the concepts of the vConcept:

• The syntax as well as the examples will be illustrated in Courier New.

• Keywords will be highlighted using bold font.

• Elements and expressions that will be substituted by concrete values, identifiers, etc. will be highlighted using italics font.

• Identifiers will be used to define names for modelling elements that allow the clear identification of these elements. I will define identifiers following the conventions typically used in programming languages. Identifiers may consist of any combination of letters and numbers, while the first character must not be a number. I recommend that the identifiers of new types start with a capital letter to easily distinguish them from variables.

• Expressions will be separated using semicolon “;”.

• Different types of brackets will be used to indicate lists “()”, sets “{}”, etc. This is closely related to the Java programming language.

• I will indicate comments using “//” and “/ * . . . * /” (cf. Java).
I will use the following structure to describe the different concepts:

- **Syntax:** this is the syntax of a concept. I will use this syntax to illustrate the valid definition of elements as well as their combination.
- **Description of syntax:** provides the description of the syntax and the associated semantics. I will describe each element, the semantics and their interaction with other elements in the model.
- **Example:** the concrete use of the abstract concepts is illustrated in a (simple) example.

In Section 2.1 I will describe the core variability concept modelling language elements. I will introduce the required elements and expressions to define a basic configuration space including Boolean and non-Boolean variability.

In Section 2.2 I will describe the business process specific concept modelling language elements. I will introduce extensions that are required to satisfy the specific requirements of variability modelling of a business process.

In Section 2.3 I will describe built-in operations that mainly refer to the core variability concept modelling elements.

### 2.1 Core variability concept modelling elements

This section describes the core elements of the vConcept. In vConcept language, a model is the top-level element that identifies the configuration space of a certain (software) project. In terms of a product line, this may either be an infrastructure as a basis for deriving products or a final product. In a model the relevant modelling elements will be defined. I describe this in the first part of this section. In the second part, I introduce the type system supported by the vConcept. These types can be used to declare different types of decision variables. The dependency management capabilities to restrict the configuration space of a model will be described next. Finally, I will introduce the configuration concept of the vConcept, which enables the definition of specific (product) configurations based on the configuration space defined in a project.

#### 2.1.1 Models

In the vConcept a model (model) is the top-level element in each model. This element is mandatory as it identifies the configuration space of a certain software project and, thus, scopes all variability of that software project. The definition of a project requires a name, which simultaneously defines a namespace for all elements of this project.

**Syntax:**

```plaintext
model name{
    /* Definition of the configuration space and configurations. */
}
```

**Description of syntax:** the definition of a new model consists of the following elements:
• The keyword **model** defines that the identifier **name** is defined as a new model or, to be more precise, as a new configuration space.

• **name** is an identifier that defines the name of the new model and, thus, the namespace of all elements within this model.

• The elements surrounded by curly brackets define the configuration space of the new model.

**Example:**

```java
model contentSharing{
    /* This will define a new model for a content-sharing project. This is related to my running example.*/
}
```

### 2.1.2 Types

In a model (cf. Section 2.1.1) different kinds of core modelling elements may be used to both represent the variability and define a configuration space appropriately. I will express these kinds as formal types in vConcept, thus defining a (strongly) typed language. I distinguish between basic types, enumerations, container types, derived and restricted types and compound types. These types can be used to declare or define concrete decision variables. Basically, all decision variables can be unset using the null keyword, i.e., explicitly assigning no value to a variable.

#### 2.1.2.1 Basic Types

Non-Boolean variability is a must for the core expressiveness of the vConcept language. Thus, the vConcept supports as basic types Boolean (**Boolean**), integer (**Integer**), real (**Real**) and string (**String**) with their usual meaning. The names of the basic types are aligned to OCL (19). These types support the definition of basic variability, e.g. the **Boolean** type may be used for modelling optional variability. In addition, types like **Integer** or **Real** provide a basis for defining advanced variability, e.g. using an **Integer** to define a quantitative property for Quality of Service (QoS).

#### 2.1.2.2 Enumerations

Enumerations allow the definition of sets of named values. This is used to describe a set of possible resolutions of a decision.

**Syntax:**

```java
enum Name1{value1, ..., value_n};

enum Name2{value1=n1, ..., value_n=n_n};
```

**Description of syntax:** the definition of a new enumeration type consists of the following elements:

• The keyword **enum** defines that the identifier **Name** is defined as a new enumeration.
• *Name* is an identifier and defines the name of the new type.

• The identifiers surrounded by curly brackets are the concrete elements of the enumeration. A specific element of an enumeration can be accessed using the "."-notation, e.g. *Name*.value.

• Specifying concrete numeric values for elements of an enumeration (*value* = *n*) turns the enumeration into an ordered enumeration. This enables relations like greater than (>) or less than (<) and operations like next (next) or previous (previous) on the values to be used.

**Example:**

code
```c
enum Colors {green, yellow, black, white};
enum BindingTimes {configuration=0, compile=1, runtime=2};
```

### 2.1.2.3 Container Types

The vConcept provides two container types, sequences and sets. Sequences can contain an arbitrary number of elements of a given content type (including duplicates), while sets are similar to sequences, but do not support duplicate elements. These types can be used to describe a number of possible options out of which several can be selected at the same time. Elements in a container (both sequences and sets) can be accessed by their position in the container using an index ([index]).

The vConcept supports a set of operations specific for container types, e.g. adding or appending elements to a container, deleting elements of a container, selecting specific elements, etc. I will introduce the full set of operations in the related section.

**Syntax:**

code
```c
// Declaration of a new sequence and a new set.

sequenceOf(Type) variableName1;

setOf(Type) variableName2;

/* Access to elements of a sequence. Sets do not have index-based access. I will discuss variables in Section 0. */

variableName1[index] = value;
```

**Description of Syntax:** the definition of a container type consists of the following elements:

• The `sequenceOf` and `setOf` keywords refer to a container of the respective type followed by the `Type` of the elements contained in brackets.

• The identifiers *Name1* and *Name2* are the names of the new containers.

• Accessing a specific element of a sequence container type (variable) requires the specification of an index ([index]). An index is either "0" or a positive integer
value specifying the position of an element in a container. Accessing a specific position is only a valid operation, if this position has previously been set by different means like the add function (the set of operations is introduced in the related section).

Example:

```c
/* Definition of a new enumeration. "threeD" means "3D". */
enum ContentType {text, video, audio, threeD};
```

```c
/* Denotes types of contents supported by a system */
sequenceOf(ContentType) basicContents = {ContentType.text, ContentType.audio};
```

2.1.2.4 Type Derivation and Restriction

The vConcept allows the derivation of new types based on existing types. This supports extensibility and adaptability as users may define their own types based on basic types, enumerations or container types as well as on previously derived types. The derivation may also include restrictions to the existing type, e.g. to restrict the possible values of the new type to a subset of the values of the existing type. The restrictions are defined by one or more constraints (I will discuss constraints in detail below). Multiple constraints are implicitly combined by a Boolean OR. Thus, at least one constraint has to be satisfied by the new type. The constraints will be defined in OCL style as described in related section.

**Syntax:**

```c
typedef Name1 Type;

typedef Name2 Type with (constraint1, ..., constraintN);
```

**Description of Syntax:** the definition of a derived type consists of the following elements:

- The **typedef** keyword indicates the derivation of a new type based on an existing type.
- The identifiers `Name1` and `Name2` are the names of the new types.
- The identifier `Type` denotes the basic type from which the new type (`Name1` or `Name2`) will be derived.
- The optional keyword `with` introduces a non-empty set of constraints, surrounded by brackets, out of which at least one must hold for `Name2`. In case of deriving `Name2` from `String` the constraints may define regular expressions.

**Example:**
/* Definition of a type "AllowedBitrates" which is a set of Integers, i.e. a kind of alias for a complex type definition. */

typedef AllowedBitrates setOf(Integer);

/* A new modelling type of the basic type integer that is restricted to assume values between "128" and "256". */

typedef Bitrate Integer with (Bitrate >= 128 and Bitrate<= 256);

2.1.2.5 Compounds
A compound type groups multiple types into a single named unit (similar to structs or records in programming languages or groups in feature modelling). This allows combining semantically related decisions from which each element has to be configured individually.

Syntax:

```
compound Name { 
    Typename1;
    
    ... 
}
```

Description of Syntax: The definition of a compound type consists of the following elements:

- The `compound` keyword indicates the definition of a new compound type.
- The identifier `Name` defines the name of the new compound type.
- The set of elements surrounded by curly brackets defines the types of the compound type. Each declaration of a typed element is separated by a semicolon.

Example:

/* A new compound type for the configuration of different (web) content. The content may vary in terms of name and bitrate. "Content.bitrate" is the integer within the compound content. */

```
compound Content { 
    String name; 
    
    Integer bitrate; 
}
```

In vConcept a compound may extend the definition of a previously defined (parent) compound. This is indicated by the `refines` keyword. Extending compound types is similar to subclassing in object-oriented languages, i.e. `parentType` becomes a subtype of `compoundType` and `compoundType` may define further decision variables.

Syntax:
compound Name_1 refines Name_2 {
    // Define additional elements.
}

**Description of Syntax:** the definition of an extended compound type consists of the following elements:

- The compound keyword indicates the definition of a new compound type.
- The identifier Name_1 defines the name of the new compound type.
- The refines keyword indicates that the new compound type (Name_1) is an extension of a previously defined compound type (Name_2).
- The set of elements surrounded by curly brackets defines the additional elements that make up the extensions to the inherited elements of compound Name_2.

**Example:**

```java
/* A compound type for the configuration of different (web) content. */
compound Content {
    String name;
    Integer bitrate;
}
/* A new compound type that refines the previous compound type. "ExternalContent" will subsume all elements of "Content" and all additional elements defined below. */
compound ExternalContent refines Content {
    String contentPath;
    String accessPassword;
}
```

### 2.1.3 Decision Variables

The types introduced in Section 2.1.2 can be used to declare (decision) variables representing a concrete variability. A decision variable is an element of a project (configuration space) that basically accepts any value of its type. Constraints may further restrict the possible values by removing certain combinations of values from the allowed configuration space. The value given to a decision variable defines the variant of the represented variability.

In vConcept a decision variable may either be declared with or without a default value (this is an optional parameter). Decision variables with a default value can be further configured by overwriting their (default) value at a later point in time. However, overwriting the default value is not necessary.
Syntax:

// Declaration of a decision variable.

Type name1;

/* Declaration of a decision variable with a default value. The "valueAssignment"-expression will be described in detail below. */

Type name2 = valueAssignment;

Description of Syntax: the basic declaration of a new decision variable (excluding the declaration of an optional default value) consists of the desired type (one of the basic types, an enumeration, a container type, a derived or a restricted type, or a compound type) followed by an identifier (name1) that states the name of the variable.

Optionally, a default value can be assigned to a decision variable appending "=" followed by a "valueAssignment"-expression after the name (name2) of the decision variable. The form of the "valueAssignment"-expression depends on the specific type of the declared decision variable:

- Basic types and Enumerations: an expression that yields a value of the corresponding type and can be actually calculated, i.e., it either consists of constants or the values of the variables are known.
- Container types: either an expression of the type of the container, which can be statically evaluated, or a set of values separated by commas in curly brackets after the name of the decision variable. Expressions may be used but must be stated in parenthesis due to technical reasons. The allowed values within the curly brackets are determined based on the base type of the container.
- Compounds: either an expression of the type of the compound, which can be statically evaluated, or a set of individual assignments, given in curly brackets. Each assignment explicitly gives the field in the compound that the assignment is made to, followed by a "=" and an expression of the corresponding element type. Again this expression needs to be statically evaluated.
- Derived type: the assignment follows the rules of the base type.

Example:

/* Declaration of a new variable of type integer with a default value. */

Integer bitrate =128;
Colors backgroundColor = Colors.black;

sequenceOf(ContentType) baseContent = {ContentType.text,
            ContentType.audio};

Content complexContent = {name = "Text", bitrate = 128};

2.1.4 Constraints

Constraints are used to define validity rules for a variability model, e.g. by specifying dependencies among decision variables. The syntax of constraints in the vConcept basically follows the structure of expressions in propositional logic and, thus, is composed of:

- Simple sentences, which represent constants, decision variables and types which can be named by (qualified) identifiers.
- Compound sentences created by applying the operations to simple sentences and, in turn, to compound sentences. A correct compound sentence requires that the arguments passed to operations match the parity of the operation and the types of the parameters or operations comply, respectively.

The operations available in vConcept as well as the type compliance rules will be discussed in the remainder of this section.

The constraints in vConcept will mostly rely on the relevant part of the syntax as well as on a large subset of the operations defined in OCL. In vConcept I use the constraint expression syntax of OCL, but omit the OCL contexts used to relate constraints to UML modelling elements. Similar to OCL, all elements defined in a vConcept model will be accessible to constraints. Two examples for constraints are given below, one propositional and one first-order logic example using a quantifier:

- $(10 <= a \text{ and } a <= 20) \implies b == a;$
  
  If $a$ is in the range $(10; 20)$ this implies that $b$ must have the same value as $a$.
- Standalone constraints: Constraints are given as statements in a project or within a compound so that compound fields are directly accessible without qualification. As standalone constraints are used like statements, they end with a semicolon (as shown in the two examples above).
- Embedded constraints: One or more constraints are used as part of a statement, for example a typedef. Here the constraint is written in parenthesis and not ended by a semicolon.

Below I will discuss individual elements of constraints in vConcept and, in particular, the difference (in particular regarding an adapted notation) to the related elements in OCL.
Large parts of the remainder of this section are directly taken over from the OCL specification (19) and adapted to the vConcept context.

Keywords

Keywords in vConcept constraint expressions are reserved words. That means that the keywords cannot occur anywhere in an expression as the name of a decision variable or a compound. The list of keywords is shown below:

- `and`
- `def`
- `else`
- `endif`
- `if`
- `iff`
- `implies`
- `in`
- `let`
- `not`
- `or`
- `then`
- `xor`
- `null`

Prefix operators

vConcept defines two prefix operators, the unary

- Boolean negation `not`.
- Numerical negation `-` which changes the sign of a Real or an Integer.

Infix operators

Similar to OCL, in vConcept the use of infix operators is allowed. The operators `+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=`, `<>` are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

\[ a + b \]

is conceptually equal to the expression:

\[ a .+(b) \]

that is, invoking the `+` operation on `a` (the operand) with `b` as the parameter to the operation. The infix operators defined for a type must have exactly one parameter. For the infix operators `<`, `>`, `<=`, `>=`, `and`, `or`, `xor`, `implies`, `iff` the return type is Boolean.

Please note that, while using infix operators, in vConcept Integer is a subclass of Real. Thus, for each parameter of type Real, you can use Integer as the actual parameter. However, the return type will always be Real.

Equality and assignment operators (default logic)
In contrast to OCL, vConcept provides two operators which are related to the equality of elements with different semantics, namely the default assignment ‘=’ and the equality constraint operator ‘==’. I will explain the difference in this section.

Basically, a decision variable in vConcept is considered as undefined, i.e., the variable does not have an effect on the variability resolution. Constraints may explicitly refer to the undefined state via the operation “isDefined”.

A default value may be assigned to a variable. Default values can be used to define a basic configuration (a kind of basic profile) which applies to all products in the product line. A default value can be defined as part of the variable declaration (using the ‘=’, cf. Section 0) or in terms of an individual default assignment using the ‘=’ operator. However, a default value may only be modified (assigned or changed) once in a given model. This restriction is required due to the fact that vConcept does not provide support to define the sequence of evaluations.

As the ‘=’ operator defines a default value which may be overridden, it is not possible to use that operator to express that a decision variable must have a certain value (under some conditions). This can be achieved using the equality operator ‘==’. Basically, the equality operator checks whether the left hand and the right hand operand have equal values. In two distinct cases, the equality operator enforces the value specified by the right hand operand. The cases are the

- Unconditional value constraint, e.g., a == 5.
- Conditional value constraint given as the right side of an implication, e.g., c < 5 implies a == 5.

In these two cases, the equality operator expresses that the left hand operand (an expression denoting a decision variable) must have the same value as the right hand operand. If the left hand operand contains a default value, then the default value will be overridden. However, if two expressions aim at enforcing different values for the same decision variable, the model becomes unsatisfiable.

**Precedence rules**

The precedence order for the operations, starting with highest precedence, in vConcept is:

- dot operations: ‘.’ (for element and operation access).
- unary ‘not’ and unary minus ‘~’
- ‘*’ and ‘/’
- ‘+’ and binary ‘−’
- ‘if-then-else-endif’
- ‘<’, ‘>, ‘< =’, ‘> =’
- ‘==’ (equality), ‘<>’, ‘!=’ (alias for ‘<>’)
- ‘and’, ‘or’ and ‘xor’
- Default assignment ‘=’
- ‘implies’, ‘iff’ Parentheses ‘(’ and ‘)’ can be used to change the precedence of operators in expressions.
Type conformance
Type conformance in vConcept constraints is inspired by OCL (cf. OCL section 7.4.5):
• AnyType is the common superclass of all types. All types comply with AnyType. However, AnyType is typically used for defining the built-in operations. The only value of AnyType is null, which explicitly makes a decision variable undefined.
• Each type conforms to its (transitive) supertypes. Figure 2.1 depicts the vConcept type hierarchy.
• Type conformance is transitive.
• The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real (actually the type Integer is considered as a subclass of Real).
• Containers are parameterized types regarding the contained element type. Containers comply only if they are of the same container type and the type of the contained elements complies.
• The refines keyword induces a hierarchy of compounds where the subtypes are compliant to their parent types, i.e. the parent type may be replaced by each subtype.
• Derived types are compliant to their base type as long as if no constraints were specified.
• MetaType is a specific type denoting types, e.g. to constrain types of elements within a collection.

Type operations
vConcept provides the isTypeOf(), isKindOf() and typeOf() operations. The first two operations are similar to the related operations in OCL. The latter one returns the actual type (MetaType) of a decision variable, compound field or container element. MetaType allows equality and inequality comparisons. Currently, vConcept neither supports re-typing or casting.

Enumeration Types
Enumerations literals are used just like qualified names, i.e. using a dot. For a certain enumeration type only the enumeration literals may be used with default assignment (‘=’), equality (‘==’) or unequality (‘! =’, ‘<>’) operators. In case that ordinals are explicitly specified for enumeration literals, also relational operators (‘<’, ‘>’, ‘<=’, ‘>=’) may be used.
**Compound Types**
Decision variable declarations defined within a compound can be accessed using the dot operator ‘.’.

**String Type**
In addition to the string operations defined for OCL, I added two operations based on regular expressions, namely matches and substitutes.

**Configuration Type**
A decision variable of type Configuration represents a variable constraint. Such a variable needs to be used somewhere in a vConcept model in order to become active. Further statements or constraints may override the constraint in such a variable.

**If-then-else-endif Expressions**
The if-then-else-endif construct supports determining a value depending on a Boolean expression, similar to distinction of cases in mathematics. Exactly one expressions must be used within the then and else parts, both yielding the same type. The else part is not optional.

```plaintext
if contents[0].type == “video”
  then contents[0].bitrate
  else contents[0].highBitrate
```

### 2.2 Business process concept modelling elements

In this section I discuss the individual workflow modelling concepts of vConcept. Benefits, that are targeted to be gained by specifically delimiting business process modelling, are following:

- vConcept allows to specify business processes independently from the underlying implementation technology, i.e., it allows to abstract and (if needed) switch among possible business process automation engines (jBMP, Activiti, Spring Web Flow, etc.).
- vConcept is tailored for a specific application domain (business process modelling) and business process modellers expertise.
- It offers appropriate and familiar domain-specific notations from the start and aims at productivity improvement of analysts and programmers. Further, it shields the user from typical business process modelling concepts which are not relevant for most cases of business-process management systems (such as parallelism or required end states implicitly handled by default exit transitions).
- vConcept provides integrated variability selectors which are syntactically aligned to those to be used in the developed software source code.
- vConcept gives analyst and programmers a common language.
- vConcept is based on appropriate and established notations for modelling workflows. Terminology is cross-referenced with UML Activity Flow and BPMN, syntactical structures are related to Java.
- vConcept has predefined business rules and error handling that allows standardization, optimization and verification of workflow modelling.
• Customized business processes can be automated with minimal time and effort due to simplicity of the tailored language.
• Spring Web Flow, jBMP or Activiti source code patterns involved are too complex for on-the-fly modelling. vConcept makes models human-readable.
• vConcept has full Eclipse IDE support like syntax colouring, content assist and template proposals.

2.2.1 Basic Business Process Modelling Concepts

2.2.1.1 Workflow
Workflow represents a named container as a single unit. A workflow may contain states, choices, transitions.

Syntax:

```plaintext
workflow Name {  
  Nested elements;  
}
```

**Description of Syntax:** the definition of a workflow consists of the following elements:

- The `workflow` keyword indicates the definition of a new workflow.
- The identifier `Name` defines the name of the new workflow.
- The set of elements surrounded by curly brackets defines the nested elements of the workflow. Each declaration of an element is separated by a semicolon.

**Attributes:** name (ID), nested elements.

**Nested elements:** package (multiple), import (multiple), start (multiple), state (multiple), choice (multiple), end (multiple).

**Example:**

```plaintext
workflow ContentSharing {  
  start ContentSharingStart {  
    transition to LogIn on EVENT;  
  }  
  state LogIn {  
    transition to Exit;  
  }  
  state exit {  
  }  
}
```
2.2.1.2 Start
Start element defines the entry point into a workflow.

Syntax:

\[
\text{start } \text{Name} \{ \\
\hspace{1em} \text{Nested elements;} \\
\}
\]

Description of Syntax: the definition of a start element consists of the following elements:

- The start keyword indicates the definition of a new start element.
- The identifier Name defines the name of the new start element.
- The set of elements surrounded by curly brackets defines the nested elements of the start element. Each declaration of an element is separated by a semicolon.

Attributes: name (ID), nested elements.

Nested elements: Alternative group (variability element, multiple), optionality (variability element, multiple), service (multiple), transition (multiple).

Example:

\[
\text{start ContentSharingStart } \{ \\
\hspace{1em} \text{transition to LogIn;} \\
\}
\]

2.2.1.3 End
End element stops the execution of the workflow.

Syntax:

\[
\text{end } \text{Name} \{ \\
\hspace{1em} \text{Nested elements;} \\
\}
\]

Description of Syntax: the definition of an end element consists of the following elements:

- The end keyword indicates the definition of a new end element.
- The identifier Name defines the name of the new end element.
- The set of elements surrounded by curly brackets defines the nested elements of the end element. Each declaration of an element is separated by a semicolon.
Attributes: name (ID), nested elements.

Nested elements: Service (multiple).

Example:

```plaintext
end ContentSharingEnd {
}
```

2.2.1.4 State

State element is the main element that contains initialization of related events (methods). Workflow can contain multiple state elements.

Syntax:

```plaintext
state Name {
    Nested elements;
}
```

Description of Syntax: the definition of a state element consists of the following elements:

- The `state` keyword indicates the definition of a new state element.
- The identifier `Name` defines the name of the new state element.
- The set of elements surrounded by curly brackets defines the nested elements of the state element. Each declaration of an element is separated by a semicolon.

Attributes: name (ID), nested elements.

Nested elements: Alternative group (variability element, multiple), optionality (variability element, multiple), service (multiple), transition (multiple).

Example:

```plaintext
state LogIn {
    transition to Exit;
}
```

2.2.1.5 Transition

Transition defines the path from a workflow element to at least one successor. Containers may hold multiple transition elements that are triggered in order of their definition by a specific event (optional).

Syntax:

```plaintext
transition to destination on Event;
```
Description of Syntax: the definition of a transition element consists of the following elements:

- The transition to keywords indicates the definition of a new transition element.
- The identifier destination defines the destination of the transition.
- The identifier destination after a keyword on defines the event that will trigger the transition.

Attributes: destination (state, end, decision), event.

Nested elements: none

Example:

```
transition to End;
```

2.2.1.6 Choice

Choice element directs workflow path based on logical decisions. Workflow can contain multiple choice elements. Choices are similar to gateways in BPMN. Syntax is a typical Java if statements. Condition can be any logical expression based on attribute from a class or hard coded values.

Syntax:

```
choice Name{
    if (condition;){
        Nested elements;
    }
    else{
        Nested elements;
    }
}
```

Description of Syntax: the definition of a choice element consists of the following elements:

- The choice keyword indicates the definition of a new choice element.
- The identifier Name defines the name of the choice element.
- The identifier condition after a keyword if defines the constraint that need to be evaluated.
• The set of elements surrounded by brackets defines the nested elements of the choice element. Depending on constraint resolution outcome corresponding nested elements will be triggered.

**Attributes:** name (ID), condition, nested elements.

**Nested elements:** Alternative group (variability element, multiple), optionality (variability element, multiple), service (multiple), transition (at least one).

**Example:**

```java
choice ContentChoice{
    if (contentType == ContentType.audio) {
        transition to UploadAudio;
    }
    else {
        transition to Exit;
    }
}
```

### 2.2.2 Advanced Business Process Modelling Concepts

#### 2.2.2.1 Source

Defines the source classes in the project that contains methods triggered in workflow states.

**Syntax:**

```java
source Name;
```

**Description of Syntax:** the definition of a source element consists of the following elements:

- The `source` keyword indicates the definition of a new source element.
- The identifier `Name` defines the name of the related class.

**Attributes:** name (Qualified name with wildcard).

**Nested elements:** none

**Example:**

```java
source nl.ru.fnwi.LogIn;
```
2.2.2.2 Service
Method that should be executed in the workflow. Containers may hold multiple service elements that are triggered in order of their definition.

Syntax:

```
service method;
```

Description of Syntax: the definition of a service element consists of the following elements:

- The `service` keyword indicates the definition of a new service element.
- The identifier `method` defines the name of the related method.

Attributes: method

Nested elements: none

Example:

```
service logIn.execute();
```

2.2.3 Variability Modelling Concepts

2.2.3.1 Optionality
Optionality enables nested element in the derived workflow if the condition is met.

Syntax:

```
@Optionality(condition;
  Nested elements;
@/Optionality
```

Description of Syntax: the definition of an optionality element consists of the following elements:

- The `@Optionality` keyword indicates the definition of a new optionality element.
- The identifier `condition` inside the brackets represents constraint that needs to be validated.
- The set of elements following previous syntax defines the nested elements of the optionality element.
- The `@/Optionality` keyword indicates borders of an optionality element.

Attributes: condition, nested elements
**Nested elements:** service, transition, start, state, end, decision.

**Example:**

```java
@Optionality(deploymentTarget == DeploymentTarget.Azure;)
    transition to CountTraffic;
@/Optionality
```

### 2.2.3.2 Alternative

Alternative acts similar to switch statement in object oriented programming languages. It evaluated the given decision variable and based on its value enables the nested element.

**Syntax:**

```java
@AlternativeGroup (configured variability attribute)
    @Alternative(alternative value 1)
        service ToDo.Run();
    @Alternative(alternative value 2)
        service ToDo.Run();
@/AlternativeGroup
```

**Description of Syntax:** the definition of an alternative element consists of the following elements:

- The `@AlternativeGroup` keyword indicates the definition of a new alternative element.
- The identifier `configured variability attribute` inside the brackets represents variable which value needs to be validated.
- The set of elements following previous syntax defines the nested elements of the alternative element. Depending on the matching of `alternative value x` to the value of `configured variability attribute`, related nested elements will be enabled.
- The `@/AlternativeGroup` keyword indicates borders of an alternative element.

**Attributes:** configured variability attribute, alternative value, nested elements

**Nested elements:** service, transition, start, state, end, decision.

**Example:**

```java
@AlternativeGroup (deploymentTarget)
```
2.3 Built-in operations

Similar to OCL, in the vConcept language all operations are defined on individual vConcept types and can be accessed using the “.” operator, such as `set.size()`. However, this is also true for the equality, relational and mathematical operators but they are typically given in alternative infix notation, i.e. `1 + 1` instead of `1.+(1)`.

In this section, I denote the actual type on which an individual operation is defined as the operand of the operation (called self in OCL). The parameters of an operation are given in parenthesis. Further, similar to the declaration of decision variables in vConcept, I use in this section the Type-first notation to describe the signatures of the operation.

2.3.1 Internal Types

2.3.1.1 AnyType

AnyType is the most common type in the vConcept type system. All types in vConcept are subclasses of AnyType, i.e. they are type compliant and inherit the operations listed below.

- **Boolean == (AnyType a)**
  True if the operand is the same as a. This operation is interpreted as a value assertion if it is used standalone (empty implication) or on the right side of an implication. It is interpreted as an equality test if used on the left side of an implication.

- **Boolean <> (AnyType a)**
  True if the operand is different from a.

- **Boolean != (AnyType a)**
  True if the operand is a different object from a. Alias for !=.

- **MetaTypetypeOf ()**
  The type information of the actual type.

- **Boolean isTypeOf (MetaType type)**
  True if the type and the actual type of operand are the same. This operation can be seen as an alias for `typeOf() == type`.

- **Boolean isKindOf (MetaType type)**
  True if type is either the direct type or one of the supertypes of the actual type of the operand.

2.3.2 Basic Types

2.3.2.1 Real

The basic type Real represents the mathematical concept of real following the Java range restrictions for double values. Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter.
• **Real + (Real r)**
  The value of the addition of *self* and the *operand*.

• **Real - (Real r)**
  The value of the subtraction of *r* from the *operand*.

• **Real * (Real r)**
  The value of the multiplication of the *operand* and *r*.

• **Real - ()**
  The negative value of the *operand*.

• **Real / (Real r)**
  The value of the *operand* divided by *r*. Leads to an evaluation error if *r* is equal to zero.

• **Real abs()**
  The absolute value of the *operand*.

• **Integer floor ()**
  The largest integer that is less than or equal to the *operand*.

• **Integer round()**
  The integer that is closest to the *operand*. When there are two such integers, the largest one.

• **Real max (Real r)**
  The maximum of the *operand* and *r*.

• **Real min (Real r)**
  The minimum of the *operand* and *r*.

• **Boolean < (Real r)**
  True if the *operand* is less than *r*.

• **Boolean > (Real r)**
  True if the *operand* is greater than *r*.

• **Boolean <= (Real r)**
  True if the *operand* is less than or equal to *r*.

• **Boolean >= (Real r)**
  True if the *operand* is greater than or equal to *r*.

• **Boolean = (Real r)**
  Assigns the value *r* to the variable *operand* and returns true.

2.3.2.2 Integer
The standard type Integer represents the mathematical concept of integer following the Java range restrictions for integer values. Note that Integer is a subclass of Real.

• **Integer - ()**
  The negative value of the *operand*.

• **Integer + (Integer i)**
  The value of the addition of the *operand* and *i*.

• **Integer - (Integer i)**
  The value of the subtraction of *i* from the *operand*.

• **Integer * (Integer i)**
  The value of the multiplication of the *operand* and *i*.

• **Real / (Integer i)**
The value of the operand divided by \( i \). Leads to an evaluation error if \( i \) is equal to zero.

- **Integer abs()**
  The absolute value of the operand.

- **Integer div (Integer i)**
  The number of times that \( i \) fits completely within the operand.

- **Integer mod (Integer i)**
  The result is the operand modulo \( i \).

- **Integer max (Integer i)**
  The maximum of the operand and \( i \).

- **Integer min (Integer i)**
  The minimum of the operand and \( i \).

- **Boolean < (Integer i)**
  True if the operand is less than \( i \).

- **Boolean > (Integer i)**
  True if the operand is greater than \( i \).

- **Boolean <= (Integer i)**
  True if the operand is less than or equal to \( i \).

- **Boolean >= (Integer i)**
  True if the operand is greater than or equal to \( i \).

- **Boolean = (Integer i)**
  Assigns the value \( i \) to the operand and returns true.

2.3.2.3 Boolean
The basic type Boolean represents the common true/false values.

- **Boolean or (Boolean b)**
  True if either self or \( b \) is true.

- **Boolean xor (Boolean b)**
  True if either self or \( b \) is true, but not both.

- **Boolean and (Boolean b)**
  True if both \( b1 \) and \( b \) are true.

- **Boolean not ()**
  True if self is false and vice versa.

- **Boolean implies (Boolean b)**
  True if self is false, or if self is true and \( b \) is true. The rightmost implication is interpreted as an assertion of the right side of the expression. Further implications on the left side of an implication as well as implication in a Boolean expression are just evaluated to a Boolean value.

- **Boolean iff (Boolean b)**
  Shortcut for (a.implies(b) and b.implies(a)).

- **Boolean = (Boolean b)**
  Assigns the value \( b \) to the operand and returns true.

2.3.2.4 String
The standard type String represents strings, which can be ASCII.
• **Integer size ()**
  The number of characters in the operand.

• **String concat (String s)**
  The concatenation of the operand and s.

• **String substring (Integer lower, Integer upper)**
  The sub-string of the operand starting at character number lower, up to and including character number upper. Character numbers run from 0 to size().

• **Boolean matches (String r)**
  Returns whether the operand matches the regular expression r. Regular expressions are given in the Java regular expression notation. For example, the following operation will check whether mail is a valid e-mail-address:
  ```java
  mail.matches(\w*@[\w]*.\w*);
  ```

• **Boolean substitutes (String r, String s)**
  Replaces all occurrences of the regular expression r in the operand by s. Regular expressions are given in the Java regular expression notation. For example, the following operation will substitute the occurrence of “@” with “{at}” in an e-mail-address:
  ```java
  mail.substitutes(“@”, “{at}”);
  ```

• **Integer toInteger ()**
  Converts the operand to an Integer value.

• **Real toReal ()**
  Converts the operand to a Real value.

• **Boolean = (String s)**
  Assigns the value s to the operand and returns `true` Error! Bookmark not defined.

2.3.3 Enumeration Types
Enumerations allow the definition of sets of named values.

2.3.3.1 Enum
Enums inherit all operations from AnyType and adds the following operation:

• **Boolean = (Enum e)**
  Assigns the value e to the operand and returns `trueError! Bookmark not defined..`

2.3.3.2 OrderedEnum
In contrast to Enums, individual ordinal values for the literals in an OrderedEnum are specified. Thus, an OrderedEnum defines a (total) ordering on its literals so that further operations in addition to those defined for Enum are available.

• **Boolean < (OrderedEnum l)**
  True if the operand is less than the ordinal value of the literal l.

• **Boolean > (OrderedEnum l)**
  True if the operand is greater than the ordinal value of the literal l.

• **Boolean <= (OrderedEnum l)**
  True if the operand is less than or equal to the ordinal value of the literal l.

• **Boolean >= (OrderedEnum l)**
  True if the operand is greater than or equal to the ordinal value of the literal l.
2.3.4 Collection Types
This section defines the operation of the collection types. The two vConcept collections Set and Sequence are both subtypes of the abstract collection type Collection. Each collection type is actually a template type with one parameter. ‘T’ denotes the parameter. A concrete collection type is created by substituting a type for the T. So a collection of integers is referred in vConcept by setOf(Integer).

2.3.4.1 Collection
Collection is the abstract superclass of all collections in vConcept.

- **Integer size ()**
  The number of elements in the collection operand.

- **Boolean includes (T object)**
  True if object is an element of operand, false otherwise.

- **Boolean excludes (T object)**
  True if object is not an element of operand, false otherwise.

- **Integer count (T object)**
  The number of times that object occurs in the collection operand.

- **Boolean isEmpty ()**
  Is the operand the empty collection?

- **Boolean notEmpty ()**
  Is the operand not the empty collection?

- **Boolean isDefined()**
  Returns whether (a variable of) the operand is defined, i.e. that an instance was already assigned.

- **T sum()**
  The addition of all elements in the operand. Elements must be of a type supporting the + operation (Integer or Real).

- **T product()**
  The multiplication of all elements in the operand. Elements must be of a type supporting the * operation (Integer or Real).

- **T min()**
  The minimum of all elements in the operand. Elements must be of a type supporting the < operation (Integer or Real).

- **T max()**
  The minimum of all elements in the operand. Elements must be of a type supporting the > operation (Integer or Real).

- **T avg()**
  The average of all elements in the operand. Elements must be of a type supporting the / operation (Integer or Real).

- **Boolean forAll (Iterators | expression)**
  Results in true if expression evaluates to true for each element in the operand collection.

- **Boolean exists (Iterators | expression)**
  Results in true if expression evaluates to true for at least one element in the operand collection.
- **Boolean isUnique (Iterator | expression)**
  Results in true if expression evaluates to a different value for each element in the operand collection; otherwise, result is false. isUnique may have at most one iterator variable.

- **T any (Iterator | expression)**
  Returns any element in the source collection for which expression evaluates to true. If there is more than one element for which expression is true, one of them is returned. any may have at most one iterator variable.

- **Boolean one (Iterator | expression)**
  Results in true if there is exactly one element in the operand collection for which expression is true. one may have at most one iterator variable.

- **Collection<T> collect (Iterator | expression)**
  The Collection of elements that results from applying expression to every member of the source set. collect may have at most one iterator variable.

- **Collection<T> select (Iterator | expression)**
  The sub-collection for which expression is true. select may have at most one iterator variable.

- **Boolean reject (Iterator | expression)**
  The sub-collection for which expression is false. reject may have at most one iterator variable.

- **<R> apply (Iterator, R result | result = expression)**
  Applies the given expression to the operand collection using the specified iterator and stores the result in the last iterator (used here as a local variable declaration) which is returned as the result of this operation. Expression shall use the result “iterator” for aggregating values. Apply may have at most one iterator variable and needs to specify the result “iterator”.

2.3.4.2 Set
The Set is the mathematical set. It contains elements without duplicates. Set inherits the operations from Collection.

- **Boolean == (Set<T> s)**
  Evaluates to true if operand and s contain the same elements.

- **Set<T> union (Set<T> s)**
  The union of operand and s.

- **Set<T> intersection (Set<T> s)**
  The intersection of operand and s (i.e., the set of all elements that are in both operand and s).

- **Set<T> excluding (T object)**
  The set containing all elements of operand without object.

- **Set<T> including (T object)**
  The set containing all elements of operand plus object.

- **Set<T>asSet ()**
  A Set identical to operand. This operation exists for convenience reasons.

- **Sequence<T>asSequence ()**
  A Sequence that contains all the elements from operand, in undefined order.

- **Set<T>typeSelect (MetaType T)**
  Results the subset of elements from operand which are of type T.
• **Set<T>** typeReject (MetaType T)
  Results the subset of elements from operand which are not of type T.

• **Boolean = (Set<T> s)**
  Assigns the value s to the operand and returns trueError! Bookmark not defined..

2.3.4.3 Sequence
A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once. Sequence inherits the operations from Collection.

• **Boolean == (Sequence<T> s)**
  Evaluates to true if operand and s contain the same elements.

• **Sequence<T> union (Sequence<T> s)**
  The union of operand and s.

• **Set<T>asSet ()**
  The Set containing all the elements from operand, with duplicates removed.

• **Sequence<T>asSequence ()**
  The Sequence identical to the operand itself. This operation exists for convenience reasons.

• **T at (Integer i)**
  The i-th element of the sequence operand. Valid indices run from 0 to size()-1.

• **T [] (Integer i)**
  The i-th element of the sequence operand. This operation is an alias for at. Valid indices run from 0 to size()-1.

• **T first ()**
  The first element in operand.

• **T last()**
  The last element in operand.

• **Sequence<T> append (T object)**
  The sequence of elements, consisting of all elements of operand, followed by object.

• **Sequence<T> prepend(T object)**
  The sequence consisting of object, followed by all elements in operand.

• **Sequence<T>insertAt(Integer index, T object)**
  The sequence consisting of operand with object inserted at position index. Valid indices run from 0 to size()-1.

• **Integer indexOf(T object)**
  The index of object object in the sequence operand.

• **Sequence<T>typeSelect (MetaType T)**
  Results the subset of elements from operand which are of type T.

• **Sequence<T>typeReject (MetaType T)**
  Results the subset of elements from operand which are not of type T.

• **Boolean = (Sequence<T> s)**
  Assigns the value s to the operand and returns trueError! Bookmark not defined..
2.3.5 Compound Types

A compound type groups multiple types into a single named unit. A compound
inherits all its operations from AnyType. Access to variable declarations within a compound
are specified using ".". Using the type name of the compound on the left side of a "." is a
shortcut for an all-quantification on all instances of that compound. In addition, it defines
the following operation:

- **Boolean isDefined()**
  
  Returns whether (a variable of) the operand is defined, i.e. that an instance
  was already assigned.

- **Boolean = (Compund c)**
  
  Assigns the value c to the operand and returns true.
3 Proof of concept with running example

In this section I will try to provide proof of concept for vConcept and variability resolution at compile-time. The vConcept language is described in Section 2 and the related theoretical background is covered in Section 1.

In Section 3.1 prerequisites for using vConcept in Eclipse are described.

In Section 3.2 some features of tool support for the vConcept in Eclipse environment are provided.

In Section 3.3 I will describe the running example of business process management system using a content-sharing platform case. The emphasis is on the variability in this system.

In Section 3.4 variability of a content-sharing system, described in section 3.3, is modelled using vConcept. Also the variability resolution during compile-time result is provided.

In Section 3.5 I will provide examples of the same business process used for vConcept model implemented in main business process automation engine languages – Activiti, jBPM and Spring Web flow. Each example will contain a model with runtime variability and a model of business process with variability resolved at compile-time.

3.1 Prerequisites for running vConcept in Eclipse

vConcept is developed as an Eclipse\(^1\) plug-in and requires Xtext\(^4\) version 2.3.1. Thus, in general, any Eclipse installation with Xtext version 2.3.1 is fine for installing and running vConcept. However, developers do not guarantee that any combination of Eclipse and Xtext version 2.3.1 will work together. Thus, the following Eclipse versions are recommended as they are tested with Xtext version 2.3.1:

- Eclipse 3.6 (Helios)
- Eclipse 3.7 (Indigo)
- Eclipse 4.0 (Juno)

I recommend using Eclipse 3.7 (Indigo) as this is the most exhaustively tested version of Eclipse with Xtext version 2.3.1. Download an Eclipse package from http://www.eclipse.org/downloads/.

Please note that Eclipse 4.2 does not work with Xtext 2.3.1 due to incompatible dependencies.

Further, Xtext version 2.3.1 has to be installed in the newly downloaded Eclipse instance.

After installation was completed successfully vConcept projects must be imported into Eclipse workspace.

---

\(^1\)Eclipse website: www.eclipse.org/

\(^4\)Xtext website: http://www.eclipse.org/Xtext/
To run vConcept as a plug-in, create a configuration as **Eclipse Application** with default parameters.

![vConcept projects](image)

**Figure 3.1:** vConcept projects.

**Figure 3.2:** Run configuration creation in Eclipse.

### 3.2 Tool support for vConcept

**Creating a new model**

1. File -> New -> Other -> File
2. Define the container for the vConcept file (at this point there is no strict project structure) and the name of the file (files extension must be *vconcept*)
3. A new modelling file is created.

**Syntax colouring**

Java style syntax colouring is implemented for vConcept (Figure 3.3).

**Outline view**

Dynamic outline view with description of all elements is implemented (Figure 3.3).
3.3 Running example of a business process management system

The example will model the variability of the instantiation and deployment of a content-sharing application. A content-sharing application allows its users to upload, annotate, release and share content of various types. In this example, concrete applications may differ with respect to:

- The **supported content types** such as text, video, audio, 3D content.
- The **hosting infrastructure** which consists of a) a web container being responsible for serving the content and b) the database, which stores user and content data.
- The **deployment target**, which may either, be a traditionally hosted server or a cloud environment. The cloud environment may be private, like a local installation of the Eucalyptus\(^5\) cloud software or public, in this example we will allow from the Amazon\(^6\) or Azure\(^7\) cloud.

---

\(^5\)http://www.eucalyptus.com

\(^6\) http://aws.amazon.com/ec2

\(^7\) http://www.windowsazure.com
Without going into functional details of the content-sharing application, the variability introduced by content types, web container, database and deployment target allow to derive a large number of different application instances. Whether it is possible to describe a specific set of applications at all or whether it is possible to state individual details depends on the expressiveness of the languages used for modelling variability and dependencies.

Dependencies below:
1) At least one content type must be present as otherwise the content-sharing platform is useless.
2) The combination of supported content types may be restricted based on the capabilities of the web container or the deployment platform, e.g. due to load problems only a limited number of content types may be available on the traditional deployment target.
3) Additional charges on data traffic to Amazon and Azure cloud services must be accounted for.

Here is a visual variability model of a content-sharing system:

![Variability Model](image)

**Composition rules:**
- Traditional requires MySQL, Cloud excludes MySQL, Eucalyptus requires Amazon S3,
- Amazon requires Amazon S3, Azure requires Azure SQL

Next step is to focus on a simple content upload business process and model it in different languages (all based on BPMN) without variability resolution (3.2.1 – 3.2.3). When this is done, vConcept can be used to create a variability model described in the running example and model a workflow defining variability and then providing a derived workflow with resolved variability (3.4).

### 3.4 Variability modelling with vConcept

Below I created a variability model of the content-sharing system using vConcept language. The model consists of both core variability, described in running example
requirements, and the example business process of content uploading with implemented variability.

model ContentSharingPlatform {

    // Conceptual model
    enum ContentType {Text, Video, Audio, ThreeD};
    enum ContainerType {Tomcat, IIS, JBoss};
    enum DatabaseType {AzureSQL, AmazonS2, MySQL};
    enum DeploymentTarget {Traditional, Eucalyptus, Amazon, Azure};

    ContainerType containerType;
    containerType != null;
    DatabaseType databaseType;
    databaseType != null;
    DeploymentTarget deploymentTaget;
    deploymentTarget != null;

    compound Content {
        ContentType type;
    }

    compound VideoContent refines Content {
        Content.type = ContentType.Video;
        Integer bitrate;
    }
    VideoContent videoContent;

    compound ThreeD refines Content {
        Content.type = Content.TreeD;
    }
    ThreeD threeD;

    compound Audio refines Content {
        Content.type = Content.Audio;
    }
    Audio audio;

    compound Text refines Content {
        Content.type = Content.Text;
    }
    Text text;

    setOf(Content) providedContent;

    providedContent.length> 0;
    providedContent.length> 2 implies deploymentTarget != DeploymentTarget.Traditional;
    deploymentTarget == DeploymentTarget.Traditional implies databaseType == DatabaseType.MySQL;
    deploymentTarget != DeploymentTarget.Traditional implies databaseType != DatabaseType.MySQL;
    (deploymentTarget == DeploymentTarget.Eucalyptus or deploymentTarget == DeploymentTarget.Azure) implies databaseType == DatabaseType.Azure;
    deploymentTarget == DeploymentTarget.Amazon implies databaseType == DatabaseType.Amazon;

    // Configuration
    videoContent.bitrate = 128;
}
providedContent = {audio, text};
containerType = ContainerType.Tomcat;
databaseType = DatabaseType.AmazonS2;
deploymentTarget = DeploymentType.Amazon;

// Workflow
workflow ContentUploading {

source nl.ru.fnwi.ContentSharingPlatform.LogIn;
source nl.ru.fnwi.ContentSharingPlatform.SelectContent;
source nl.ru.fnwi.ContentSharingPlatform.UploadContent;
source nl.ru.fnwi.ContentSharingPlatform.TrafficCounter;

start ContentUploadingStart {
    transition to LogIn;
}

state LogIn {
    service logIn.execute();
    transition to SelectContent;
}

state SelectContent {
    service selectContent.execute();
    transition to UploadContent;
}

state UploadContent {
    service uploadContent.execute();

    @Optionality (deploymentTarget == DeploymentTarget.Azure or
    deploymentTarget == DeploymentTarget.Amazon;)
    transition to CountTraffic;
    @/Optionality

    @Optionality (deploymentTarget == DeploymentTarget.Traditional
    or deploymentTarget == DeploymentTarget.Eucalyptus;)
    transition to ContentUploadingEnd;
    @/Optionality
}

state CountTraffic {
    @AlternativeGroup (deploymentTarget)
    @Alternative (DeploymentTarget.Amazon)
    service trafficCounter.amazonCounter();
    @Alternative (DeploymentTarget.Azure)
    service trafficCounter.azureCounter();
    @/AlternativeGroup
    transition to ContentUploadingEnd;
}

end ContentUploadingEnd {
}
}

After the variability is resolved compile-time, it is possible to derive a product for the
specific configuration, in this specific case, deploying the platform to Amazon cloud
(configuration part of the vConcept model). Below a code sample of the derived business process is illustrated (workflow part of the vConcept model):

```java
workflow ContentUploading {
    start ContentUploadingStart {
        transition to LogIn;
    }
    state LogIn {
        service logIn.execute();
        transition to SelectContent;
    }
    state SelectContent {
        service selectContent.execute();
        transition to UploadContent;
    }
    state UploadContent {
        service uploadContent.execute();
        transition to CountTraffic;
    }
    state CountTraffic {
        service trafficCounter.amazonCounter();
    }
    end ContentUploadingEnd {
    }
}
```

### 3.5 Variability resolution examples with BPMS engines

As vConcept is only a modelling language an underlying business process management system is needed to automate business process execution. For this reason configurations of 3 most wide spread BPMS engines are used in following example. For each of given engines (Activiti, jBPM, Spring Web Flow) 2 type of business processes are modelled. At first, a business process with runtime variability is modelled. That involves a usage of control flow elements and decisions are made during business process execution. Second model is a business process with variability resolved at compile-time. That means the derived business process already has most of decisions resolved.

In a small-scale example difference in business process complexity and execution time is not so obvious as in large-scale industrial cases, but it should give an overview of improvements possible.

#### 3.5.1 Business process model with Activiti

Runtime variability:
```xml
<definitions id="definitions"/>
```

---

*Business Process Management System*
targetNamespace="http://activiti.org/bpmn20"
xmlns:activiti="http://activiti.org/bpmn"
xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL">

<process id="ContentUploading">

<startEvent id="ContentUploadingStart"/>

<sequenceFlow id='flow1' sourceRef='ContentUploadingStart' targetRef='LogIn'/>
<serviceTask id="LogIn" activiti:expression="#{logIn.service()}"/>
<sequenceFlow id='flow2' sourceRef='LogIn' targetRef='SelectContent'/>
<serviceTask id="SelectContent" activiti:expression="#{selectContent.service()}"/>
<sequenceFlow id='flow3' sourceRef='SelectContent' targetRef='UploadContent'/>
<serviceTask id="UploadContent" activiti:expression="#{uploadContent.service()}">

<sequenceFlow id='flow4' sourceRef='UploadContent' targetRef='TraficCounterEvaluator'/>

<exclusiveGateway id="TraficCounterEvaluator"/>

<sequenceFlow id='flow5' sourceRef='TraficCounterEvaluator' targetRef='AmazonCountTraffic'>
<conditionExpression xsi:type="tFormalExpression">${deploymentTarget == DeploymentTarget.Amazon}</conditionExpression>
</sequenceFlow>

<sequenceFlow id='flow6' sourceRef='TraficCounterEvaluator' targetRef='AzureCountTraffic'>
<conditionExpression xsi:type="tFormalExpression">${deploymentTarget == DeploymentTarget.Azure}</conditionExpression>
</sequenceFlow>

<sequenceFlow id='flow7' sourceRef='TraficCounterEvaluator' targetRef='ContentUploadingEnd'>
<conditionExpression xsi:type="tFormalExpression">${deploymentTarget == DeploymentTarget.Traditional}</conditionExpression>
</sequenceFlow>

<sequenceFlow id='flow8' sourceRef='TraficCounterEvaluator' targetRef='ContentUploadingEnd'>
<conditionExpression xsi:type="tFormalExpression">${deploymentTarget == DeploymentTarget.Eucalyptus}</conditionExpression>
</sequenceFlow>

<serviceTask id="AmazonCountTraffic" activiti:expression="#{trafficCounter.amazonCounter()}"/>
<sequenceFlow id='flow9' sourceRef='AmazonCountTraffic' targetRef='ContentUploadingEnd'/>

<serviceTask id="AzureCountTraffic" activiti:expression="#{trafficCounter.azureCounter()}"/>
Variability resolved at compile-time:

```xml
<process id="ContentUploading">
  <start name="ContentUploadingStart">
    <transition to="LogIn"/>
  </start>
  <java name="LogIn"
    class="nl.ru.fnwi.ContentSharingPlatform.LogIn"
    method="logIn.execute()">
    <on event="ok">
      <event-listener class="nl.ru.fnwi.ContentSharingPlatform.LogIn">
        <transition to="SelectContent"/>
      </event-listener>
    </on>
  </java>
  <sequenceFlow id='flow3' sourceRef='SelectContent' targetRef='UploadContent'/>
  <serviceTask id="UploadContent" activiti:expression="#{uploadContent.execute()}"
    targetRef='AmazonCountTraffic'/>
  <serviceTask id="AmazonCountTraffic" activiti:expression="#{trafficCounter.amazonCounter()}/">
    <sequenceFlow id='flow8' sourceRef='AmazonCountTraffic' targetRef='ContentUploadingEnd'/>
  </serviceTask>
</process>
```

### 3.5.2 Business process model with jBPM

Runtime variability:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<process name="New entry" xmlns="http://jbpm.org/4.4/jpdl">
  <start name="ContentUploadingStart">
    <transition to="LogIn"/>
  </start>
  <java name="LogIn"
    class="nl.ru.fnwi.ContentSharingPlatform.LogIn"
    method="logIn.execute()">
    <on event="ok">
      <event-listener class="nl.ru.fnwi.ContentSharingPlatform.LogIn">
        <transition to="SelectContent"/>
      </event-listener>
    </on>
  </java>
  <sequenceFlow id='flow2' sourceRef='LogIn' targetRef='SelectContent'/>
  <serviceTask id="SelectContent" activiti:expression="#{selectContent.execute()}"/>
  <sequenceFlow id='flow4' sourceRef='SelectContent' targetRef='UploadContent'/>
  <serviceTask id="UploadContent" activiti:expression="#{uploadContent.execute()}"/>
  <sequenceFlow id='flow5' sourceRef='UploadContent' targetRef='AmazonCountTraffic'/>
  <serviceTask id="AmazonCountTraffic" activiti:expression="#{trafficCounter.amazonCounter()}/">
    <sequenceFlow id='flow9' sourceRef='AmazonCountTraffic' targetRef='ContentUploadingEnd'/>
  </serviceTask>
</process>
```
method="selectContent.execute()">
<on event="ok">
<event-listener class="nl.ru.fnwi.ContentSharingPlatform.SelectContent">
<transition to="UploadContent"/>
</event-listener>
</on>
</java>

<java name="UploadContent" class="nl.ru.fnwi.ContentSharingPlatform.UploadContent" method="uploadContent.execute()">
<on event="ok">
<event-listener class="nl.ru.fnwi.ContentSharingPlatform.UploadContent">
<transition to="TraficCounterEvaluator"/>
</event-listener>
</on>
</java>

decision name="TraficCounterEvaluator" expr="#{configuration.getDeploymentTarget()}">
<handler class="nl.ru.fnwi.Configuration"/>
<transition on="Amazon" to="AmazonCountTraffic"/>
<transition on="Azure" to="AzureCountTraffic"/>
<transition on="Traditional" to="ContentUploadingEnd"/>
<transition on="Eucalyptus" to="ContentUploadingEnd"/>
</decision>

<java name="AmazonCountTraffic" class="nl.ru.fnwi.ContentSharingPlatform.TrafficCounter" method="trafficCounter.amazonCounter()">
<on event="ok">
<event-listener class="nl.ru.fnwi.ContentSharingPlatform.TrafficCounter">
<transition to="ContentUploadingEnd"/>
</event-listener>
</on>
</java>

<java name="AzureCountTraffic" class="nl.ru.fnwi.ContentSharingPlatform.TrafficCounter" method="trafficCounter.azureCounter()">
<on event="ok">
<event-listener class="nl.ru.fnwi.ContentSharingPlatform.TrafficCounter">
<transition to="ContentUploadingEnd"/>
</event-listener>
</on>
</java>

<end name="ContentUploadingEnd"/>
</process>

Variability resolved at compile-time:

<process name="Newentry" xmlns="http://jbpm.org/4.4/jpdl">
<start name="ContentUploadingStart">
<transition to="LogIn"/>
</start>

<java name="LogIn" class="nl.ru.fnwi.ContentSharingPlatform.LogIn"
3.5.3 Business process model with Spring Web Flow

Runtime variability:

```xml
<flow version="1.0" encoding="UTF-8">
<on-start>
  <transition on="submit" to="LogIn"/>
</on-start>

<action-state id="LogIn">
  <evaluate expression="LogIn.execute()"/>
  <transition on="success" to="SelectContent"/>
</action-state>
</flow>
```
</action-state>

<action-state id="SelectContent">
<evaluate expression="selectContent.execute()"/>
<transition on="success" to="UploadContent"/>
</action-state>

<action-state id="UploadContent">
<evaluate expression="uploadContent.execute()"/>
<transition on="success" to="TrafficCounterEvaluator"/>
</action-state>

<action-state id="TrafficCounterEvaluator">
<evaluate expression="configuration.getDeploymentTarget()"/>
<transition on="Amazon" to="AmazonCountTraffic"/>
<transition on="Azure" to="AzureCountTraffic"/>
<transition on="Traditional" to="ContentUploadingEnd"/>
<transition on="Eucalyptus" to="ContentUploadingEnd"/>
</action-state>

<action-state id="AmazonCountTraffic">
<evaluate expression="trafficCounter.amazonCounter()"/>
<transition on="success" to="ContentUploadingEnd"/>
</action-state>

<action-state id="AzureCountTraffic">
<evaluate expression="trafficCounter.azureCounter()"/>
<transition on="success" to="ContentUploadingEnd"/>
</action-state>

<end-state id="ContentUploadingEnd"/>

</flow>

Variability resolved at compile-time:

<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.springframework.org/schema/webflow
                         http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">
  <on-start>
    <transition on="submit" to="LogIn"/>
  </on-start>

  <action-state id="LogIn">
    <evaluate expression="logIn.execute()"/>
    <transition on="success" to="SelectContent"/>
  </action-state>

  <action-state id="SelectContent">
    <evaluate expression="selectContent.execute()"/>
    <transition on="success" to="UploadContent"/>
  </action-state>

  <action-state id="UploadContent">
    <evaluate expression="uploadContent.execute()"/>
    <transition on="success" to="AmazonCountTraffic"/>
  </action-state>
</flow>
<action-state>  
<action-state id="AmazonCountTraffic">  
<evaluate expression="trafficCounter.amazonCounter()"/>  
<transition on="success" to="ContentUploadingEnd"/>  
</action-state>  
</action-state>  
<end-state id="ContentUploadingEnd"/>  
</flow>
Conclusions

In this thesis I have presented an approach to modelling and implementing variability in Business Processes. To achieve this goal, a set of tasks was identified in the introduction and they all were successfully accomplished.

To provide a theoretical base for this work a considerable amount of literature and practical studies was performed. This resulted in findings described in Section 1. As software product line engineering was a new topic for me, this gave an opportunity not only to study variability in business processes, but to get an understanding of a much broader topic, that could give some benefits in the future and open new opportunities in research and career.

The main amount of work was dedicated to creating a domain specific language that would allow modelling variability in business processes and business process management systems. Overall, vConcept meet main requirements set by the industry (main ones are quoted in the introduction) even as it was only implemented on a conceptual level:

- **R1**: It is possible to model variability both on core level and specifically in business processes. Modelling language is based on well-known structures like data types from Object Oriented programming languages, constraints from OCL expressions and business process modelling elements from BPMN and UML Activity diagram. This is something both developers and analysts are familiar with and will be able to relate to. Modelling simplicity is an arguable question, but with certain expertise in notation mentioned above and modelling itself potential users will be able to successfully put vConcept language to use.

- **R2**: Usage of domain specific language allowed distancing modelling from the underlying business process automation framework. As a result, the same business process model could be a basis for business process implementations with Activiti, JBoss jBPM, Spring Work Flow technologies. As all of these technology providers base their business process configurations on BPMN elements it is possible to map them to vConcept. Thus, allowing fulfilling **R5**.

- **R3**: As technologies used in business process automation (technology providers are mention in **R2**) are open-source products and development would take place in an open-source environment, bringing modelling into Eclipse proved to be a successful idea. With vConcept, model is no detached from the implementation and is not limited by 3rd party tools. Also, Xtext provided broad capabilities for domain specific language implementation and development environment support like code colouring, outline view and content assist.

- **R4**: As the main goal of modelling a business process is to automate it in the future, each process elements (state) would have a dedicated functionality attached to it. In other words, a related Java method from a specific class may be triggered. This concept is supported in vConcept (source and service elements). So pre-developed functionality could be modelled as variability and later reused in derived products.

- **R5**: Automation of the variability resolution and implementation of an automated business process would be final requirements that the industry are trying to bring to life. In the context of the thesis this is only looked as
from a conceptual level, as theoretical part describes the way of domain engineering and application engineering and in the running example the outcome of such practice is shown.

This is a good starting point for an on-going research. For this 2 main future research directions can be identified:

• Broadening the scope of covered modelling concepts in vConcept (corresponding to UML Activity diagrams, BPMN and others);
• Implementing implementation functionality (code generation, concept mapping, supported BPMS).

Both directions would require a considerable amount of effort and could end up with a PhD thesis.

Another strong advantage of the work done is a real-life prototype. A modelling language that is not just a on the specification level but is implemented for use in Eclipse. Something that can be demonstrated and experimented with is always better than a plain text. Section 3 illustrates hoe vConcept is used in action.

As this is a project aimed to fit industrial needs, downsides could not be avoided. First of all, rapidly changing requirements from the industry make it rather hard to conduct a fundamental research, as every customers case is individual and time to market is critical, industrial partners are not very interested to invest a lot of time and resources in preparation phases. Expected volume and market depth would not cover all these expenses, so without a strong academic and governmental support this kind of project is unlikely to be ended. Second problem is rather technical, as used tools are open source and not connected to each other explicitly, getting them to operate together is a tricky task. For example, migrating from one version of Xtext to another would end up in language reengineering, as Xtext change dramatically and no detailed documentation on changes is provided by the developers. Achieving BPMS configuration like Activiti, JBoss jBPM and Spring Web Flow to be generated automatically and be deployed without human interference is a problem not tackled yet and would bring a big challenge in the future.

To finish on a positive note, I would like to stress out that the idea introduces in the thesis is innovative in nature and demanded by the market. First steps into the topic proved to be successful and could be used in a future study and development.
Bibliography


**A.1  Appendix 1 – vConcept source code**

```plaintext
grammar nl.ru.fnwi.vconcept.VConcept with org.eclipse.xtext.common.Terminals

generate vConcept "http://www.ru.nl/fnwi/vconcept/VConcept"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Model:
  'model' name=Identifier
  '{'
  imports+=ImportStmt*
  content=Content
  '}'? ;

Content:
  {Content}
  |
  elements+=Typedef
  |
  elements+=VariableDeclaration
  |
  elements+=ExpressionStatement
  |
  elements+=Workflow
)* ;

ImportStmt:
  'import'
  name=QualifiedNameWithWildCard
  ';' ;

//------------------------------------ Types -----------------------------

Typedef:
  tEnum=TypedefEnum
  | tCompound=TypedefCompound
  | tMapping=TypedefMapping
;

TypedefEnum:
  'enum'
  name=Identifier
  '{'
  literals+=TypedefEnumLiteral
  (','
   literals+=TypedefEnumLiteral
  )* 
  '}'
  constraint=TypedefConstraint?
  ';' ;

TypedefEnumLiteral:
  name=Identifier
  ( '==' value = NumValue
```
typedef Compound:
'compound'
name=Identifier
{
   'refines'
super=Identifier
}
{" 
   (elements+=VariableDeclaration
    | elements+=ExpressionStatement
   )* 
}{"';'?
;

typedef Mapping:
'typedef'
newType=Identifier
type=Type
constraint=TypedefConstraint?
';'
;

typedef Constraint:
'with'
'{
   expressions += Expression
   ( 
      '
      expressions += Expression
   )*
   '}
;

variable Declaration:
   type=Type
decls+=VariableDeclarationPart
   ( 
      ,
      decls+=VariableDeclarationPart
   )* 
';'
;

Variable Declaration Part:
   name=Identifier ('=' default=Expression)?
;

Basic Type:
   type='Integer'
   | type='Real'
   | type='Boolean'
   | type='String'
   | type='Constraint'
;

Type:
type=BasicType
  | id=QualifiedName
  | derived=DerivedType
  
NumValue :
  val=NUMBER
  
QualifiedName :
  ( qName+=Identifier
    qName+='::'
    ( qName+=Identifier
      qName+='::'
    )* )
  qName+=Identifier
  
Value :
  nValue = NumValue
  | sValue = STRING
  | qValue = VarQualifiedName
  | bValue = ('true' | 'false')
  | mValue = MethodQualifiedName
  | nullValue = 'null'
  
DerivedType :
  ( op='setOf'
    | op='sequenceOf'
  )
   '{'
   type=Type
   '}'
  
//------------------------------------ OCL ----------------------------

ExpressionStatement:
  expr=Expression
  ';'
  
Expression:
  expr=ImplicationExpression
  | collection=CollectionInitializer
  
AssignmentExpression:
  left=LogicalExpression
  right+=AssignmentExpressionPart?
  
AssignmentExpressionPart:
  op=AssignmentOperator
(ex=LogicalExpression | collection=CollectionInitializer)
;
AssignmentOperator: 
'=' 
;
ImplicationExpression:
left=AssignmentExpression
right+=ImplicationExpressionPart* 
;
ImplicationExpressionPart: 
op=ImplicationOperator
ex=AssignmentExpression 
;
ImplicationOperator: 
'implies' 
| 'iff' 
;
LogicalExpression:
left=EqualityExpression
right+=LogicalExpressionPart* 
;
LogicalExpressionPart: 
op=LogicalOperator
ex=EqualityExpression 
;
LogicalOperator: 
'and' 
| 'or' 
| 'xor' 
;
EqualityExpression:
left=RelationalExpression
right=EqualityExpressionPart?
;
EqualityExpressionPart: 
op=EqualityOperator
(ex=RelationalExpression | collection=CollectionInitializer) 
;
EqualityOperator: 
'==' 
| '<>' 
| '!=' 
;
RelationalExpression:
left=AdditiveExpression
right=RelationalExpressionPart?
;
RelationalExpressionPart:
    op=RelationalOperator
    ex=AdditiveExpression

RelationalOperator:
    |
    >
    |
    >=
    |
    <=

AdditiveExpression:
    left=MultiplicativeExpression
    right+=AdditiveExpressionPart*

AdditiveExpressionPart:
    op=AdditiveOperator
    ex=MultiplicativeExpression

AdditiveOperator:
    +
    -

MultiplicativeExpression:
    left=UnaryExpression
    right=MultiplicativeExpressionPart?

MultiplicativeExpressionPart:
    op=MultiplicativeOperator
    expr=UnaryExpression

MultiplicativeOperator:
    *
    /

UnaryExpression:
    op = UnaryOperator?
    expr=PostfixExpression

UnaryOperator:
    not
    -

PostfixExpression:
    (call=FeatureCall (fCalls+=Call)* access=ExpressionAccess?)
    | (left=PrimaryExpression)

Call:
    . call=FeatureCall
FeatureCall:
    name=Identifier
    '{'
    param=ActualParameterList?
    '})'
    ;

SetOp:
    name=Identifier
    '{'
    decl=Declarator
    declEx=Expression?
    '})'
    ;

Declarator:
    decl+=Declaration (';' decl+=Declaration)* ']'|
    ;

Declaration:
    id+=Identifier (',' id+=Identifier)*
    (':' type=Type)?
    ('=' init=Expression)?
    ;

ActualParameterList:
    param+=Expression
    (','
    param+=Expression
    )*|
    ;

ExpressionAccess:
    '.' name=Identifier
    calls+=Call*
    access=ExpressionAccess?
    ;

PrimaryExpression:
    (lit=Literal
    |    '{' ex=Expression '}
    |    ifEx=IfExpression
    |    'refBy'('refName=Identifier ')
    calls+=Call*
    access=ExpressionAccess?
    ;

CollectionInitializer:
    {CollectionInitializer}
    type=QualifiedName?
    '{'
    init=ExpressionListOrRange?
    '})'
    ;
ExpressionListOrRange:
  list+=ExpressionListEntry
  (',', list+=ExpressionListEntry)*
;
ExpressionListEntry:
  (name=Identifier ('.', attrib=Identifier)? '=='?
   (value=LogicalExpression | collection=CollectionInitializer)
  )?
;
Literal:
  val=Value
;
IfExpression:
  'if'
  ifEx=Expression
  'then'
  thenEx=Expression
  'else'
  elseEx=Expression
  'endif'
;
//------------------------------- Workflow -------------------------------
Workflow:
  'workflow' name=Identifier '{'
  source+=Codesource*
  state+=AbstractStateElements*
  '}'
;
Codesource:
  'source' source=VarQualifiedName';'
;
AbstractStateElements:
  Start | State | End | Choice
;
Start:
  'start' name=Identifier '{'
  nestedelements += AbstractNestedElements*
  '}'
;
State:
  'state' name=Identifier '{'
  nestedelements += AbstractNestedElements*
  '}'
;
End:
  'end' name = Identifier '{'
  services+=Service*
  '}'
;
Choice:
'choice' name = Identifier '{'
  'if' '(' condition = ExpressionStatement ')' '('
    then += AbstractNestedElements+ ')' '
  =>' 'else' '(' else += AbstractNestedElements+ ')' ')'?
'}

AbstractNestedElements:
AlternativeGroup | Optionality | Service | Transition

Service:
'service' service = MethodQualifiedName';'

Transition:
'transition to' state = [AbstractStateElements] ('on' event=MethodQualifiedName)? '
  ;'

Optionality:
'@Optionality' '(' ocondition = ExpressionStatement ')' varelement += AbstractVariabilityElements+
'@/Optionality'

AlternativeGroup:
'@AlternativeGroup' '(' acondition = VarQualifiedName ')' alternative += Alternative+
'@/AlternativeGroup'

Alternative:
'@Alternative' '(' value = Value ')' varelement += AbstractVariabilityElements+

AbstractVariabilityElements:
  Service | Transition

//---------------------------------------------------------- Identifiers --------------------------

Identifier:
  ID | EXPONENT

MethodQualifiedName:
  (ID)(((('.'))(ID)('))))+

VarQualifiedName:
  ID ('.' ID)*

QualifiedNameWithWildCard: QualifiedName('.*')?;
// --------------------- Terminals ----------------------------------

terminal ID: ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')* ;

terminal NUMBER: '
'?
  (('('0'..'9')+ ('.' ('0'..'9')* EXPONENT)?)
  | ('.' ('0'..'9')+ EXPONENT?)
  | ( ('0'..'9')+ EXPONENT)
  ;

terminal EXPONENT:
  ('e'|'E') ('+'|'-'|'')? ('0'..'9')+ ;

terminal STRING :
  ""( '"' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|""|"\""|"\""|"\""|"\""|"\""|"\""|"\""|"\""|"\""|"\""|"\""|"\""|"\""") | !("\""|"\""|")* "" | ""(""(""(""(""(""(""(""(""(""(""(""(""(""")* """
  ;

terminal ML_COMMENT:
  '/*' ->'*/'
  ;

terminal SL_COMMENT:
  '//' !(('
'|'')* (''? '
')?
  ;

terminal WS: (' '|'	'|''|'
')+ ;

terminal ANY_OTHER: ; ;