Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

Validation of Process Diagrams in BORM Method

Jaroslav Bambas

Supervisor: Ing. Robert Pergl, Ph.D.

10th May 2015

Acknowledgements

I would like to thank supervisor of my thesis Ing. Robert Pergl, Ph.D. for his valuable advice and Mgr. Martin Podloucký for his time in consultations. I also thank my family for their support throught the period of study.

Declaration

In Prague on 10th May 2015 . . . .

Citation of this thesis

Bambas, Jaroslav. *Validation of Process Diagrams in BORM Method.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Tato bakalářská práce se zabývá procesními diagramy metody BORM. Hlavním cílem je provedení syntaktické a sémantické analýzy ORD diagramů metody BORM, následné definování pravidel pro validní procesní diagram a navržení algoritmů obstarávající ex-post kontrolu dodržování těchto pravidel. Následná implementace algoritmů je provedena v jazyce Smalltalk do nástroje DynaCASE. Práce je zakončena manažerským vyhodnocením přínosů, které ex-post validace poskytují začínajícím analytikům.

Klíčová slova BORM, ORD, Validace, Procesní diagram, Modelování procesů, DynaCASE, Smalltalk, Pharo

Abstract

This bachelor thesis deals with process diagrams of BORM method. The main goal is to perform syntactic and semantic analysis of ORD diagrams BORM method, then define formal fundamentals for valid process diagram and devise algorithms ensuring ex-post control of compliance of rules. These algorithms are implemented in Smalltalk language to DynaCASE tool. The thesis is concluded by managerial study of benefits that ex-post validation brings to novice analysts.

Keywords BORM, ORD, Validation, Process diagram, Process modelling, DynaCASE, Smalltalk, Pharo

Contents

List of Figures

List of Tables

Introduction

The role of process management and process mapping are increasingly important activities in the middle and big companies. Mapped processes are for example used to optimize production or to design and implementation of information systems. Flawless capture of processes is thus necessary for the further continuation of development.

BORM is one of the methods and notations used for mapping, analyzing and optimizing of processes. To correctly use of this method is necessary to have certain degree of technical knowledge and experience. If the analyst does not have this knowledge and experience, problems often arise in the syntax and semantics of the diagram. This situation does not facilitate the fact that is not specified enough formal fundamentals for correct creating and validation of process diagrams BORM method.

This problem can be reduced by defining clear rules of the diagrams and their automated checking. Implementation of these controls to the editor of diagrams can eliminate many potential faults and give the user feedback about the faults and their location.

Goals of work

The main goal is the formulation of rules for ORD diagrams. On the basis of these rules design algorithms and create a plugin providing validation to DynaCASE tool. Designed algorithms and following implementation is focused on ex-post validations that are not related to process simulation. Ex-post validation runs on completed ORD diagram after user request.

Objective of work is not analyze in detail all techniques and tools of BORM method, deal with validation associated with process simulation and implementation of validation feasible in editor. Specific goals of thesis are:

1. Perform an analysis of BORM ORD diagrams in terms of syntax and semantic

2. Formulate rules for well-formed ORD diagram
3. Design algorithms for ex-post validation and create a plug-in to DynaCASE tool
4. Test implemented validation
5. Perform a managerial study of improvements that brings validation of ORD diagrams

Structure of work

The thesis is structured into 5 parts:

1. Background research analyzing syntactic and semantic features of ORD diagrams BORM method. This part is covered in chapter 1.
2. Defining a well-formed ORD diagram and description of possible bugs based on the analysis in the first part. This part is free continuation of the first part. This is covered in chapter 2.
3. Design and description of validation algorithms. For each algorithm is specified its purpose, asymptotic complexity and text description. It is in chapter 3.
4. Part focused on the implementation of algoriths and testing. Algorithms are implemented in pure object oriented language Smalltalk. This part is covered in chapters 4 and 5.
5. Managerial study of benefits that validation brings to teaching BORM method and novice process analyst. This is covered in chapter 6.

Chapter 1

Introduction to BORM and Analysis ORD

For a better understanding of the following chapters in this work is the first chapter focused on the basic description of BORM (Business Object Relation Modelling) method and detailed analysis of ORD (Object Relation Diagram) diagrams.

1.1 Business Object Relation Modelling

The work on the method BORM started in 1993. The method was primarily invented to provide support for the building of object-oriented software system based on pure object-oriented languages, such as Smalltalk, together with pure object database [1].

Due to the original purpose is BORM designed to cover all phases of software development [5]. In comparison with the other methods, BORM takes development process as a gradual transformation of models from business engineering to software engineering and implementation. BORM starts on the initial specification of a problem and by using a set of rules and techniques converts task to information system solution. The result is the union of business engineering and software engineering [6], two separate disciplines with their own theoretical base and own terminology. Figure 1.1 shows indirect relationship between business and software engineering. Between these sectors is a gap, which BORM covers [7].

Figure 1.2 shows transformation models throughout the lifecycle. For each lifecycle stage, BORM provides a set of terms. Together with the transformations of the model is increased level of detail.

Later use of BORM in practice was found that some of techniques and tools, provided by this metod, are useful as a independent method of business engineering [5]. These are OBA (Object Behavioral Analysis) method, BAD

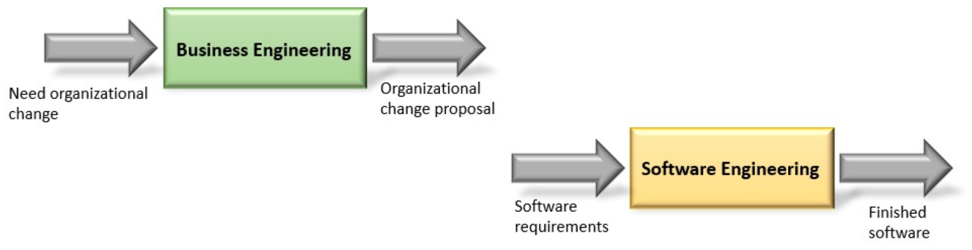


Figure 1.1: Gap between Business and Software Engineering

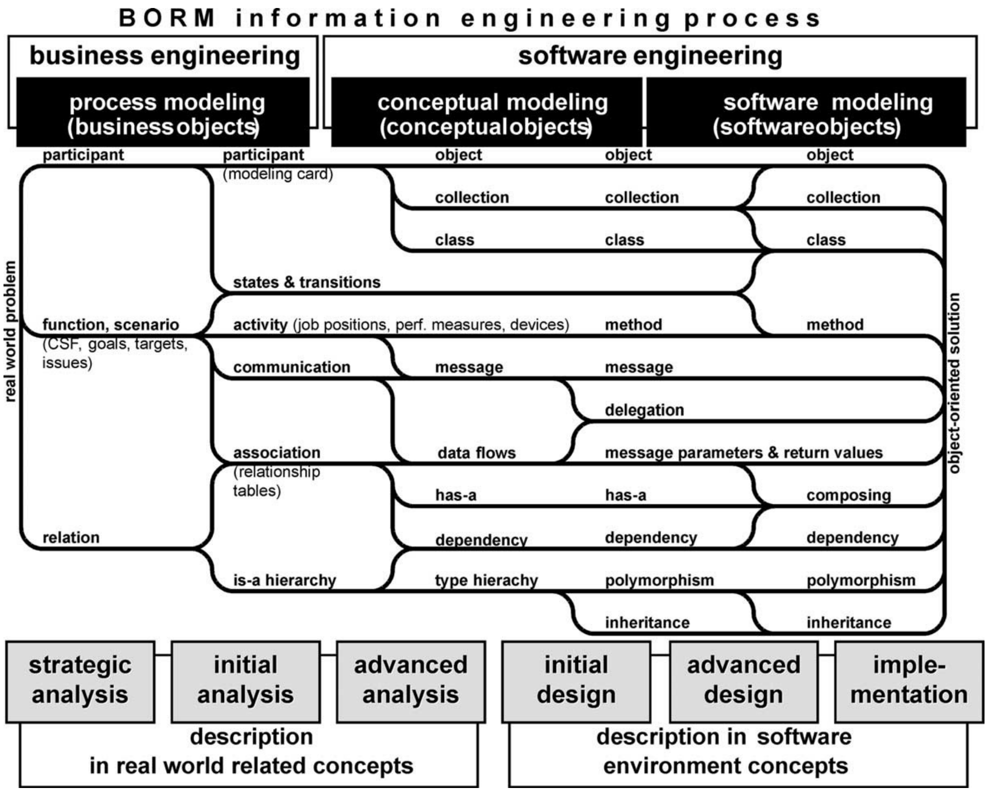

BORM information engineering process
business engineering
software engineering
process modeling (business objects)
conceptual modeling (conceptual objects)
software modeling (software objects)
participant
participant (modeling card)
object
collection
class
states & transitions
function, scenario (CSF, goals, targets, issues)
activity (job positions, perf. measures, devices)
method
communication
message
delegation
association (relationship tables)
data flows
message parameters & return values
has-a
composing
relation
dependency
is-a hierarchy
type hierachy
polymorphism
inheritance
real world problem
object-oriented solution
strategic analysis
initial analysis
advanced analysis
initial design
advanced design
imple-mentation
description in real world related concepts
description in software environment concepts

Figure 1.2: BORM evolution of concepts [1]

(Business Architecture Diagram) and ORD (Object Relation Diagram). OBA method is used to get informations that are needed to create the first object model and BAD is used to represent process architecture, ie. clearly shows the links between processes and their assign to working unit. OBA method and BA diagrams are detailed described in [6] and [5].

1.2 Object Relation Diagram

ORD diagram is a visual representation of process by using connected and oriented graph. Informations about processes and participating objects, which are obtained by OBA, are illustrated in simply and clearly diagram. The diagrams do not use large amounts of symbols and terms. This makes BORM suitable for the following groups of people:

- beginning business analysts who are learning a new notation for business modelling
- people who are not trained in the techniques of software and business engineering, for example, in consultation with customers

1.2.1 Syntax

To understand the syntax ORD is necessary to introduce terms and symbols that present them. Symbols used for terms below are from OpenCASE [8] tool and in other tools can be used slightly different. The description of terms used in BORM ORD is based on [5] and [9]. Terms in ORD are following:

Participant - Participant performs activities in the process. There are 3 types of participants - Person, Technical participant (e.g. information system) and Organizational (e.g. work department). In ORD is participant represented by a rectangle with name and type in the upper part, see figure 1.3.

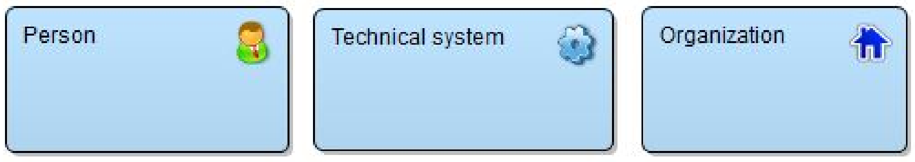


Figure 1.3: Symbols for participants

State - State is the first type of process nodes. It is the point in time, when the participant is waiting to completion of activity and transition to the next state. States indicates gradual changes participants over time. There are these types of states:

- Start state - start sequence of activities of any participant
- Final state - end of activities of participant in the process
- Combination of start and final state

- Inner state

Symbols for all types of states are in figure 1.4.

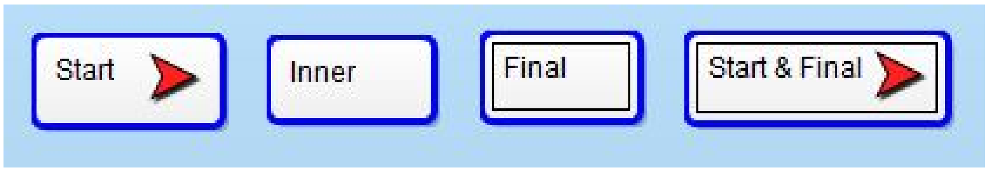


Figure 1.4: Symbols for states

Activity - Activity is the second type of process nodes. It means an operation performed by participant. Activity occurs throughout the transition between two states. Transition to next state is completed when Activity is finished. Symbol of Activity is rectangle with rounded corners or oval, see figure 1.5.

Figure 1.5: Symbol of Activity

Communication and Data flow - The communication represents interdependence between activities of two different participants. Only case where the communication may be between activities of the same participant is service oriented, who does not have states. In other semantic types of participants, there is a problem with time sequence of activities. Symbol of communication is arrow, which connects two activities. Direction of the arrow defines who starts communication and who receives it. Data flow is a term for information, money or another exchanged data between two communicating activities and participants. Symbols for communication and data flow are in figure 1.6.

Transition - The transition shows direction of change from one state to another. The transition is represented by arrow from one state to another. This states are assigned to same participant. In the middle of the arrow is an activity, which is performed during the transition. In figure 1.7 is example of transition between two states.

Conditional transitions and communication - Symbol for conditional transition (communication) is same as for unconditional transition (communication) but in addition there is text description of the condition and

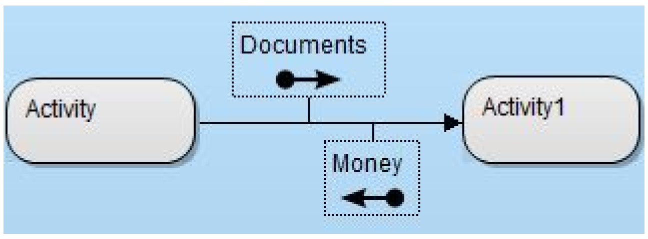


Figure 1.6: Communication and Data flows between two activities

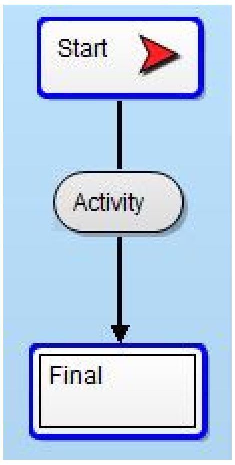


Figure 1.7: Transition from state Start to state Final

the arrow is crossed out. The condition determines when the transition is valid. In figure 1.8 is conditional transition from state *Start* to state *Final* through *Activity*.

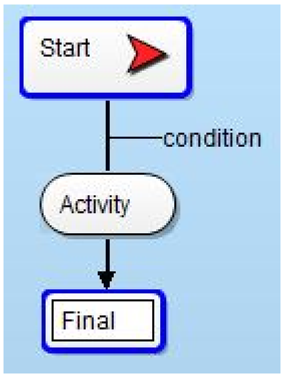


Figure 1.8: Conditional transition

In [10] is described that process is captured to ORD as connection of states, activities and transitions. This connection is constructed by manner *"State-Activity-State"* with transition, see figure 1.7. Into ORD is not possible to add activity, which is not linked to the existing state or is not linked by communication to another activity [10]. All states and activities must belong to one of the participants.

1.2.2 Semantics

Semantic part of ORD diagrams is closely related to the syntax part. Connection between these two sectors is evident in transitions and communications, where opposite drawn direction of communication changes the meaning.

As stated in the article [5], a process can be seen in two ways. First, that each participant in the ORD diagram represents a role, which has own specific behavior and individual activities. Each participant is actually FSM (Final State Machine), which is described by a sequence states and activities. If we look at the paticipants with their states and transitions as FSM, the process is made by merging FSM [5]. The other view is that process is formed by events of participants, ie. activities and states of each participants and communication between the activities of participants. Event flow of process goes across all participants, see in figure 1.9.

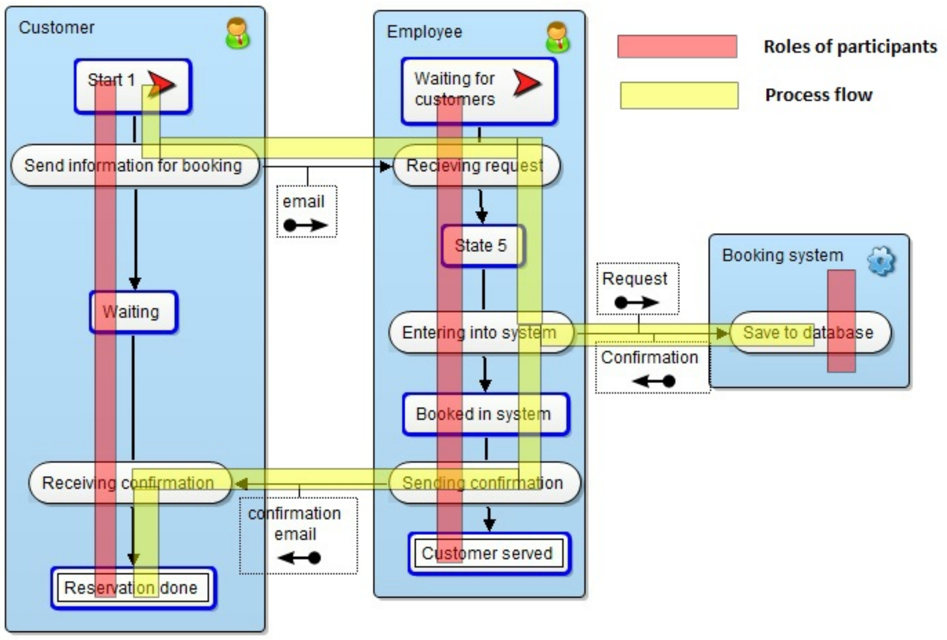


Figure 1.9: Roles of participants and process flow

In creating ORD diagrams are used several basic constructions. Mergers and combinations of these constructions allows to reach of required process diagram. Description of constructions used in ORD:

Sequence - Individual activities sequentially follow on other activity. This flow of activities can be seen in figure 1.9, where participant *Employee* performs one activity after another.

Parallel branches - Parallel branches represents two or more simultaneously performed activities. It is used in cases, if do not matter the order of execution activities and the activities belongs to different branches, which do not depend on each other [11]. At the beginning of the parallel run is split part, which it divides into multiple branches. At the end of the parallel run is join part, where after completion of all branches is continued by one branch, see in figure 1.10. Then it is allowed to continue in next activities as a sequence flow.

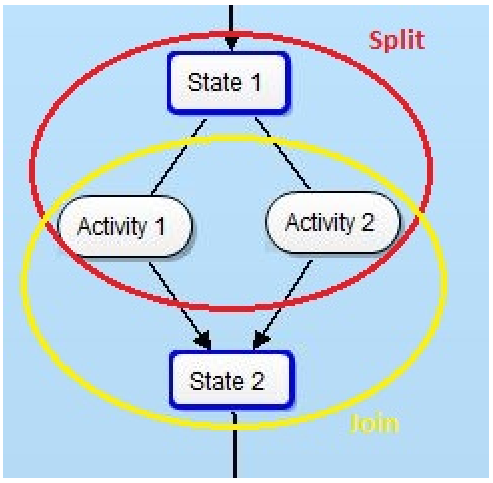


Figure 1.10: Split and join part of parallel branches

Decision making of process flow - Split is performed based on the conditions of conditional transitions. Decision making branches are two or more and process flow continues by all branches, where conditions are evaluated to true. At the end is again a state that provides joining branches. In the figure 1.11 at *Participant 1* you can see decision-making process with 2 branches.

Control flow by foreign decision - Splitting of flow is based on incoming communication from other participant. This type of split is shown in figure 1.11, where *Participant 2* is waiting for incoming communication. According decision of the *Participant 1* is performed communication, which will determine behavior of *Participant 2*.

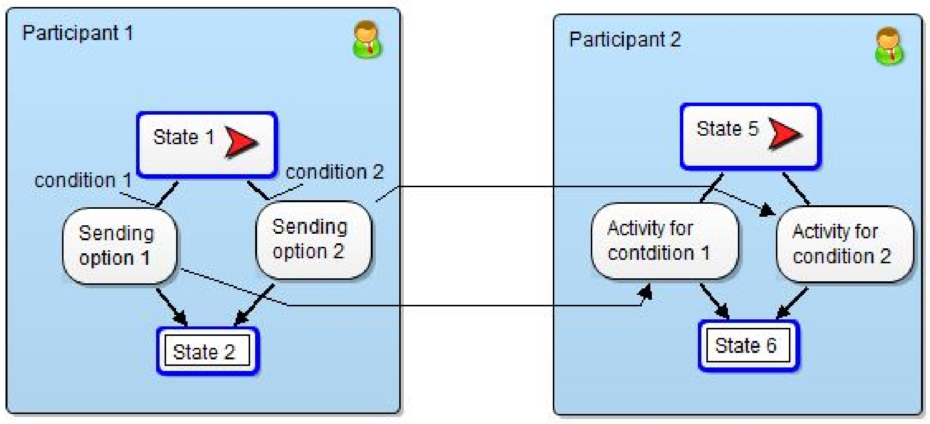


Figure 1.11: Control flow by foreign decision

In terms of semantic, we can distinguish these three types of participant:

Full-fledged participant - It is a participant who includes start and final state. Between these states is a sequence of inner states and activities. In ORD must be at least one participant of this type, otherwise process can not correctly begin and finish.

Service oriented participant - It is a participant who has stateless behavior, ie. for other participants provides services type request-reply. Sample is shown in figure 1.12.

Triggered participant - Process role of participant is not started by start state but incoming communication. Technically, there is a default start state, which is not shown and where the participant waits until it receives the request [12]. This semantic type of participants is completed by final state. Sample of triggered participant is shown in figure 1.12.

1.2.3 Extension BORM ORD

To further machine processing of ORD diagram are word conditions in decision making insufficient because it does not include the boolean logic of possible combinations of outgoing transitions. One of the major problem is a fact that ORD diagram is combination of Mealy automata and Petri nets [7], but these do not cover control of pass [13]. For this reason was defined formal automata - prefix machine [2]. This machine defines the semantics of the ORD diagram to run and simulation. To enable complicated options of parallelism and understand to prefix machine is necessary introduce new syntactic and semantic terms. Articles [11] and [2] describe following terms:

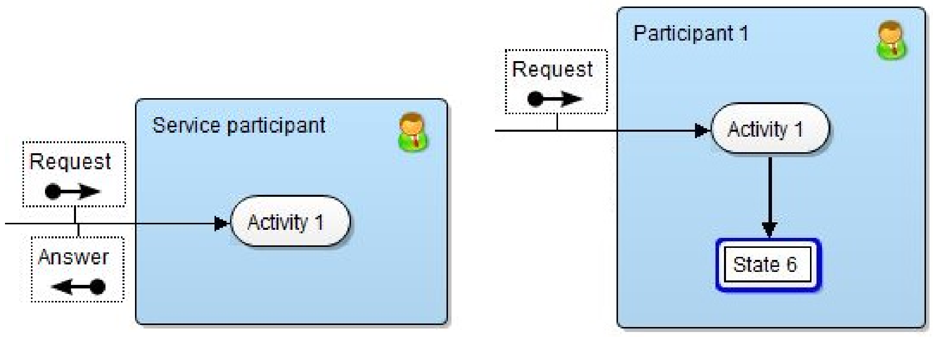


Figure 1.12: Service oriented participant (left), triggered participant (right)

Output condition - Output condition of state is a boolean expression, which specifies admitted combination of branches into which the process execution may split itself from this state [11]. Variables are outgoing transitions from the given state.

Input condition - Input condition of state is a boolean expression, which specifies that the execution of the process cannot advance further from the given state until its input condition is met, ie. until corresponding boolean expression is evaluated as being true [11]. Variables are incoming transitions to the given state.

Simultaneity principle - The simultaneity principle states that no participant can in fact split itself into multiple instances and actually do several tasks in parallel [2].

Dependency principle - The dependency principle says that the task can be completed only after completing tasks on which it is dependent [2]. For expample, to complete the join of three branches in the diagram, it needs completed any two branches. After completion any two branches is the input condition evaluates to true and the process flow can continue.

Using boolean expressions input and output conditions of state, consisting of logical operators (XOR, AND, OR) and their variables, dependency principle can be deploy at parallel branches in ORD. The following check compliance of the dependency principle and the simultaneity principle in simulation can be done by using the prefix machine. Logical operators are assigned in input and output conditions like pairing of parentheses. It means that if at split is set output condition to XOR, in section join is expected in input condition same condition. In figure 1.13 is shown use input/output conditions. The split near the state A is output condition X *xor* Y, which means that process can continue by only one branch of this two. The join is set similar conditions, ie. xor for incoming branches. The join in this example is made of two states into one state without performing any activities. This is possible by using epsilon transitions, ie. participant can change state without making activities. Syntactically is the join of activities in the state also correctly.

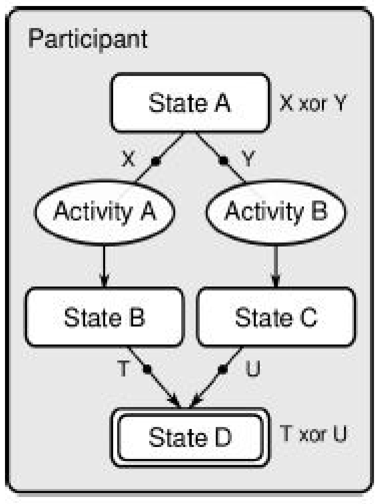


Figure 1.13: Sample of using input and output conditions. [2]

Chapter 2

Requirements for Well-formed ORD

Big disadvantage of ORD is a lack of sound formal foundations which would allow to clearly and precisely define the structure and semantics of ORD and other concepts related to BORM [11]. To effectively validate ORD diagrams, it must be firstly defined the requirements for well-formed diagram, ie. define properties required for valid diagram. This is such diagram that is consistent with all syntactic and semantic properties of ORD diagrams. If the diagram meets the requirements described in this chapter, it is called a well-formed or valid ORD diagram. Drawing up of the requirements for a valid ORD diagram was based on syntactic and semantic priciples of BORM method. Into consideration were also taken results of the works of novice users, who are learning BORM method. Requirements for well-formed ORD are as follows:

1. Each process node (state or activity) belongs to one of participants.

2. Communications are between two activities of different participants. Exceptions are service oriented participants, there may be communication between activities.

3. All communications are associated with data flows.

4. The transition between states is implemented in a manner *"state-activity-state"* or *"state-state"* by using epsilon transition. The transition takes place within one participant.

5. Conditional transitions and communications are labeled by text conditions, in which case is the transition valid.

6. Participant corresponds to one of three semantic types, ie. full-fledged, service oriented or triggered.

7. When passing through the diagram, it is possible to visit all states and activities, ie. all states and activities are connected in process flow.
8. From all branches is reachable any final state by transitions.
9. In the diagram must be at least one start and finale state. It follows that at least one participant must be full-fledged type.
10. Between start and final states must be a path.
11. In case of parallelism and decision making is correctly set input and output conditions in parts split and join.
12. There is no deadlock while waiting for incoming communication.
13. Participants in one diagram communicate with each other. If not, it is possible to divide the diagram into individual parts.

2.1 Validation feasible in the diagram editor

Some of the possible bugs in ORD can be avoided during creation in editor. This is achieved in that during the formation of the diagram is not possible to perform operations that are against syntactical requirements ORD. The fact that an user is not allowed in the editor to create these bugs, it is not necessary to validate ex-post. During editing ORD diagram can be checked:

- New state or activity is assigned any participant.
- Communication is between two activities, not between states or between state and activity.
- Data flows are assigned only to communication, not for transitions.
- Transition takes place within one participant, ie. can not be confused the communication for the transition.
- Transition is routed *"State-Activity-State"* or *"State-State"* and belongs to one participant.

These rules can be checked in the editor because we do not need entire structure of ORD diagram and semantics. We need only information about individual elements in the diagram. For example, when an user wants to add a state, the state can not be inserted into the diagram freely but must be inside one of participants. Implementation of these validations to the editor is recommendation for developers of diagram editor. Next parts of this thesis are focused on the validation ex-post.

2.2 Rules requiring ex-post validation

Rules which can not be checked in creating ORD must be checked ex-post. Validation ex-post must be performed if we need to know the structure of diagram and check the accuracy of semantics. Bugs, caused by violation of the required syntactic and semantic rules of well-formed ORD, are divided into three groups by severity; *Notice, Waring and Error.*

2.2.1 Notice

This type of faults is informative and ORD diagram does not contain serious errors that could have a significant effect on the process. These violations may make it difficult to understand the diagram.

Missing data flow in communication - This is a syntax error when communication is not associated data flow. In the diagram is not described information and data which are exchanged between participants. Presence of data flows at communications can not be guaranteed by diagram editor because at the moment of creation diagram is not known direction and the number of data flows. For this reason there is need to validate ex-post.

Missing conditions for decision making - When deciding must be all transitions to branches labeled with condition when is the transition valid. When all the branches are not labeled with conditions but only some of them, arises a situation where it is not clear that the transition should be used.

Participant without states and activities - Participant without activities and states can be removed from the diagram. This participant is useless in the diagram because it does not have any effect in process flow.

Diagram contains multiple process flows - This error is caused if some of participants do not establish communication to the main process flow, ie. in the diagram is captured more than one process. The solution is divide the diagram into individual processes.

2.2.2 Warning

This is a group of warning errors that indicate possible failure in start the process. This failure is not necessary always, but only in some cases.

Unreachable states or activities - If it is impossible pass by some states and activities, this is due to missing transitions or communications in the diagram. The effect of this error on the process flow is twofold. Firstly, that do not performed all necessary activities, and secondly, that some of activities are redundant in the diagram.

Multiple roles of participant - This occurs if is not fulfilled semantic type of participant. Concretely, if one participant in the diagram presents more roles and can be divided into several participants. Example is shown in figure 2.1. There is the participant with three roles.

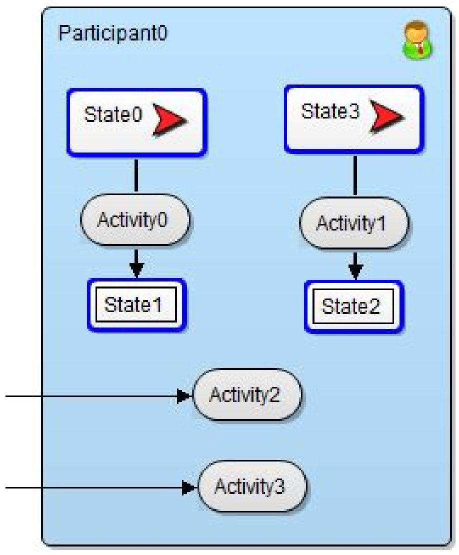


Figure 2.1: Example of Multiple roles, the participant should be divided into three participants.

Possible deadlock in communication - The deadlock occurs if any participant is expected incoming communication but it is not sent. For this reason, the participant can not get to the final state. This semantic error is caused by occurrence at least one of three constructs shown in figure 2.2. They are incoming communication from service oriented participant, sent communication only in some cases of condition and communication that do not respect time sequence. Especially the last one construction is difficult to detect because it can occur in the context of communications between many participants. For this reason it is necessary to detect of this error to perform the simulation of the process, ie. test all existing path through the diagram. Simulation of ORD diagrams is out of scope this work. Solution of incoming communications from service oriented participant would eliminate this problem but not solve completely. For this reason, solution of this problem is the subject of further works on simulation of ORD process diagrams.

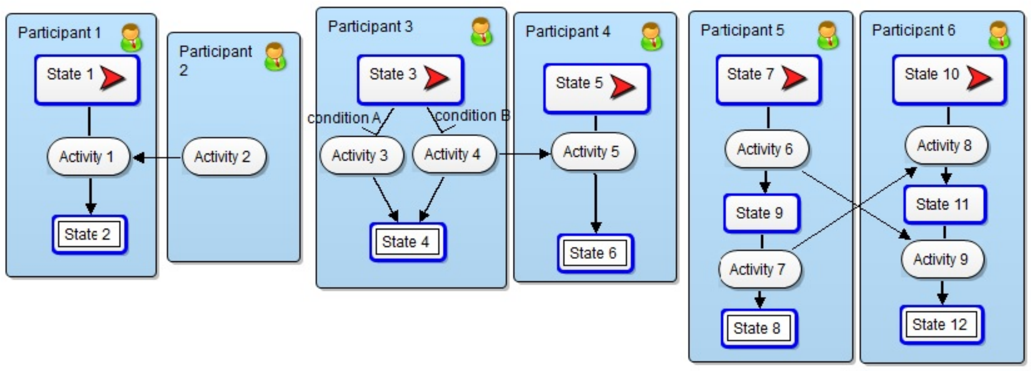


Figure 2.2: Constructions causing deadlocks in ORD

2.2.3 Error

These are fatal errors that do not allow correctly go through the diagram. During simulation process captured by diagram should fail.

Communication between activities of one participant - The communication can be in one participant only in one case, if the participant is service oriented. In other cases, it is a semantic error. The reason is failure to comply with time sequence because referring to activities in the future or in the past.

Missing start or final state - This syntax error occurs in two cases. The first case is that some of the participants have start state and lacks final state or conversely. The second case is that start or final state is missing across the diagram. If in the diagram is missing participant with start and final state, it is not possible to begin or finish a process.

Blind branches - This semantic error occurs if from the process node (state or activity) is no path to any final state by transitions. This is not applicable for service oriented participants who do not include any states. The cause of this error are missing transitions between process nodes or absence of final state of the participant. The result of this error is blind branches in ORD.

Violated dependency principle - To run correctly prefix machine is necessary select equivalent input and output conditions for split and join parts. This is a semantic error in branching, in which do not correspond output condition and input condition. The correct method of choice experssions input and output conditions is similar to issue nested parentheses. Logical operators AND, XOR and OR correspond to three types of parentheses. Example of correct application of input/output conditions is shown in figure 2.3.

Input and output conditions are corresponding to each other if logical equivalence of these condition is a tautology. This means that for all true evaluation of output condition is input condition evaluated true and conversely. For all false evaluation output condition is input condition also false.

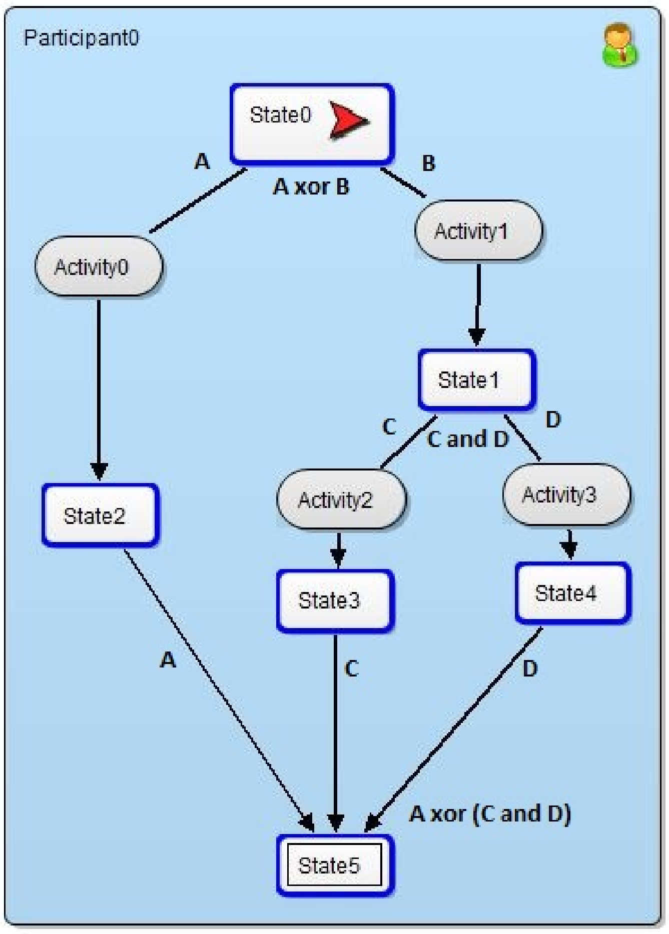


Figure 2.3: Example of correct use of input and output conditions

Chapter 3

Validation algorithms

This next part of the thesis is focused on algorithms providing validation. The design of these validation algorithms with defining the requirements for well-formed diagram is a key part of this work. These algorithms often use various modified forms of algorithms for searching in graphs, like DFS and BFS. The aim of the algorithms is first to get information about whether diagram is valid or not. It is also necessary to identify the specific type of violation of the requirements for well-formed diagram, find out how these bugs are serious *(notice, warning, error)* and where in the diagram the bug arises, ie. find concrete participant, state, activity, transition or communication. By finding a specific type of the fault and its position, this place can be highlighted and give the user more feedback.

Validation algorithms are designed over BORM ORD data model used in the tools OpenCASE [8] and DynaCASE [4]. UML diagram of this model is shown in figure 3.1. The data model corresponds exactly with the graphical representation of ORD diagrams and their syntax and semantics described in section 1.2, ie. ORD diagram includes participants, each participant includes its states and activities between which there are ongoing communications and transitions.

Each algorithm performs checking compliance exactly of one rule. The algorithm always expects input ORD diagram for validation. For each algorithm is defined its purpose, output and text description of the algorithm. For completeness is for each algorithm also intended asymptotic time complexity, although this is not totally critical detail because the expected size of diagrams is in the range of units of participants, the number of states is in the range of several tens. Asymptotic time complexity of the algorithms BFS and DFS was taken from [14]. Diagrams, which are roughly in this range, are not so computationally intensive to it would showed slowing down of editor.

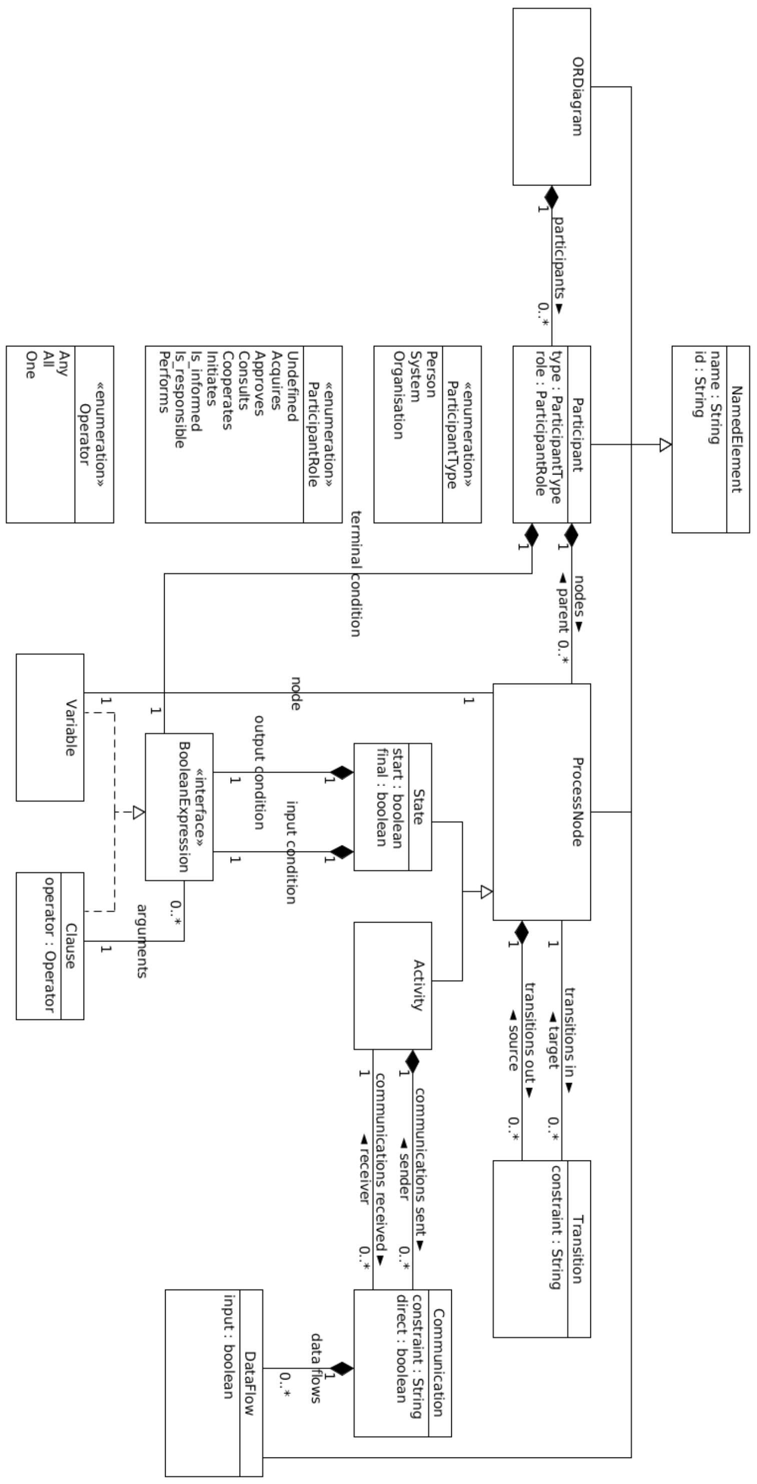


Figure 3.1: Data model of BORM ORD in DynaCASE and OpenCASE tools [3]

3.1 Missing data flow in communication

The goal of the algorithm is to find communications which is associated with no data flow. Due to the design of data model is not necessary go through the ORD diagram by graph algorithms DFS or BFS. To perform is sufficient just work with collections and data structures. The output of the algorithm is a collection containing communications that associate no data flows. If the output collection is empty, the ORD diagram does not contain this type of bug. It is a linear algorithm with asymptotic time complexity $\mathcal{O}(n + c)$, n represents total number of process nodes (sum of states and activities) and c represents total number of communications in validated ORD. The algorithm is as follows:

1. get list of participants in ORD
2. for each participant
   a) browse each process node of the participant
   b) if the process node is the activity, add outgoing communications from this activity to list of communications
3. for each communication from the list of all communications
   a) browse collection of data flows
   b) if there is no data flows, add this communication to results

3.2 Missing conditions for decision making

The goal of the algorithm is to find process nodes (ie. states and activities) whose some outgoing transitions are conditional and others are unconditional. For this validation algorithm is not again necessary to use graph algorithms. This is again a work with the collections from which we select required data. The output of the algorithm is a collection of process nodes with outgoing transitions, where it occurs this error regarding conditional transition. The asymptotic time complexity of this algorithm is $\mathcal{O}(n + t)$, n represents total number of process nodes (sum of activities and states) and t represents total number of transitions in ORD. The algorithm is as follows:

1. get list of participants in ORD
2. for each participant
   a) add process nodes of the participant to collection of process nodes of throughout ORD, ie. get all process nodes in ORD
3. for each process node in the collection

   a) find out number of outgoing transitions, where the condition is empty
   b) if the number of outgoing transitions without condition is different than total number of transitions and the number of outgoing transitions without condition is not equal to zero, add the process node to results

3.3 Participant without states and activities

The goal of the algorithm is to find unnecessary participants in ORD diagram, ie. those who does not include any state or activity. This is a simple algorithm and the output is a collection of these empty participants. The asymptotic time complexity of the algorithm is $\mathcal{O}(n)$, where n represents total number of participants in ORD. The algorithm is as follows:

1. get list of participants in ORD
2. for each participant
   a) get number of process nodes of the participant
   b) if the number of nodes is equal to zero, add the participant to results

3.4 Multiple process flows in diagram

The goal of the algorithm is to find all process flows in ORD diagram. One process flow is represented by connected component of diagram. If in the ORD is more process flows than one, each process flow can be put into a separate diagram. Component of diagram means maximal connected subgraph of the ORD. Nodes in the component are states and activities, edges are represented by transitions and communications and orientation of transitions and communications is ignored. Counting components of ORD diagram is performed using a modified DFS algorithm. The output of the algorithm is a collection of process nodes that have associated component number in which it occurs. The asymptotic time complexity of DFS algorithm is $\mathcal{O}(n+e)$, n represents a number of nodes and e represents a number of edges in diagram. In this case, edges are transitions and communications, thus e is sum of these items. The algorithm is as follows:

1. for each participant
   a) add all process nodes into array and set their flags as FRESH
2. set counter of components to zero

3. for each process node in the array with FRESH flag
   a) set as OPEN, incremente the counter and execute the DFS algorithm
4. return the array with numbered process nodes

The DFS algorithm is as follows:

1. assign to node number of component in which the node is, ie. actual value of the counter
2. FRESH neighboring process nodes connected by transitions and communications push into the stack and set them as OPEN
3. set the process node as CLOSE
4. if the stack is not empty pop next node and go to point 1

3.5 Multiple roles of participant

This algorithm is similar to the algorithm to finding multiple process flows in ORD. In the previous algorithm determines the number of components in the entire ORD diagram, in this case the number of components is determined for each participant and communications between activities are ignored. The goal of this algorithm is to find participants who can be divided into more participants because it contains multiple participant roles. These participants can be distinguished by facts that it is not a service oriented participant and the participant contains many components of graph. One component in a participant is the largest subgraph of process nodes of the participant connected by transitions. The output of the algorithm is a collection of participants that contain multiple roles. The algorithm again uses the DFS algorithm. The asymptotic time complexity of the algorithm is identical to DFS namely $\mathcal{O}(n + t)$, n represents a number of process nodes and t represents a number of transitions. The algorithm is as follows:

1. for each non-service oriented participant
   a) set the counter of participants roles to zero
   b) add all process nodes of the participant to array and set their flags as FRESH
   c) for each process node with flag FRESH
      i. set as OPEN, incrementing counter and execute the DFS algorithm

   d) if the counter of roles is greater than 1, add the participant into results

The DFS algorithm is as follows:

1. FRESH neighboring process nodes connected by transitions push into the stack and set them as OPEN
2. set the process node as CLOSE
3. if the stack is not empty, pop next node and go to point 1

3.6 Unreachable states or activities

The goal of this algorithm is to find process nodes that are unreachable from any initial state. This algorithm uses the BFS algorithm, which is progressively starting from all initial states. Process nodes that are not visited after completing of all BFS are unreachable. The output of the algorithm is a collection of these unreachable process nodes. The asymptotic time complexity of the algorithm is identical to the asymptotic time complexity of BFS algorithm. It is $\mathcal{O}(n+e)$, n represents a number of process nodes in ORD and e is the sum of transitions and communications in ORD. The algorithm is as follows:

1. for each participant in ORD
   a) add all process nodes into array and set their flags as FRESH
2. select from the array with process nodes initials states
3. for each FRESH initial state
   a) set flag as OPEN, execute the BFS algorithm
4. select from the array, which contains all process nodes, nodes with flag FRESH and add them into results

The BFS algorithm is as follows:

1. add to the queue FRESH neighboring nodes connected by outgoing transitions and sent communications
2. set these neighbors as OPEN
3. set the node as CLOSE
4. if the queue is not empty, dequeue next node and go to point 1

3.7 Communication between activities of one participant

The goal of the algorithm is to find communications that are taking place between activities of one participants. If the participant is service oriented, it is not a fault and this design of the communication is correct. The output of the algorithm is a collection of communications that violate this rule. The asymptotic time complexity of the algorithm is $\mathcal{O}(n+c)$, n represents total number of process nodes in ORD and c represents the number of communications in ORD. The algorithm is as follows:

1. get list of participant in ORD
2. for each participant
   a) browse each process node of the participant
   b) if the process node is a activity, add outgoing communications from this activity to list of communications
3. for each communication
   a) if owner of sending activity and owner of receiving activity of communication is equal, test if the owner of the activities contains no states
   b) if it is not service oriented participant, add the communication to results.

3.8 Missing start or final state

The goal of the algorithm is to find cases where in the ORD diagram completely missing start or final states. The second case is missing final state in any of the participants and does not correspond to the semantic type of participants. It is findind information on the occurence of these states. The output of this algorithm does not say anything about whether these states are included in the only process flow or these states are reachable. The output is a collection of participants that are missing these states. In the case that these states are not in whole ORD, collection includes all participants of ORD diagram. The asymptotic time complexity of the algorithm is $\mathcal{O}(n)$, n represents a number of process nodes in the validated algorithm. The algorithm is as follows:

1. set counters of start states and final states in ORD to zero
2. for each participant in ORD

   a) check if the participant is service oriented, if it is, continue by next participant
   b) count start states in the participant
   c) count final states in the participant
   d) if the number of final states is equal to zero, add the participant to results
   e) add to counter of start states in ORD the number of start states in the participant
   f) add to counter of final states in ORD the number of final states in the participant

3. if the total number of start or final states is equals to zero, remove all from results and add to results all participants

3.9 Blind branches

The goal of the algorithm is to find process nodes that are part of a blind branch. Blind branches mean sequence of states and activities connected by transitions, from which is no path in the direction of transitions to any final state. This algorithm is similar to the algorithm for finding unreachable states and activities. It uses a opposite orientation of transitions in the diagram and BFS algorithm is gradually started on all final states. Process nodes tagged as FRESH after completing all BFS algoritms are unreachable from final states. It follows that in the original ORD is no oriented path by transitions between this process nodes and any final state. The output of the algorithm is a collection of process nodes, which form blind branches in the ORD diagram. The output does not cover activities of service oriented participants because there is no states and transitions. The asymptotic time complexity of the algorithm is identical to BFS namely $\mathcal{O}(n + t)$, n represents a number of process nodes in ORD and t represents a number of transitions in ORD. The algorithm is as follows:

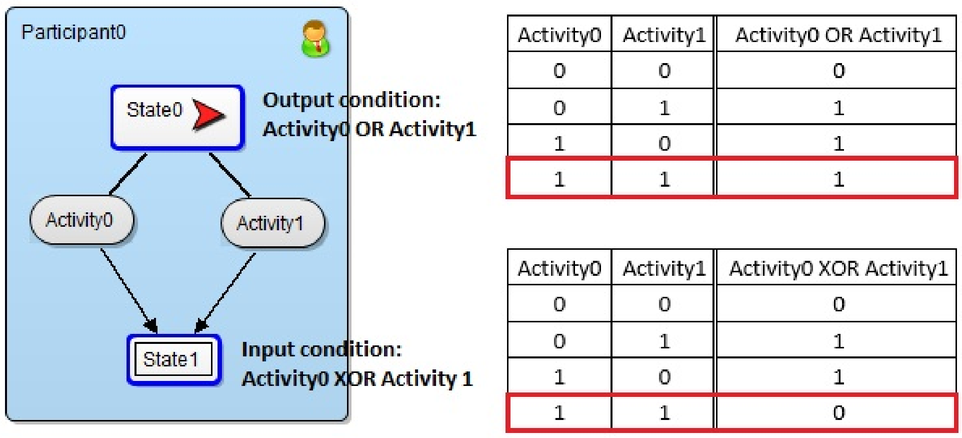
1. for each non-service oriented participant in ORD
   a) add all process nodes into array and set their flags as FRESH
2. select from the array with process nodes final states
3. for each FRESH final state
   a) set flag as OPEN, execute the BFS algorithm
4. select from the array, which contains all process nodes, nodes with flag FRESH and add them into results

The BFS algorithm is as follows:

1. add to the queue FRESH neighboring nodes connected by incoming transitions
2. set these neighbors as OPEN
3. set the node as CLOSE
4. if the queue is not empty, dequeue next node and go to point 1

3.10 Violated dependency principle

The goal of this algorithm is to check the truth tables of input and output conditions. Dependency princip is violated in two cases. The first case is if combination of outgoing transitions, for which is the output condition evaluates to true, the input condition is evaluates to false, example in figure 3.2. Output condition in the part split allows branching, which are evaluated to false by input condition in the join part.



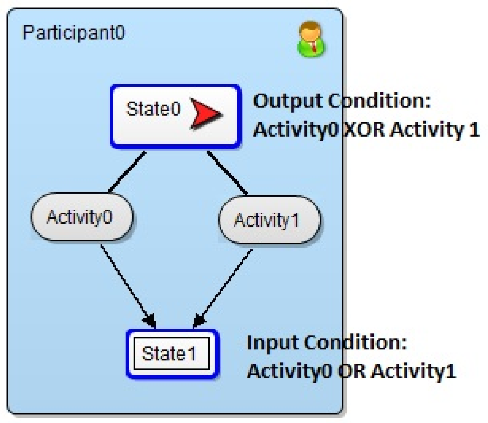
Activity0	Activity1	Activity0 OR Activity1
0	0	0
0	1	1
1	0	1
1	1	1

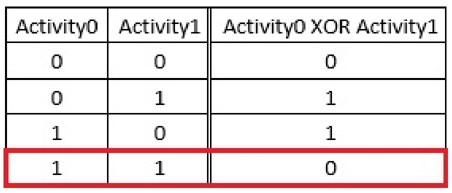
Activity0	Activity1	Activity0 XOR Activity1
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3.2: Input condition does not cover all true combinations of Output condition

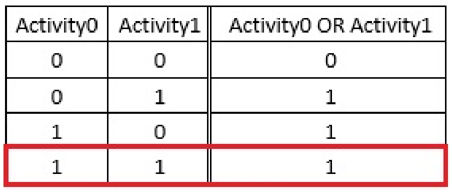
The second case is conversely. Input condition is evaluated to true for the combination of incoming transitions that can not be sent by the states with output condition because for this combination of outgoing transitions is output condition evaluated to false, example in figure 3.3. The output of this algorithm is a collection of states for which there is a conflict in the truth tables of boolean expressions of input and output conditions.

The algorithm is based on the algorithm to derivation of output conditions from the input conditions. This algorithm is described in [13] and [2].





Activity0	Activity1	Activity0 XOR Activity1
0	0	0
0	1	1
1	0	1
1	1	0



Activity0	Activity1	Activity0 OR Activity1
0	0	0
0	1	1
1	0	1
1	1	1

Figure 3.3: Input condition cover combination that are evaluated by condition to false

This algorithm is exponential and is based on BFS algorithm with backtracking. Due to backtracking is asymptotic time complexity exponential, namely $\mathcal{O}((n-1).(2^d)^n+2^d n)$ [13], n represents total number of process nodes in ORD and d represents maximal number of outgoing transition from one state. First part $(n-1).(2^d)^n$ represents asymptotic complexity of generating all possible configuration from states with input condition. The second part $2^d n$ represents asymptotic complexity of comparing truth tables of output conditions with generated configurations.

Sample creating configurations from node with input conditions is shown in figure 3.4. From state with input condition are constructed all possible path through ORD. Visited process nodes during the execution the process are marked with a number 1 (true), unvisited process nodes are marked with a number 0 (false). The process of deriving output conditions from input conditions is as follows. For each state with input conditions is created initial configuration. In the initial configuration is the state with input condition marked by number one (true) and enqueue for BFS algorithm. For each configuration is performed modified BFS algoritm.

During the BFS algorithm may arise for dequeued process node 3 situations.

1. process node is associated with input condition
   a) process node is marked by number one (true) - in the truth table select rows with true evaluation of input condition; for each true evaluation create new configuration which is a deep copy; neighboring process nodes (senders of incoming transitions) mark by values of selected rows in the truth table; neighboring process nodes enqueue and continue by BFS algorithm in new configurations

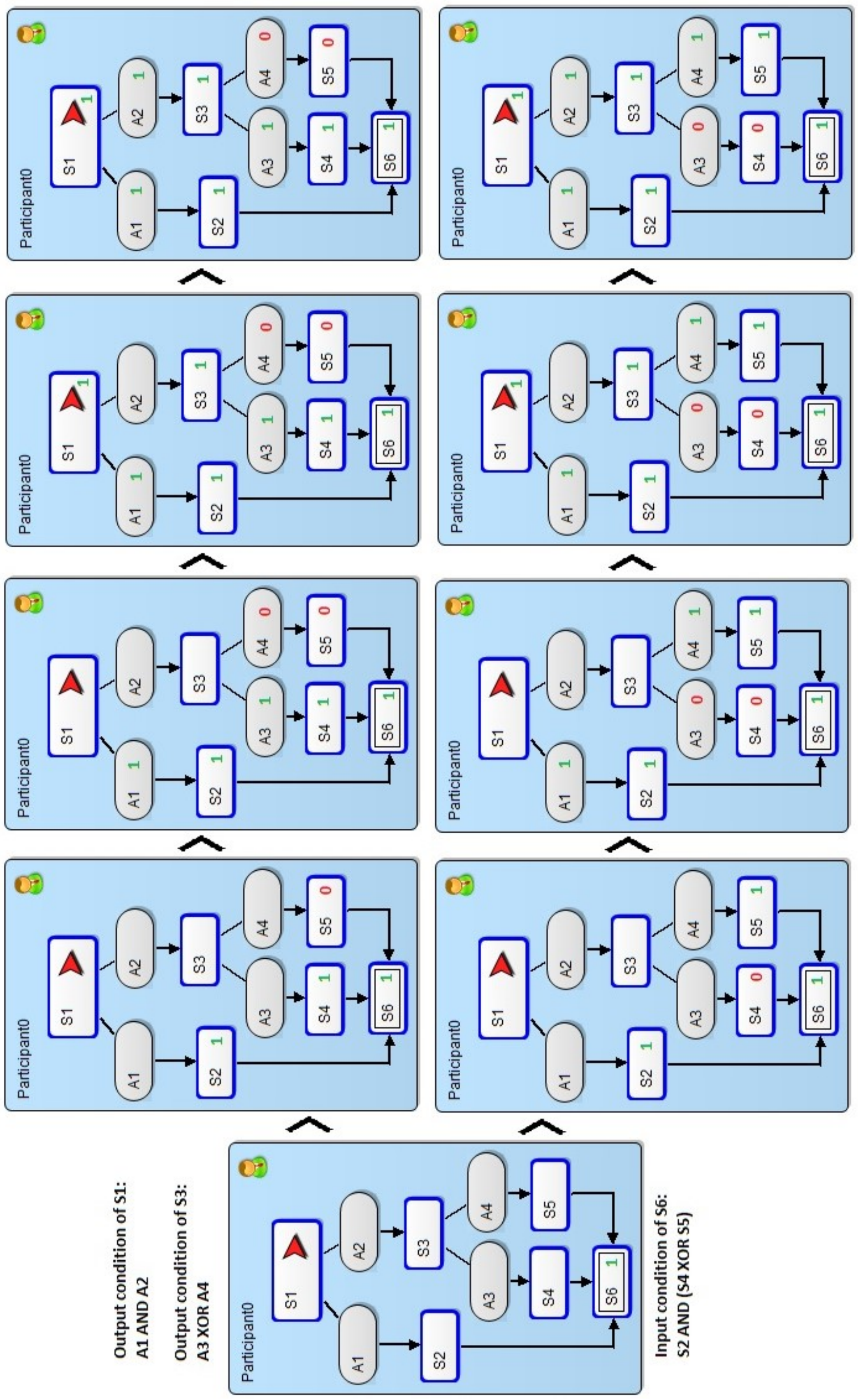

Output condition of S1: A1 AND A2
Output condition of S3: A3 XOR A4
Input condition of S6: S2 AND (S4 XOR S5)
Participant0
S1
A1
A2
S2
S3
A3
A4
S4
S5
S6

Figure 3.4: Creating configurations from nodes with input conditions

   b) process node is marked by number zero (false) - in the truth table select rows with false evaluation of input condition; for each false evaluation create new configuration which is a deep copy; neighboring process nodes (senders of incoming transitions) mark by values of selected rows in the truth table; neighboring process nodes enqueue and continue by BFS algorithm in new configurations

2. process node is associated with output condition
   a) all predecessors are marked (receivers of outgoing transitions) - mark the process node by evaluation from truth table of output condition, mark identical neighboring process nodes (senders of incoming transitions) and enqueue them
   b) all predecessors are not marked - continue BFS another node from queue

3. process node is not associated with any condition - neighboring process nodes (senders of incoming transitions) mark identically as the dequeued node, enqueue them and continue in BFS

The next step is for each state with output condition compare derived output conditions with defined conditions. In each configuration is found the state with output condition and in its truth table is marked combination of outgoing transitions, which is in the configuration. After marking all configurations is performed check if all combinations, which are evaluated to true by output condition, are included in configurations. If in the truth table are not marked all true evaluation, the state is added to results. For all states in diagram must be specified output conditions equivalent to its deriving output conditions, otherwise the diagram is not valid.

For clarity is not description of algorithm very detailed but it focuses only on the base points, which are described above. The validation algorithm is as follows:

1. get collection of states with input condition
2. for each state with input condition
   a) create and fill in truth table for input condition
3. get collection of states with output condition
4. for each state with output condition
   a) create and fill in truth table for output condition
5. for each state with input condition
   a) create possible configurations

6. for each state with output condition
    a) perform marking of truth table according to configurations
    b) perform check of truth table; if the check fails, add the state to results

Chapter 4

Implementation

To use validation of ORD diagrams in practice, it is necessary perform the implementation of designed algorithms in chapter 3 into a diagram editor that supports BORM notation. For this purpose was selected newly developed CASE tool for creating diagrams and models called DynaCASE. This tool is being developed within subjects BI-SP1 and BI-SP2.

4.1 Technology and DynaCASE

DynaCASE tool is based on Pharo project. Pharo is modern open-source development environment for the Smalltalk programming language [15]. For DynaCASE has Pharo dual function, firstly function of framework and secondly function of development environment. DynaCASE is attemping to follow in some respect MVC architecture [4].

Base data model of DynaCASE is shown in figure 4.1. Using this model can be DynaCASE extended by other data models of other modelling notations. Expanding functionalities is possible by adding packages. Currently in the tool are included packages for creating diagrams of FSM, OntoUML and BORM. For the purpose of validation algorithms is needed only part of the model, specifically data model of ORD diagrams shown in the figure 3.1.

To implement is used programming language Smalltalk. It is pure object oriented language. It means that everything, with which works in the language, is an object. This language allows to solve elegantly some of validations without complicated browsing whole diagram. This is made possible through easy work with collections and designed BORM data model. Another advantage of Smalltalk is its clearly and simple syntax.

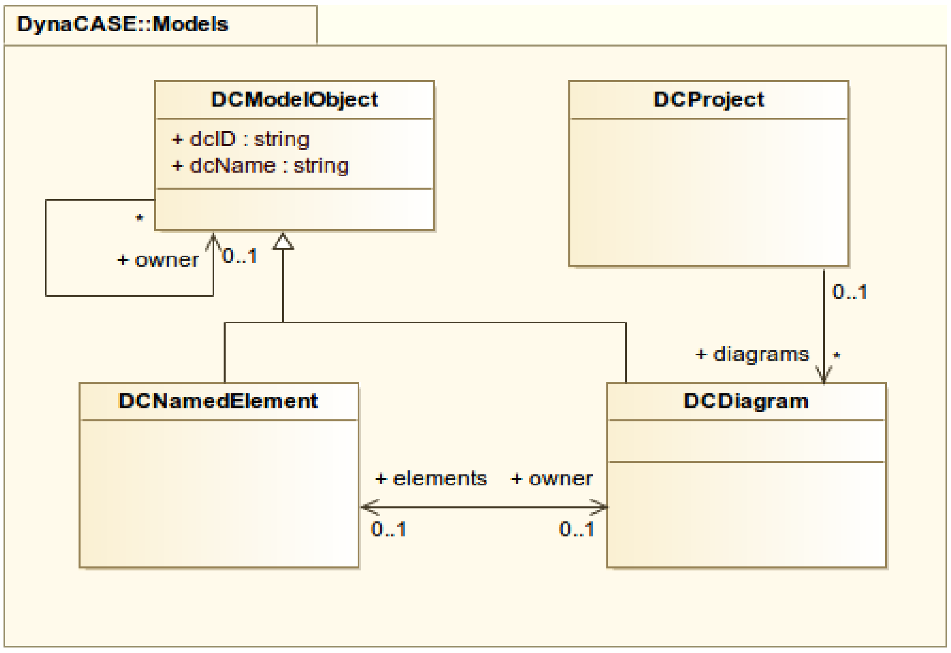


Figure 4.1: Data model of DynaCASE [4]

4.2 Validation plug-in

Validations are implemented as a extension package, which needs for its function package of BORM data model. This package include these 4 groups of classes:

1. Class BormValidator
2. Classes represents errors of ORD
3. Classes of tests
4. Auxiliary classes for BFS and DFS algorithms

Classes of tests are described in the chapter 5. The following sections are described the first two items.

4.2.1 BormValidator

BormValidator is the class that provides the process of the validation. This class includes two instance variables, namely *aORD*, this is validated ORD diagram, and *listOfFailures*, which is a collection that contains instances of error classes described in section 4.2.2.

The class also includes 10 methods, which are implemented validation algorithms described in the chapter 3. Samples of source codes of any validation algorithms are given in appendix B. If any validation algorithm finds an error, it is created instance of the error type and is added into collection *listOfFailures.*

Due to work only on the package of validations was not possible to implement auxiliary methods in the package of BORM data model and they are inadequately implemented in the BormValidator class. To maintain the purity of the object code and design, it is recommended in the future to move these auxiliary methods to package of BORM data model. These include methods to obtain all process nodes of ORD, all communications of ORD, etc.

4.2.2 Error classes

The package with validation also include classes that represent faults against requirements on well-formed ORD. These faults are described in section 2.2. These classes are the output of whole process of validation. These classes are used to output of entire process of validation and each class contains information about its severity (notice, warning, error) and collection of items, which are worth of faults. These classes are subclasses of existing *Error* class in environment Pharo. Error classes are following:

- CommunicationInOneParticipant
- EmptyParticipants
- MissingConditions
- MissingDataFlows
- MissingStartOrFinalState
- MultipleProcessFlow
- MultipleRolesOfParticipant
- NodesOfBlindBranches
- NotCorrectIOConditions
- UnreachableNodes

In the future, after implementation of validation deadlocks, there will be added one more class indicating the presence of this error.

Chapter 5

Tests of validation algorithms

Testing is inseparable part of development and extending the functionality of software. Tests of validation algorithms is divided into 2 parts. At first are performed unit tests to test correctness of output each validation algorithm. Unit tests are performed in environment Pharo by using class TestCase. The second group of tests is performed with complex diagrams.

5.1 Unit tests

In total are performed 34 unit tests. Each test is focused on one validation algorithm and its outputs. The size of tested diagrams is about 5 process nodes and in each ORD is only error related to testing algorithm. Unit tests are divided into different classes, one class includes tests for just one validation algorithm. All unit tests were done correctly. Classes of tests are as follows:

MissingDataFlowTest - The class that includes tests of the algorithm to finding missing data flows. The algorithm is described in section 3.1. There are 3 unit tests of the algorithm, the first is test of valid ORD, the second is test of invalid ORD with missing data flow and the last is test of invalid ORD that is corrected during the test to valid.

MissingConditionsTest - The class that includes 3 tests of the algorithm to finding missing conditions in outgoing transitions. The algorithm is described in section 3.2. Two tests are performed on valid ORD and third testing ORD contains combination of conditional and unconditional transitions.

EmptyParticipantTest - The class of tests of the algorithm for finding participants without process nodes, described in section 3.3. This class includes 3 unit tests. The first test is performed on the diagram that contains the empty participant. The second test is performed on the diagram where this participant does not occur and the last is the diagram when the empty participant is not included at the beginning and then is added. After that the test is repeated.

MissingStartOrFinalStateTest - The class of tests of the algorithm to finding missing start or final states. The class includes 4 tests of the algorithm described insection 3.8. The first tests are performed on diagrams with missing final state, missing start state, missing both states and valid diagram.

MultipleProcessFlowTest - The class of tests of the algorithm described in section 3.4. This class includes 3 tests on ORD diagrams with 1 process flow, 2 process flows and on empty diagram with no participants.

MultipleRolesTest - The class of tests of the algorithm described in section 3.5. The class includes total 3 tests. The first is a test on a valid diagram, the second is a test containing the participant with two roles and finally the last one, test on the diagram containing participant with two initial states.

UnreachableNodesTest - It is class which includes tests of algorithms for finding reachable process nodes described insection 3.6. There are 3 total test. One of these tests is test on the valid diagram, the second is test on the diagram where is the activity unreachable and the last test on the diagram with unreachable state.

CommunicationInOneParticipantTest - The class including tests of algorithms to finding communication processing within one non-service oriented participant. The algorithm is described in section 3.7. There are total 3 tests. The first test is on the valid diagram, the second one is on the diagram containing this error and the last one is the test on the diagram where is the communication processing within one service oriented participant.

BlindBranchTest - The class includes tests of algorithm to finding process nodes that are components of any blind branch. This algorithm is described in section 3.9. The class contains 2 tests, the first on valid diagram and the second on diagram containing one blind branch from which is not able to get to the final state.

ViolateDependencyTest - The class includes tests of algorithm that testing correctness of input and output conditions. This algorithm is described in section 3.10. The class includes total 7 tests, including tests of nesting conditions.

5.2 Test on complex diagram

Tests on complex diagrams are made on ORD diagrams in size about 30 process nodes and about 5 participants. This size is equivalent to the average modeled process diagrams. ORD diagrams used for these tests are given in appendix C. For each ORD used in this tests are started all validation algorithms and controls output of all errors. For this test was made 3 diagrams. The first diagram is valid and contains the correct use of all elements of BORM ORD. In the second diagram are all types of errors that can be detected by these validation algorithms and last diagram is selected from semestral projects of subject BI-ZPI. These semestral projects are described in chapter 6. All tests on complex diagrams were done correctly. Time of running validation on complex diagrams is less than one second. Ex-post validation of ORD do not limit the user during his work.

Chapter 6

Managerial study the benefits of validation ORD

The final part of this thesis is focused to evaluation of benefits that brings validation ORD diagrams for novice users and analysts. Specifically, the aim is to determine whether validation ORD diagrams will help reduce the number of errors and improve their quality. The second goal is to verify if the requirements for diagram and method validation ex-post, as described herein, are corresponding to real problems that occur in the diagrams of novice users.

6.1 SWOT analysis

First is performed a brief SWOT analysis of BORM method, namely Objective Relative Diagrams as tool of process managemet. This analysis captures the state of original ORD without extensions described in section 1.2.3. The SWOT analysis is way to identify these items:

- ***S**trengths:* internal properties that represent advantage over other methods used in process management
- ***W**eaknesses:* internal properties that represent disadvantage relative to others methods used in process management
- ***O**pportunities:* external elements that could be exploited to its advantage
- ***T**hreats:* external elements that could have negative impact on the further development and use of BORM method

One of the opportunities in the analysis in the figure 6.1 is the realization of validation ORD. As part of this work has been used this opportunity and was designed and implemented validations into real software tool DynaCASE.

Focus of further sections of this chapter is whether the realization of this opportunity may have a positive effect on the results of novice analysts.

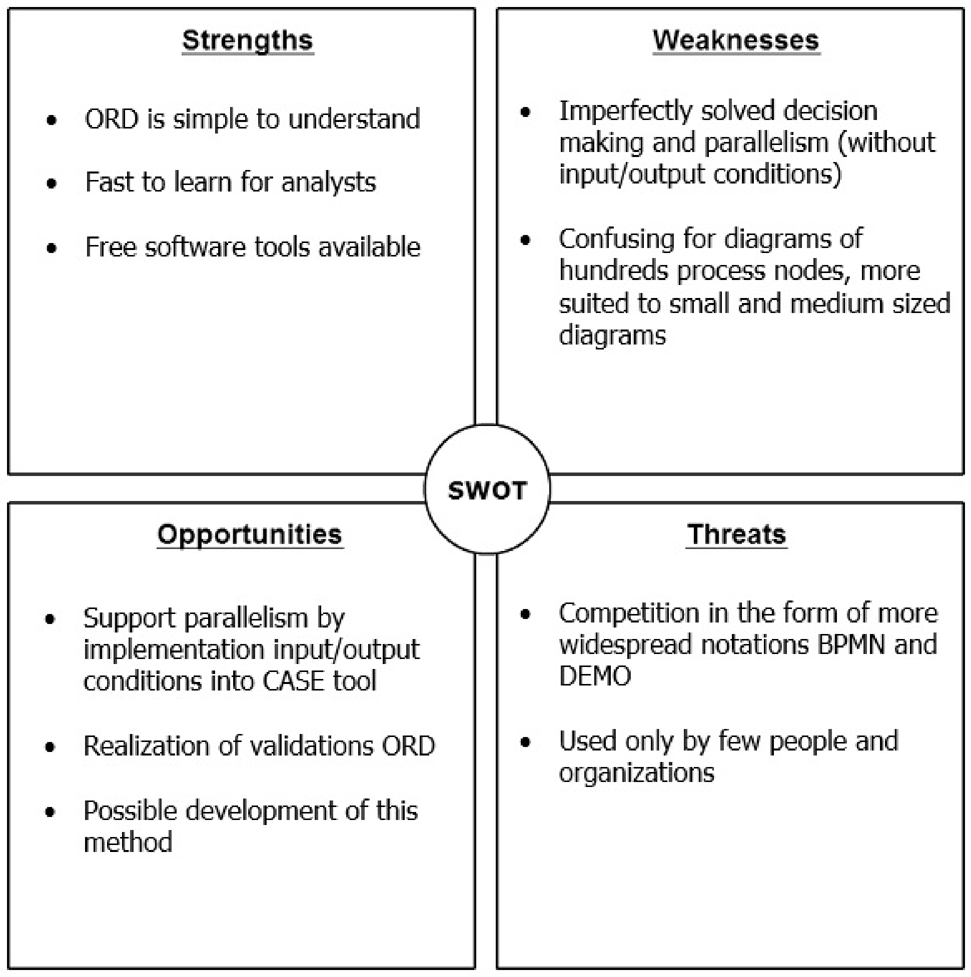

Strengths
ORD is simple to understand
Fast to learn for analysts
Free software tools available
Weaknesses
Imperfectly solved decision making and parallelism (without input/output conditions)
Confusing for diagrams of hundreds process nodes, more suited to small and medium sized diagrams
SWOT
Opportunities
Support parallelism by implementation input/output conditions into CASE tool
Realization of validations ORD
Possible development of this method
Threats
Competition in the form of more widespread notations BPMN and DEMO
Used only by few people and organizations

Figure 6.1: The SWOT analysis of BORM ORD

6.2 Methodology

Data for this part of the work has been semestral works in BI-ZPI. This subject is taught by Ing. Robert Pergl, Ph.D. on the Faculty of Information Technology CTU. The goals of this course are introduce students to the basics of process modeling and the correct use of notation applied in this field of study. This is precisely the target group of users who should validation ORD most help. Validation provides them feedback about the type of fault and where it occurs. The user can then respond and fix it.

There were semestral works from years 2013 and 2014. To survey of bugs was 13 semestral works containing a total 144 ORD process diagrams in BORM method. These diagrams are created in editor OpenCASE [8], which already contains checks carried out during the creation of the diagram. This editor but contains no ex-post controls of validity. To find information about the occurrence of errors was therefore necessary manually check each diagram. List of controlled bugs in ORD diagrams is consistent with the well-formed diagram and the errors described in section 2.2. The list is following:

- Missing data flow in communication
- Missing conditions for decision making
- Participant without states and activities
- Diagram contains multiple process flows
- Unreachable states or activities
- Multiple roles of participant
- Possible deadlock in communication
- Missing start or final state in ORD
- Participant contains start state but no final (or conversely)
- Communication between activities of one participant
- Blind branches
- Exist unfinished branches

Violations against the input/output conditions and dependency principle is not included in this list because in the moment of making these semestral works this concept was not designed and implemented in the diagram editor.

6.3 Results

During the inspection of diagrams were recorded several pieces of information about the numbers of bugs in ORD diagrams. At first whether it is valid diagram or diagram contains some errors which violate the definition of well-formed diagram. Further were recorded the severity of errors, ie. according to the division on notice, warning and error. In conclusion was in detail recorded frequency bugs according to particular species. Results of the first two measurements are shown in the graphs in figures 6.2 and 6.3.

As you can see in 6.2, most of the ORD diagrams in semestral works, namely 58 %, contains a bug, which can be found using the ex-post validation and 42 % of ORD diagrams are valid. In the second graph in figure 6.3 are the results of comparing frequency of bugs according to severity, ie. whether it was a bug of severity notice, warning or error. If the bug is more serious, it can cause greater problems for future work with ORD diagram and process, for example simulation, generation of reports, etc. Percentage representation is 31 % for notice bugs, 20 % for warning bugs and 49 % for error bugs.

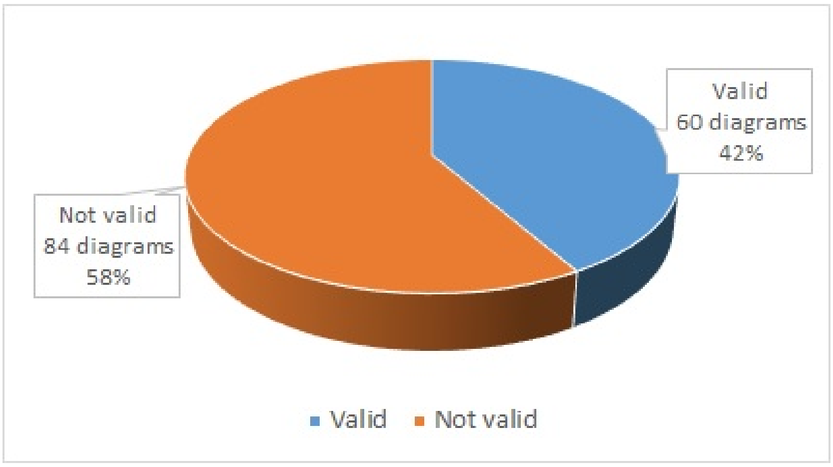


Figure 6.2: Ratio of valid and invalid ORD

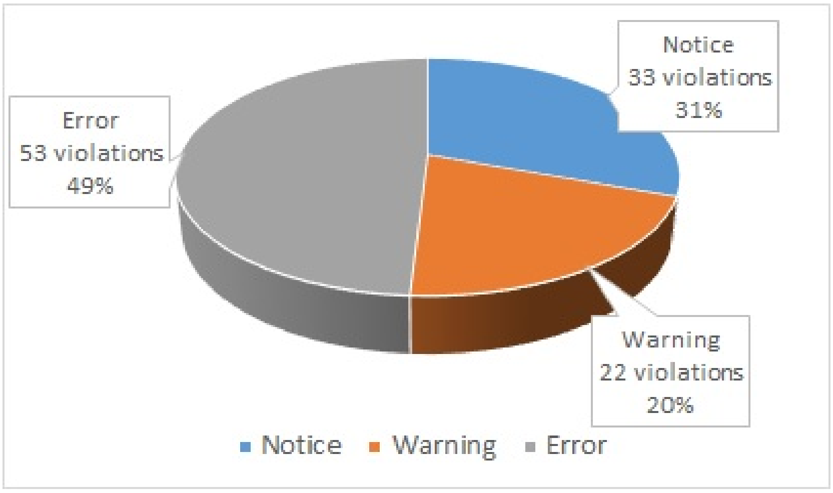


Figure 6.3: Distribution of bugs according to severity

Table 6.1 shows the frequency of various types of violations against well-formed diagram for each ORD and for each semestral work. The list of violations is identical to the list set out in part Methodology. Many ORD diagrams contained the several bugs of same types. All these errors are recorded in table

6.1 as one occurence of this species. For example, in the diagram is missing three times data flow in communication. This case is recorded as one occurrence of missing data flow in communication in diagram and one occurrence of this bug in the semestral work. If this bug occurred already in another diagram of the same work, in error rate of the semestral work is recorded only once. In case of several types of bugs in one diagram are equally recorded all types of errors.

Many types of these bugs (especially missing data flows, missing transitions, unreachable states or activities) do not arise due to ignorance of basics terms in BORM but due to inattention and lack of feedback, which would notify you. Other types of bugs (multiple roles of participant, participant without final state) are a sign of unawareness of syntactic and semantic elements of BORM method.

Type of error	**Number of violations in ORD**	**Number of violations in semestral work**
Missing data flow in communication	22	8
Missing conditions for decision making	10	5
Participant without states and activities	0	0
Diagram contains multiple process flows	1	1
Unreachable states or activities	3	2
Multiple roles of participant	16	7
Possible deadlock in communication	3	3
Missing start or final state in ORD	8	5
Participant with start state and no final	25	9
Participant with final state and no start	5	3
Communication between activities of one participant	1	1
Blind branches	6	4
Exist unfinished branches	8	6

Table 6.1: Summary of representation of errors and quantity

In conclusion must be said to results that in the cases of less complex diagrams was presence of errors exceptionally. In the case of diagrams of complex processes containing parallelism, communication between several participants and a large number of states and activities, the number of bugs greatly increased.

6.4 Summary

From the results it is seen that bugs, which can be found using ex-post validation, are in semestral works many and most of them are a consequence of lack of feedback. This can be deduced from the fact that some type of bugs occurs only with one diagram in whole semestral work. If the type of bugs is repeated periodically in almost every ORD, it indicates incomprehension or lack of knowledge of some syntactic or semantic features BORM ORD.

Bugs with the highest severity are in almost half of semestral works. Because of these frequently fatal errors against well-formed diagram seems validation ORD as a very important due to other possible work with process and the diagram (e.g. transformation to other models in the life cycle of BORM, process simulation and generation reports). Due to the occurrence of deadlock in semestral works that are not solved in this thesis, implemented validation does not cover all types of errors that were in these works. Percentages of this error is less than 3 % of all bugs. In spite of missing validation of deadlocks it seems proposed that method of validation as benefical.

As is seen from the results, ex-post validation is performing control of elements in ORD, where they produce frequently bugs. These bugs are really found in creative work of novice analysts, who are learning the syntax of BORM. Into teaching method BORM can validation bring easier control the correctness of created diagrams. Another and perhaps more important benefit is feedback for users and the possibility of verify that the diagram is correct and valid. This point implies a reduction rate of bugs in semestral projects.

Conclusion

The goals of this thesis were successfully fulfilled by the following way. Formulation of rules for well-formed ORD diagram was made based on the background research aimed to analyze syntactic and semantic features ORD diagrams. Then were detected possible violations of these rules. These faults were divided into 2 groups. The first group consists of rules that can be implemented to the diagram editor. The second group consists of rules that must be validated ex-post. Further sections of the thesis deal with ex-post validation. These faults were subdivided by severity into 3 groups (notice, warning, error) and were described specific situations of their occurence.

For each fault detectable by ex-post validations was designed way to detect this fault and get problematic element of ORD diagram. For DynaCASE tool was developed validator plug-in, in which are implemented designed algorithms. Implementation was performed in pure object oriented programming language Smalltalk.

In part of tests of validation algorithms was found no problem and all tests were done according to requirements and expectations. Testing was carried out in unit tests on simply diagrams and on complex diagrams with size that correspond to diagrams created by students in their semestral projects. In tests on complex diagrams was found that the validation of ORD diagrams, in range many tens process nodes, are not so computationally intensive to limit work with DynaCASE.

The thesis is concluded by managerial study of improvements that brings validation. This study aims to find out whether designed and implemented validations are focused on weaknesses that are actually present in diagrams of novice users. The intention of the study is also find out whether is realistic eliminate these faults in future semestral projects.

Future work

Due to missing validation of possible deadlocks is needed in future work to implement process simulation. This would cover all detected semantic problems. This part could be realized in eventual master thesis.

In the future is planned further extension of DynaCASE tool by modules used to process simulation, reporting, etc.

Further possibilities for development of BORM method and semantics of ORD diagrams are communication conditions or nested processes [11]. This constructs should be also studied in future works.

Personal benefits

Through this work I have acquainted in detail with BORM method, namely about its possible use in process management. I also recognized other side of this method and detect the reasons why it is necessary to define the term of valid diagram and validation. I gained overview about possibilities and potential of Smalltalk language and environment Pharo. In addition to my branch of study I also acquainted with the most important graph algorithms that are included in many validation algorithms.

Bibliography

[1] Knott, R.; Merunka, V.; Polák, J. The BORM Methodology: a third generation fully object-oriented methodology. *Knowledge-Based Systems*, volume 16, 2003: pp. 77 – 89, ISSN 0950-7051.

[2] Podloucký, M.; Pergl, R. The Prefix Machine - a Formal Foundation for the BORM OR Diagrams Validation and Simulation. In *Enterprise and Organizational Modeling and Simulation*, 2014, ISBN 978-3-662-44859-5, pp. 113–133.

[3] Centre for Conceptual Modelling and Implementations: Internal documents, Czech Technical University in Prague, Faculty of Information Technology. 2014.

[4] Centre for Conceptual Modelling and Implementations: DynaCASE documentation. Online, http://github.com/dynacase/.

[5] Merunka, V.; Polák, J. BORM - Business Object Relation Modeling, Popis metody se zaměřením na úvodní fáze analýzy I.S. In *Tvorba softwaru 99*, Ostrava: VŠB-TU v Ostravě, 1999, ISBN 80-85988-39-9, pp. 202–214. Available from: `http://cev.cemotel.cz/programovani_a_tvorba_sw_1975-2003/1999/202.pdf`

[6] Vejražková, Z. *Business Process Modeling and Simulation: DEMO, BORM and BPMN*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2013.

[7] Polák, J.; Merunka, V.; Carda, A. *Umění systémového návrhu: objektově orientovaná tvorba informačních systémů pomocí původní metody BORM*. Praha: Grada Publishing, 2003, ISBN 80-247-0424-2.

[8] OpenCASE. BORM CASE tool. Online, http://opencase.net/.

[9] Moravec, J. *Orchestrace a choreografie procesů v BORM.* Dissertation thesis, Czech University of Life Sciences Prague, Faculty of Economics and Management, 2014. Available from: `http://www.pef.czu.cz/cs/?dl=1&f=29010`

[10] Merunka, V.; Polák, J.; Kofránek, J. Introduction into the BORM Method. 2000, in the symposium of 5th Annual National Conference. Available from: `http://www.grada.cz/dokums_raw/usn/objekty2000.pdf`

[11] Podloucký, M.; Pergl, R. Towards Formal Foundations for BORM ORD Validation and Simulation. In *Proceedings of the 16th International Conference on Enterprise Information Systems*, 2014, pp. 315–322.

[12] Pergl, R. BI-ZPI - Lecture No. 2, Czech Technical University, Faculty of Information Technology. 2014, unpublished presentation.

[13] Zyková, A. *Procesní stroj pro metodu BORM.* Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2014.

[14] Kolář, J. *Teoretická informatika.* Praha: Česká informatická společnost, 2004, ISBN 80-900853-8-5.

[15] Black, A. P.; Ducasse, S.; Nierstrasz, O.; et al. *Pharo by Example.* Square Bracket Associates, Switzerland, 2009, ISBN 978-3-9523341-4-0. Available from: `http://pharobyexample.org/versions/PBE1-2009-10-28.pdf`

[16] Bergel, A.; Cassou, D.; Ducasse, S.; et al. *Deep in Pharo.* Square Bracket Associates, Switzerland, August 2013, ISBN 978-3-9523341-6-4. Available from: `http://pharobooks.gforge.inria.fr/PharoByExampleTwo-Eng/latest/PBE2.pdf`

Appendix A

Acronyms

BORM Business Object Relation Modelling

ORD Object Relation Diagram

BAD Business Architecture Diagram

OBA Object Behavioral Analysis

FSM Final State Machine

MVC architecture Model, View, Controller

DFS Depth First Search

BFS Breadth First Search

UML Unifield Modeling Language

CASE Computer-Aided Software Engineering

BPMN Business Process Model and Notation

Appendix B

Samples of source codes

Source code of validation algorithm described in 3.6, which uses BFS algorithm:

```
unreachableNodes
   | DFSArray initials |
   DFSArray := OrderedCollection new.

   aORD participants do: [ :each | DFSArray addAll: (self getDFSstructsForParticipant: each)].
   initials := DFSArray select: [ :each | (each getNode className = 'BormState') and: [(each
      getNode isInitial)]  ].
   initials do: [ :each | self doBFS: DFSArray node: each ].
   ^ (DFSArray select: [ :each | each isFresh ]) collect: [ :each | each getNode ].


doBFS: DFSArray node: aNode
   | aQueue neighbors node|
   aQueue:=OrderedCollection new.

   aQueue addLast: aNode.
   aNode setOpen.
   [aQueue isEmpty] whileFalse: [
         node:=aQueue removeFirst.
         neighbors := self getFollowsNeighbors: node Array: DFSArray.
         neighbors do: [ :each | (each isFresh) ifTrue: [aQueue addLast: each. each setOpen.] ].
         aNode setClose.
   ].
```

Source code of validation algorithm described in 3.3:

```
emptyParticipants
    ^ aORD participants select: [ :each | each nodes size = 0]
```

Source code of validation algorithm described in 3.1:

```
missingDataFlows
  ^ (self getCommunications) select: [ :each | each dataFlows size = 0 ].


getCommunications
  | communications nodes activities |

  communications:=OrderedCollection new.
  nodes:=OrderedCollection new.
  activities:=OrderedCollection new.

  aORD participants do: [ :each | nodes addAll: each nodes ].
  activities:= nodes select: [ :each | each className = 'BormActivity' ].
  activities do: [ :each | communications addAll: each sent ].
  ^ communications.
```

Source code of validation algorithm described in 3.4:

```
multipleProcessFlow
  | DFSArray counter |
  DFSArray := OrderedCollection new.
  aORD participants do: [ :each | DFSArray addAll: (self getDFSstructsForParticipant: each)].
  counter := 0.

  DFSArray do: [ :each | (each isFresh)
    ifTrue: [ counter:=counter+1. self doDFS: DFSArray state: each counter: counter]
  ].

  (counter>1)ifTrue: [ listOfFailures add: ((MultipleProcessFlow new addCollection:
    DFSArray) numberOfFlows:counter)].
  ^ counter.


doDFS: DFSArray state: aState counter: aCounter
  | neighbor |

  aState setComponent: aCounter.
  neighbor:=self getNeighborWithCommunication: aState Array: DFSArray.
  neighbor do: [ :each | self doDFS: DFSArray state: each counter: aCounter ].
  aState setClose.
```

Appendix C

Diagrams used in tests of complex ORD

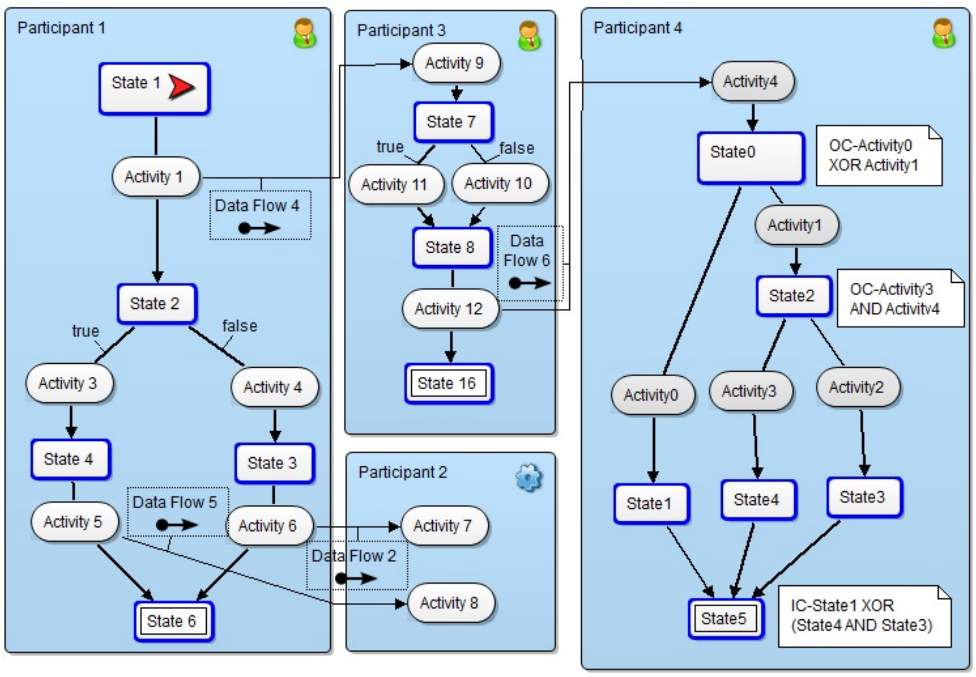


Figure C.1: Valid ORD used in test of complex diagram

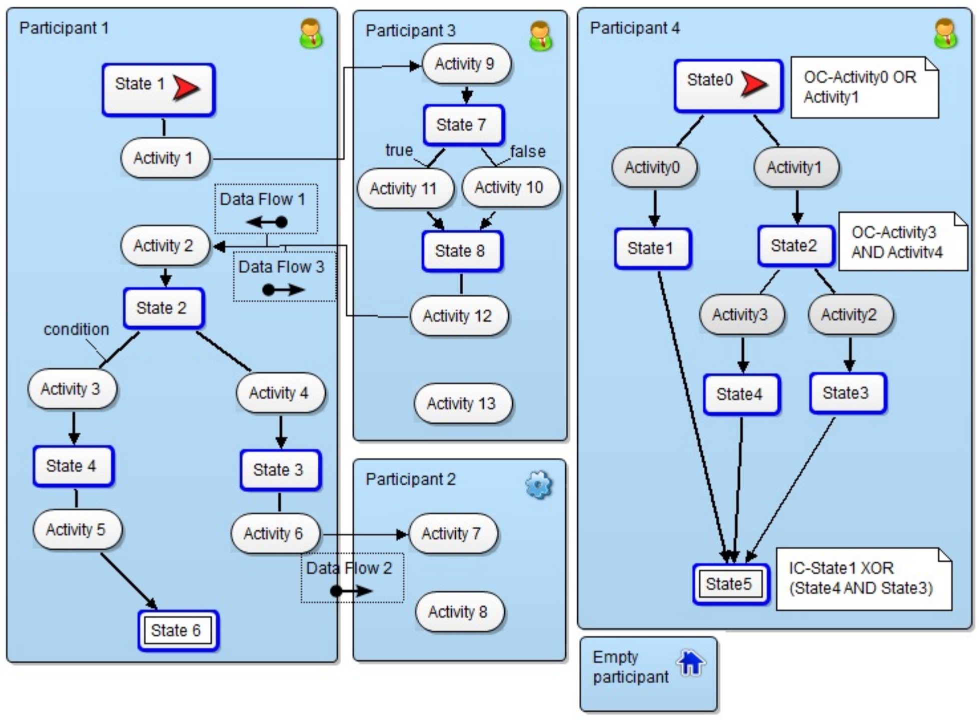


Figure C.2: Invalid ORD used in test of complex diagram

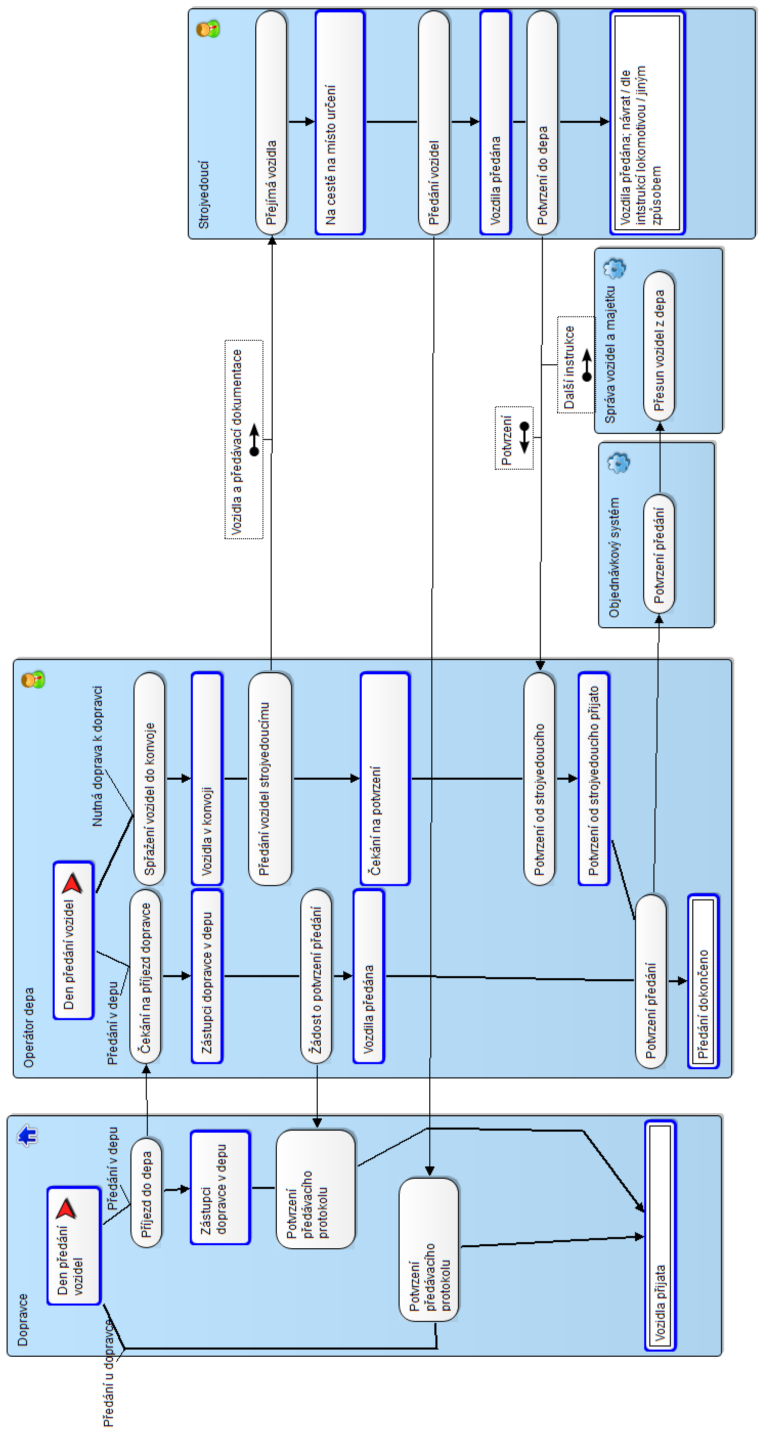


Figure C.3: Selected ORD from semestral projects used in test of complex diagram

Appendix D

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
img ............ images of semestral projects used for a managerial study
src ....................................... the directory of source codes
  BormValidator ......... the directory of source codes of implemented
  validations
  BormModel ........ the directory of source codes of BORM data model
  thesis .............. the directory of LaTeX source codes of the thesis
    img .................................. images used in the thesis
text ........................................ the thesis text directory
  BP_Bambas_Jaroslav_2015.pdf ........ the thesis text in PDF format
  assignment.pdf ........................ the assignment of the thesis
```