Development of a
Domain Specific Language

A.J.T.M. Burgers
2011
Development of a Domain Specific Language

Master’s Thesis

by

A.J.T.M. Burgers

TECHNISCHE UNIVERSITEIT EINDHOVEN
Departement of Industrial Engineering

Graduation supervisor: dr. Ir. J.J.M. Trienekens

Eindhoven, May 2011
Zetes B.V. makes use of transaction types for the support of logistical processes. These transaction types are modeled by arranging pre-constructed building blocks, i.e. executable code, in a certain order. Building blocks provide, require and/or unset settings on a shared registry, creating dependencies between building blocks. In the current situation building blocks can only be arranged in a sequential way, which limits the expressiveness of transaction types. Furthermore, the modeling tool provides no means to validate the building block dependencies, which demands considerable effort for testing. In this thesis, the feasibility is explored of the visual modeling and validation of transaction types, including the support for conditional paths and loops. For this purpose, a Domain Specific Language (DSL) is developed that formalizes the way in which transaction types can be modeled. A proof-of-concept is implemented on the basis of this DSL that demonstrates the suitability of the DSL for the visual modeling and validation of transaction types. Additionally, related to the modeling of transaction types, a set of granularity guidelines for building blocks are given, new localization mechanisms for building blocks are presented and a new database model for the storage of building blocks and transaction types is designed.

**Keywords:** Granularity, Domain Specific Language, GMF
# Table of Contents

1. **Introduction** ............................................................................................................. 1
   1.1 Motivation ............................................................................................................. 1
   1.2 Research Goal ....................................................................................................... 1
   1.3 Structure of the thesis ........................................................................................... 2

2. **Current Situation** .................................................................................................... 3
   2.1 Medea .................................................................................................................... 3
      2.1.1 Transaction types ............................................................................................ 4
      2.1.2 Building Blocks ............................................................................................. 6
   2.2 Configuration process ............................................................................................ 8
   2.3 Problems ............................................................................................................... 9

3. **Literature Study** ..................................................................................................... 11
   3.1 Granularity ........................................................................................................... 11
      3.1.1 Comparison with CBSD and SOA ................................................................. 11
      3.1.2 Effects of Granularity on Quality Aspects .................................................... 13
   3.2 Localization of Building Blocks ......................................................................... 16
      3.2.1 Classification .................................................................................................. 17
      3.2.2 Search function ............................................................................................. 17
      3.2.3 Recommender System ............................................................................... 18
      3.2.4 Templates ..................................................................................................... 19
   3.3 Development Environment ............................................................................... 20
   3.4 Transaction Type Representation ....................................................................... 21
      3.4.1 Workflow Patterns ....................................................................................... 22
      3.4.2 Domain Specific Languages ....................................................................... 23
   3.5 Overview ............................................................................................................. 25

4. **Granularity Guidelines** ......................................................................................... 27
   4.1 Concept ............................................................................................................... 27
   4.2 Implementation ..................................................................................................... 28

5. **Localization** .......................................................................................................... 34
   5.1 Search function .................................................................................................... 34
      5.1.1 Concept .......................................................................................................... 34
      5.1.2 Implementation ............................................................................................. 36
   5.2 Templates ............................................................................................................. 37
      5.2.1 Concept .......................................................................................................... 37
      5.2.2 Implementation ............................................................................................. 38

6. **DSL Development Process** .................................................................................. 39
   6.1 Decision Phase ..................................................................................................... 39
   6.2 Analysis Phase ..................................................................................................... 40
      6.2.1 Analysis Approach ......................................................................................... 40
      6.2.2 Domain Definition ....................................................................................... 41
      6.2.3 Domain Concepts ......................................................................................... 42
6.2.4 Domain Assumptions ................................................................................. 43
6.2.5 Meta-model .................................................................................................. 46
6.3 Design Phase ................................................................................................... 50
6.3.1 Design Approach ......................................................................................... 50
6.3.2 Concrete Syntax .......................................................................................... 53
6.3.3 Execution Semantics ................................................................................... 56
6.4 Implementation Phase .................................................................................... 60
6.4.1 UML ............................................................................................................. 61
6.4.2 Graph Structure .......................................................................................... 61
6.4.3 Modeling Tool ............................................................................................. 63
6.4.4 Proof-of-Concept ......................................................................................... 64
7. Database Design .................................................................................................. 70
7.1 Concept ........................................................................................................... 70
7.2 Implementation ............................................................................................... 71
7.2.1 Active registry settings & Variant codes ....................................................... 73
7.2.2 Conditional paths ....................................................................................... 73
7.2.3 Invokables .................................................................................................. 74
7.2.4 Containers .................................................................................................. 74
7.2.5 Tags ............................................................................................................. 75
7.2.6 Normalization ............................................................................................. 75
8. Validation ............................................................................................................ 76
8.1 Research Goal .................................................................................................. 76
8.2 Sub-goals ......................................................................................................... 77
9. Future Work ....................................................................................................... 79
10. Conclusions ...................................................................................................... 82
References ........................................................................................................... 84
Appendices ........................................................................................................... 87
Appendix A: Terminology ...................................................................................... 87
Appendix B: Interview Scheme 1 ........................................................................... 89
Appendix C: Interview Scheme 1 – Interview results ............................................. 90
Appendix D: Interview Scheme 2 ........................................................................... 94
Appendix E: Interview Scheme 2 – Interview results ............................................. 95
Appendix F: Thesis Presentation Survey ................................................................ 97
# List of Figures

| Figure 2.1: Part of the Medea platform | .......................................................... | 3  |
| Figure 2.2: A simplified example of a transaction type | .......................................................... | 4  |
| Figure 2.3: Transaction Type Setup Example | .......................................................... | 5  |
| Figure 2.4: Building Block Model | .......................................................... | 6  |
| Figure 2.5: Transaction Type & Building Block database tables | .......................................................... | 7  |
| Figure 2.6: Overview of Configuration Process | .......................................................... | 8  |
| Figure 3.1: Granularity Comparison [12] | .......................................................... | 13 |
| Figure 4.1: Decomposing control flow logic | .......................................................... | 29 |
| Figure 4.2: Candidate construction for decomposition | .......................................................... | 30 |
| Figure 5.1: Transaction Type table | .......................................................... | 38 |
| Figure 6.1: DSL development overview | .......................................................... | 47 |
| Figure 6.2: UML Domain model; Abstract Syntax | .......................................................... | 48 |
| Figure 6.3: Transaction Type Process Overview | .......................................................... | 51 |
| Figure 6.4: Domain Object Notations | .......................................................... | 53 |
| Figure 6.5: Sequence representation | .......................................................... | 53 |
| Figure 6.6: Exclusive Choice representation | .......................................................... | 54 |
| Figure 6.7: Merge representation | .......................................................... | 54 |
| Figure 6.8: Loop representation | .......................................................... | 55 |
| Figure 6.9: Un-allowed loop | .......................................................... | 55 |
| Figure 6.10: Example Transaction Type | .......................................................... | 56 |
| Figure 6.11: GMF process overview | .......................................................... | 65 |
| Figure 7.1: Transaction Type & Building Block database tables | .......................................................... | 71 |
| Figure 7.2: New database model | .......................................................... | 72 |
List of Tables

Table 3.1: Mapping of Technical Features to Managerial Goals [3] .......................................................... 14
Table 3.2: Literature Study Overview ........................................................................................................ 26
Table 6.1: Domain Concepts and their purposes ......................................................................................... 43
Table 6.2: Attributes for the Connection element ...................................................................................... 57
Table 6.3: Attributes for the Start State element ......................................................................................... 58
Table 6.4: Attributes for the End State element ......................................................................................... 58
Table 6.5: Attributes for the Building Block element ............................................................................... 59
Table 6.6: Attributes for the Exclusive Choice element ............................................................................. 60
Table 6.7: Vertex details .............................................................................................................................. 62
Table 6.8: Edge details ................................................................................................................................. 62
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Automated Dependency Checker</td>
</tr>
<tr>
<td>AM</td>
<td>Account Manager</td>
</tr>
<tr>
<td>ARS</td>
<td>Active Registry Setting</td>
</tr>
<tr>
<td>BB</td>
<td>Building block</td>
</tr>
<tr>
<td>BPA</td>
<td>Business Process Analysis</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Modeling Notation</td>
</tr>
<tr>
<td>CBSD</td>
<td>Component-based Software Development</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial of-the-shelf</td>
</tr>
<tr>
<td>DJ graph</td>
<td>Dominator Join graph</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>DSML</td>
<td>Domain Specific Modeling Language</td>
</tr>
<tr>
<td>DSVL</td>
<td>Domain Specific Visual Language</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>EMOF</td>
<td>Essential Meta Object Facility</td>
</tr>
<tr>
<td>GMF</td>
<td>Graphical Modeling Framework</td>
</tr>
<tr>
<td>GPL</td>
<td>General Purpose Language</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>KU</td>
<td>Key User</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-oriented Programming</td>
</tr>
<tr>
<td>PE</td>
<td>Product Engineer</td>
</tr>
<tr>
<td>RF-terminal</td>
<td>Radio Frequency terminal</td>
</tr>
<tr>
<td>SA</td>
<td>Software Architect</td>
</tr>
<tr>
<td>SE</td>
<td>Software Engineer</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SOC</td>
<td>Service Oriented Computing</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Motivation

Zetes Industries is a pan-European company that provides services and solutions in the area of automated identification of goods and people (goods-ID and people-ID). In the area of warehouse management, Zetes makes use of RF-terminals to facilitate logistical processes such as receiving, returning, restocking and order-picking of goods.

These processes are modeled as transaction types; for instance ‘move goods between warehouses’. Zetes uses a PC-application, called Medea, for the modeling of transaction types by making a selection from a list of standard building blocks. A building block represents a single action, such as scanning an item, specifying amounts or selecting a destination, although more complex actions are possible as well. A single transaction type is defined as a sequence of building blocks. For the modeling of a transaction type, Medea can be used to select building blocks and to arrange them in a certain order.

The current situation has certain drawbacks though, dependencies between the building blocks are not visible in Medea and are not checked during the modeling of a transaction type. This requires a user to continuously consult the documentation, which makes the modeling of transaction types time-consuming and unnecessarily prone to errors. Another drawback is the limitation that users are only able to arrange the building blocks in sequence; transaction types are always executed by starting at the first building block and proceeding until the last building block.

1.2 Research Goal

This thesis, submitted as part of the fulfillment of the requirements for the Degree of Master of Science in Business Information Systems at the Technical University of Eindhoven, documents the conducted research and results aimed at removing the drawbacks in the modeling of transaction types stated above.

The research goal can be stated as follows:

- Design a formal language specification for the modeling of transaction types using the standard building blocks.

This formal language specification should at least meet the following requirements:

- It should support the modeling of sequential flow.
- It should support the modeling of conditional paths.
- It should support the modeling of loops.
- It should make automated dependency checking of building blocks possible.
- It should be suitable to be used as a basis for a graphical development environment that allows for visual modeling of transaction types using the standard building blocks.

To demonstrate the suitability of the formal language specification, a proof-of-concept shall be realized. This proof-of-concept shall, using the formal language specification as a basis, consist of a graphical development environment for the modeling of transaction types by drag-and-drop of components onto a canvas. The above mentioned requirements with respect to the formal language specification shall be incorporated in the proof-of-concept.

1.3 Structure of the thesis

Chapter 2 discusses the current situation at Zetes with respect to the development of transaction types. Analysis of the modeling software, the building blocks, the configuration process and the documentation combined with interviews with several Zetes employees results in a list of limitations and problems concerning the development of transaction types.

As a result of the problems stated in Chapter 2, a literature study is conducted to further explore the problems, their causes and possible solution directions. Chapter 3 covers the result of this literature study and the concepts that shall be used to solve the identified problems.

Chapters 4, 5, 6 and 7 each deal with solving a subset of the problems mentioned in Chapter 2. In Chapter 4 granularity guidelines for building blocks are presented. Chapter 5 deals with the localization of building blocks. The development of a suitable Domain Specific Language for modeling transaction types is described in Chapter 6 and in Chapter 7 a new database model for the storage of building block and transaction type information is designed.

Several Zetes employees are asked to fill out a survey concerning the writings in the above mentioned chapters and the proof-of-concept. The results of these surveys serve as part of the validation for this thesis, which is presented in Chapter 8. Lastly, possible future work and the conclusions are presented in Chapters 9 and 10. Followed by the references and appendices.

It is assumed that readers of this document have knowledge of general IT-concepts. Furthermore, knowledge of databases and software architecture aids in the understanding of the material presented.
2. Current Situation

This section consists of a description of the current situation at Zetes, concerning the modeling of transaction types. Parts that are described are successively the Medea software and the corresponding platform with specific attention for transaction types and building blocks, the process from customer requirements to deployment and lastly the current problems that are associated with the modeling of transaction types.

2.1 Medea

Medea is a software solution for the development and execution of logistical processes support. The PC-application, part of Medea, can be used to provide a configuration of a standard platform that suits the needs of a specific customer.

Figure 2.1: Part of the Medea platform

This platform, as can be seen in Figure 2.1, consists of a database and an application server (including redundant ones for availability assurance). The database server hosts an Oracle Database and the application server hosts a variety of applications, such as the Medea PC application, a Telnet server and the applications that handle transaction types.
Both servers are connected with each other and several other hardware devices: the Radio Frequency terminals (RF-terminal) used by warehouse personnel, a label printer for printing barcodes and some other devices that are not of interest of this thesis.

In the execution of a typical transaction type, the RF-terminals connect to the Telnet server on the application server, which then connects through the RF application with the database server or the printer.

### 2.1.1 Transaction types

Transaction types represent processes to support warehouse management, such as receiving, returning, restocking and order-picking of goods. For the modeling of transaction types, building blocks with limited functionality are selected and arranged in a certain order to achieve a certain goal.

Warehouse personnel is equipped with a portable RF-terminal that is connected to a local server via a telnet client on the terminal. A user requires to login after which he is presented with a list of transaction types he has been authorized to execute. After selection of one of the available transaction types, the corresponding building blocks are executed in the predefined sequential order. During the execution the user can be requested to provide information to the terminal in one of two ways. Through the RF-scanner by scanning barcodes and through user input from the keyboard of the terminal. The building blocks can also query and update the database for information if necessary.

Execution of a transaction type on a RF-terminal is called a *transaction*. Successful execution of a transaction type will result in a registration of the transaction to be added to the database. Besides the registration, the transaction could also store or alter database information regarding, for example, storage locations and inventory. See Figure 2.2 for a simplified example of a transaction type consisting of four building blocks.

Transaction types are executed in a looped fashion; after execution of the last building block, the RF-terminal loops back to the first building block of the transaction type. Every iteration of the loop represents a single transaction and ends with a database commit. This continues until the user decides to manually return to the transaction-type-selection-menu by pressing the appropriate button.

Originally, all transaction types were meant to start by scanning an item, i.e. a box, a pallet, etc. and to end with a database commit. In between, a number of building blocks could be added to be executed in a specified, sequential order. Therefore, all transaction types start with item scanning and end with a database commit. Later, additional complexity was required to able to provide solutions to increasing complex customer needs.

More specifically, transaction type developers came upon situations where it was required that certain building blocks were executed before the scanning of an item, that database commits were required before the end of the transaction or that building blocks were executed when closing a transaction...
type. This resulted in the possibility to mark building blocks, during the modeling of a transaction type, as include in startup, as end transaction or as include in closing.

As has been mentioned earlier, transaction types are continuously being looped on execution. By specifying a building block as either include in startup or include in closing these building blocks are executed one extra time before the loop and after the last iteration of the loop respectively. Building blocks marked as end-transaction will perform a database commit as if they are the last building block of the transaction type. The automatic commit at the end of the transaction type will be skipped when one of the building blocks in the transaction type is marked as end transaction.

If a building block is not marked as one of the three above mentioned states it is considered as a standard building block; a standard building block is executed as part of the transaction type in a regular fashion.

![Figure 2.3: Transaction Type Setup Example](image)

In Figure 2.3 the setup of a transaction type can be seen. The ‘Scan Item’ block is automatically executed at the beginning of every loop. The other blocks are building blocks that are added to a transaction type with the appropriate marking.

The ‘Scan Item’ block requires the scanning of a barcode and stores this as being an item. However, it is no longer always required that a transaction type starts with the scanning of an item; sometimes locations or individual articles need to be scanned. For this reason, building blocks have been created that use the functionality for scanning an item, to scan barcodes of other entities, for example a warehouse location. In this case, ‘Scan Item’ sets the variable ITEM ID NUMBER. A subsequent building block can use the barcode stored in this setting to set the variable WAREHOUSE.

Having to deal with the ‘Scan Item’ block this way and the different markings for the building blocks have resulted from the increasing need to be able to model more complex and varied transaction types. The requirements on the transaction types have evolved beyond the initial anticipated need. The ‘Scan Item’ block ideally should not be integrated and automatically executed, but should be available in a separate building block that can be added to a transaction type when necessary at a position of the modeler’s choice. In a similar way, there should not be a automatic database commit after the last building block of the transaction type, but the point at which the commit is required should be specifiable during the modeling of the transaction type; possibly in the form of a building block replacing the functionality of the ‘End transaction’ marking. Having the possibility to define loops within a transaction type also removes the need to automatically loop transaction types, which makes
the ‘Include in startup’ marking obsolete. By being able to manually specify loops and the conditions under which the loops are exited, the user is able to model when the transaction type should end. By adding the building blocks marked as ‘Include in closing’ at this point, this marking is no longer necessary either.

Summarizing the above paragraph, contrary to the current situation, an empty transaction type should have no standard functionality. However, the modeler should have complete freedom in choosing which functionality to add and in which way.

2.1.2 Building Blocks

Building blocks are the components of which a transaction type is composed. They comprise of an interface, documentation and their implementation. The interface in turn consists of a list of required, provided and remove variables and variant codes (see Figure 2.4).

On the terminal a shared registry is available to the building blocks. Building blocks use this registry to write variables to and read variables from; so called active registry settings. In this way, building blocks can provide information to subsequent building blocks and use information provided by preceding building blocks respectively. Some building blocks also remove (or unset) certain active registry settings. The registry is one of two ways for building blocks to exchange information between each other, along with reading and writing information from and to the database. Which variables a building block reads, writes and removes, in other words a list of required, provided and unset active registry settings is part of its interface.

Through this interface a runtime dependency relation exists between building blocks; a building block requires that certain active registry settings have been set by preceding building blocks. If a building block tries to access an active registry setting that has not yet been set, the transaction type will provide an error message and terminate.

![Building Block Model](image)

Figure 2.4: Building Block Model
For every building block five variant slots are available. These variant slots can be filled with a code (a single letter or number) chosen from a domain list. They serve to slightly alter the functionality of a building block. For example, skipping or performing a certain variable check. Some building blocks require that a number of these variant codes are set during the modeling of a transaction type, others require no variant codes.

Documentation provides the information about which variant codes are available for each building block and what the functionality is that they alter. This documentation, furthermore, provides an informal description of the general functionality of the building block.

The code for the building blocks is written in the Delphi programming language, in the programming environment Delphi 7. Source files are stored on a server which runs PVCS Version Manager, which is a software package for revision control of (in particular source code) files.

Building blocks are also represented in a database such that they can be used in transaction types, which are also stored in the same database. The transaction types are stored using three tables, namely Transactiontype, Optional_function and Activated_function. An overview of these tables and their attributes can be seen below in Figure 2.5. This model uses the Crow’s foot notation to indicate the cardinality of the relationships.

![Figure 2.5: Transaction Type & Building Block database tables](image)

The table Transactiontype contains for each transaction type some general information, such as a transactiontype_description, and details related to its execution, such as max_repeat_time, min_cycle_time and message_not_found. A unique code for each transaction type is used as the primary key. Of which building blocks a transaction type is comprised is not stored in this table. The exact meaning of each of the attributes is not of interest for this thesis.

The table Optional_function contains information on the available building blocks. A description, an identifying code and a generally used hotkey for each available building block are stored in this table. Both mentioned tables are connected through the Activated_function table. Here, the functionality details of the transaction types are stored, i.e. of which building blocks the transaction type is comprised, in what order they should be executed, the variant codes that are used and the markings, indicated by either a 0 or 1 (Include in startup, Include in closing and End transaction). The primary key of this table consists of the primary keys of both previously mentioned tables and the sequence number indicating the order of a building block in the transaction type. From Transactiontype to Activated_function exists a one-to-many relation, as well as from Optional_function to Activated_function.
2.2 Configuration process

The software department at Zetes consists of 7 Software Architects (SA) and 8 Software Engineers (SE). SAs are concerned with what has to be done or constructed and how. SEs are primarily concerned with implementation issues; implementing new building blocks, changing existing building blocks, changes to the PC-application, etc.

An SA is also involved in almost the entire process of delivering logistical solutions to a customer; from defining the problem to deployment, training and support. Figure 2.6 shows the process which will be described in more detail below.

![Figure 2.6: Overview of Configuration Process](image)

Based on a problem description and preliminary information, gathered from meetings between the customer, an SA and an Account Manager (AM), an initial offer is prepared. Upon acceptance of this offer by the customer, the SA proceeds to perform a detailed Business Process Analysis (BPA) on the business in question. The result of the BPA allows the SA to decide, given the required functionality, if the Medea solution will be a 100% fit for the problem at hand. For solutions using RF-terminals it needs to be decided which transaction types are to be modeled. The next step is the design phase in which the SA models the actual transaction types; selecting building blocks, selecting variant codes, defining the execution order and setting the building block markings. If it should occur that the building block collection does not provide the necessary functionality, this should be added. For this, the SA decides upon making a new building block or altering an existing one and documents which functionality is needed; this is documented in an altered or newly created building block design document. He delivers this information to an SE who is responsible for the implementation. This along with other implementation issues of the SEs are performed in the implementation phase.

Every completed transaction type then needs to be tested, which is done by an SA again. Remaining errors are fixed by either an SA or an SE, depending on the nature of the error, after which (part of) the system is deployed. In this phase the Product Engineer (PE) takes care of establishing the required platform and installing the system. In case of deployments of large systems an SA usually will provide the customer with a general explanation of the system.

Specific training is done by either an SA, an SE or a combination of both. One person (or at most a few) from the target business is assigned as Key User (KU) and receives a full training of all aspects of the system. He is then responsible for training additional personnel for as far as that is necessary. After training, the business is ready to work with the deployed system. In case support is needed, the key
user is the first who needs to be addressed. For additional support Zetes has a Support Engineer (SuE) available and if necessary the involved SAs and SEs can provide support.

Zetes, in practice, never designs and deploys the entire system at once. It however designs and deploys the system incrementally; the phases Design until Support are executed in an iterative fashion. This way, the customer is given, from early on, the opportunity to provide feedback and validate the results. He is thus able to steer the way the eventual system works. After deployment of the full system, the support phase continuous on.

2.3 Problems

After analysis of the Medea software, the transaction types, the building blocks, the configuration process, consulting the documentation and interviews with three of the System Architects at Zetes¹, a number of problems were identified. Some of the problems have been briefly mentioned in the previous sections. In this paragraph more details will be given where necessary and a list of all problems will be given for completeness.

Firstly, there are no guidelines concerning the amount of functionality a building block should or is allowed to contain, i.e. there are no granularity guidelines. A building block with a lot of functionality becomes specific for a certain kind of activity, which limits the range of transaction types for which the building block can be used. Without guidelines to steer the granularity, the possibility exists that a large number of overly specific building blocks are introduced in the collection, which goes against the nature of the component-based development paradigm.

Secondly, searching through the documentation or the extensive list of building blocks is the only way to locate a certain building block. For a development process that relies on reuse, it is necessary that the right building block can be found with a limited amount of effort. At the moment, however, it is a time-consuming task that could be sped up by having mechanism in place that facilitate finding a certain building block.

Thirdly, during the modeling of transaction types, the modeling environment gives the user no information about what active registry settings are available at every point in the transaction and which active registry settings are required and provided by a specific building block. The only information about active registry settings can be found in the individual documentation for every building block.

This means that when a user wants to use a certain building block, he is required to consult the documentation for that building block first, look which active registry settings are required by that

¹ C. van Liempd, H. Donkers and N. Schuitvlot; the question list of the interview can be found in Appendix B: Interview Scheme. The results of the interview can be found in Appendix C: Interview Scheme 1 – Interview results.
building block and then search the documentation for building blocks that provide those active registry settings. Needless to say, this is a very inefficient way of working; users spend significant amounts of time looking up dependencies instead of actually constructing transaction types.

Fourthly, similar to active registry settings, no information regarding the variants of building blocks is currently being displayed in the modeling environment. That is, without consulting the documentation, users have no way of knowing whether or not a building block requires any variant codes, what codes can be chosen from and what functionality variations those codes represent.

Displaying the provided and required active registry settings and relevant variant information of a building block in the modeling environment will lower the time spent consulting documentation. It should be noted that experience with the modeling of transaction types and the (partial) reuse of existing transaction types helps to minimize the need to consult building block documentation.

Fifthly, besides the fact that dependencies are not displayed in the modeling environment, there are also no built-in checks for these dependencies. In other words, during design-time it is possible to construct a transaction type with any number of building blocks in any order. Only during the execution of that transaction type will the user be notified if a building block is missing a required active registry setting, namely when that building block tries to access it and it’s not there. Because the modeling environment cannot ensure that active registry settings are available when they are needed, more time is required for testing and redesign than is necessary. If the modeling environment would have the dependency information, the user could be notified during design-time which dependencies have not yet been fulfilled. This would eliminate the errors caused by missing active registry settings.

Sixthly, when a run-time error occurs, the user is only informed of which building block misses an active registry setting and not which one. A user then has to manually determine which setting is missing, which once again requires the user to look in the documentation.

Lastly, building blocks within a transaction type can only be executed in a sequential manner. For a transaction type a number of building blocks can be selected and arranged in a sequential order; there is no possibility for modeling building blocks in conditional paths or loops. Conditional paths and loops at this point are programmed inside the building blocks. By moving part of the control flow inside the building blocks the flexibility of transaction types decreases; flexibility is achieved by making a distinction between the stable parts of the system (i.e. the building blocks) and the specification of their composition (i.e. the control flow) [2]. Moving control flow inside the building blocks also leads to them encapsulating more functionality, which makes them less reusable [3].

A complete list of the problems:

1. Granularity guidelines for building blocks are missing
2. The modeling environment lacks mechanism for locating building blocks
3. Building block requirements/provisions are not visible in the modeling environment
4. Variant information is not visible in the modeling environment
5. The modeling environment delivers no automated dependency checking
6. Error messages do not provide useful information
7. Conditional paths and loops are not possible within a transaction type
3. Literature Study

Based on the problems mentioned at the end of the last chapter a literature study has been performed to more accurately determine the cause of these problems and explore concepts and techniques that could aid in solving them. First off, the lack of granularity guidelines is explored, followed by the lack of localization mechanisms for building blocks. The limitations of the current modeling environment are handled next and finally the automated dependency checker for transaction types.

3.1 Granularity

The first problem concerns the lack of granularity guidelines. This became clear during the interviews on building blocks (see Appendices B & C) and it was voiced that part of the building block collection might have a suboptimal granularity, which is an indication of the amount of functionality a building block encapsulates. Granularity is the relative size, scale, level of detail, or depth of penetration that characterizes an object or activity [4]. In the case of software engineering it indicates the extent to which a system is broken down into small parts; the more a system is broken down into small parts the more fine-grained the granularity of the system is. The opposite of fine-grained is coarse-grained. The following sections are written in order to answer the question: how to choose the right level of granularity?

Much has been written about answering this question and also the relation between granularity and flexibility, reusability and other quality aspects [1,2,3]. Most of this literature, however, has been written in the context of Component Based Software Development (CBSD) and Service Oriented Architecture (SOA). In order to effectively use the literature sources in these areas, the differences and commonalities of CBSD, SOA and transaction type modeling at Zetes shall be discussed first. Next, a list of applicable quality aspects and their relation to granularity shall be given. The results of this shall, in Chapter 4, be used to create granularity guidelines for Zetes.

3.1.1 Comparison with CBSD and SOA

CBSD and SOA are both software development methods aimed at improving the productivity and reducing cost in software development projects by composing a complex software system out of smaller parts.

"[CBSD] advocates the acquisition, adaptation and integration of reusable software components, including commercial-off-the-shelf (COTS) products, to rapidly develop and deploy complex software systems with minimum engineering effort and resource cost." [5]

This is one available definition, although many variations exist. The commonality among all of them is the composition of existing software components to aid the development of an entire software
system. This is similar to the modeling of transaction types using building blocks. The transaction
types can be seen as the software system that is being developed and that has certain functionality
requirements and the building blocks as the software components.

For the definition of a software component are a number of definitions as well. One that is often used
is the one by Szyperski;

“A software component is a unit of composition with contractually specified interfaces
and explicit context dependencies only. A software component can be deployed
independently and is subject to composition by third parties.” [8]

The interface of a component is specified by a list of methods and variables that can be accessed
externally. This is different from the interface of a building block where only the required, provided
and unset active registry settings and the required variant codes are specified. However in both cases,
the interface specifies which information is required as input and which output is delivered. Where a
component can have multiple methods and functions that can be called and executed indiscriminately,
a building block can only be executed as a whole and usually has a single goal; building blocks can be
seen as more fine-grained than components.

An important difference is in the implementation of building blocks, components and services. Building blocks consist of a single class, that inherits the basic building block structure of the base
building block class. These classes do have methods and functions, but none of them public. Building blocks are not able to directly use functionality of other building blocks. This is different from
components/services that are built from a multitude of classes and are able, through their public
interfaces, to use functionality of other component/services.

These differences are mostly of a technical nature and do not concern the way the composition of both
building blocks and component are used to aid the development of a larger software system. Therefore
most relations between granularity and quality aspects specified for components uphold for building
blocks (more on this in Section 3.1.2).

Service Oriented Architecture, also known as Service Oriented Computing (SOC), is defined as
follows.

“SOA presents well-defined business functions as services, which are made available
to multiple applications through standard protocols. Using SOA, institutions can integrate
business functions into new and interesting applications. […] SOA is built on reusable,
shared, networked services, with each service a business function.” [7]

As components, services are made publically available to be used by a variety of consumers. In order
to achieve this, services also rely on standardized interfaces and messaging protocols. Services are in
many ways similar to components. Services are however in general more coarse-grained. SOA can be
considered a progression from object-oriented programming (OOP), to CBSD and now to SOA, where
this corresponds to a progression from fine-grained objects to medium-grained components and now
to large-grained services [6], as is visualized in Figure 3.1.
Similar to components, the effect of granularity on quality aspects for services are mostly still applicable to building blocks. Services do implement more, coarser-grained functionality, which makes building blocks closer to components than services. In the next section, therefore, literature on components and CBSD will receive the most attention.

### 3.1.2 Effects of Granularity on Quality Aspects

Granularity of components and its effects on quality aspects of a software system in the area of CBSD and SOA has been well documented [1,2,3,7,9]. Each paper, however, discusses different quality aspects and has different ways for exploring the relation between granularity and those quality aspects. In this section, a list of quality attributes that are applicable to building blocks and their relation to granularity are given.

The ISO/IEC 9126 Software Engineering – Product Quality [11] standard is used internationally for the evaluation of software quality. This standard aims to develop a common understanding of a project’s goals and objectives. The first part of the standard, the Quality Model, classifies software quality in a structured set of characteristics. These characteristics are Functionality, Reliability, Usability, Efficiency, Maintainability and Portability\(^2\), which are all divided into several sub-characteristics. This standard is however designed for software systems as a whole and not for software components specifically; therefore these characteristics are not all applicable to the situation at Zetes.

In [10] a list of eleven quality attributes is given that is specific for CBSD and components. However, here again, not all attributes are applicable to Zetes building blocks; scalability and performance, for instance, are of little importance due to the limited size and performance issues in the use of building blocks. Other attributes are very similar or at least closely interrelated and can therefore be combined.

In [3] the available quality attributes, here called *managerial goals*, have been reduced, after refinement and removal of duplicates, to five attributes specific for components; all of which are desirable goals. Namely, Cost Effectiveness, Ease of Assembly, Customization, Reusability and

\(^2\) For definitions of these and quality attributes mentioned in the remainder of this paper, see Appendix A: Terminology.
Maintainability. In addition to these managerial goals, this paper presents five technical features of CBSD and components. Table 3.1 taken from the paper in question shows how making the technical features higher/stronger/coarser affects the managerial goals.

<table>
<thead>
<tr>
<th></th>
<th>COUPL</th>
<th>COHES</th>
<th>NCOMP</th>
<th>CSIZE</th>
<th>COMPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>COST</td>
<td>0</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>ASBL</td>
<td>−</td>
<td>0</td>
<td>0</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>CUST</td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>REUS</td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>MNTN</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
</tbody>
</table>

*Legend:*  
− negative impact, + positive impact, 0 no impact

<table>
<thead>
<tr>
<th>Managerial Goals (rows)</th>
<th>Technical Features (columns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. COST: Cost effectiveness.</td>
<td>1. COUPL: Coupling.</td>
</tr>
<tr>
<td>2. ASBL: Ease of assembly.</td>
<td>2. COHES: Cohesion.</td>
</tr>
<tr>
<td>3. CUST: Customization.</td>
<td>3. NCOMP: Number of components.</td>
</tr>
<tr>
<td>4. REUS: Reusability.</td>
<td>4. CSIZE: Component size.</td>
</tr>
<tr>
<td>5. MNTN: Maintainability.</td>
<td>5. COMPL: Complexity.</td>
</tr>
</tbody>
</table>

Table 3.1: Mapping of Technical Features to Managerial Goals [3]

This paper serves as a good basis for the construction of guidelines with respect to the technical features and granularity (here called component size). However, a one-on-one application to the situation at Zetes is not possible due to the way the technical features are defined.

Some definitions are not applicable to the situation at Zetes and thus neither are the metrics mentioned in the paper. In the remainder of this section, altered definitions and metrics will be given for the technical features and, relying on literature and logical arguments, it will be answered how the changed metrics affect the relation to the managerial goals.

The first technical feature, Coupling, is defined as:

“The extent to which classes within a component relate to the other classes, which are not in that component.” [3]

The importance of coupling as a technical feature lies in the fact that inter-component communication is expensive in terms of time and platform resources. Building blocks, contrary to components, do not consist of multiple classes, but of only one class. They are however, similar to components, able to call and execute other building blocks although it rarely occurs. It makes sense therefore to define the coupling of a building block, a measure of its relation with other building blocks, as the number of other building blocks it calls.

The arguments given in the paper that a high coupling has a negative effect on Ease of Assembly, Customization, Reusability and Maintainability are based on the (in)dependency of components. These arguments still hold for coupling of building blocks.

The Cohesion attribute has the following definition:

“Cohesion of a component is the extent to which its classes are interrelated.”[3]
Building blocks consist of a single class though, with a single functional goal. All building blocks can thus be seen as being highly cohesive. This is a consequence of the way building blocks are designed and deployed at Zetes. The relation of cohesion with the managerial goals is thus not of importance for the situation at Zetes.

The Number of components has been included as a technical feature for its influence in the design of software systems, or transaction types in our case. The more components, i.e. building blocks, are available to the user, the more choice he has in selecting components that together satisfy a certain user requirement. The metric for measuring this feature is straightforward the number of components available for composition as it is in the selected paper. All relations in Table 3.1 with respect to the number of components are also the same. The number of building blocks does also have an effect on the ability to easily find the building block a user wants, i.e. it becomes more difficult as the number of building blocks increase. The aspect of locating building blocks shall be discussed in Chapter 5.

The Component size (i.e. granularity) is in the paper in question defined as the number of classes in the component. Again, for building blocks, a different way has to be found to measure its size. How to measure granularity is a difficult problem, especially for building blocks as they have no classes which is often used as a metric. Function points [16] are another metric for granularity, but this is also more suited for components and services; these have public methods which can be called independently and each (or not) implement a function point. Building blocks, however, can only be executed as a whole and usually only implement a single function. A more old-fashioned metric like lines-of-code could be used, but this would require a standardized way of coding and would then still be a poor stand-alone metric for granularity [17].

A suitable and usable metric for building block granularity is described in [1]. Here the authors define business value granularity as:

“The extent to which a service directly contributes to a high-level business goal.” [1]

As an example, with respect to business value granularity the building block ‘Split and merge item’ is more coarse-grained than the building block ‘Check expiration date reception’, which is more coarse-grained than the building block ‘Scan article’. Use of this metric should be aimed at the core functionality of the building block, i.e. its functionality regardless of the variant codes used, since these merely represent variations of the same functionality.

This metric cannot be used as a precise measurement for granularity, but does serve as a way to compare building blocks on the granularity scale, which is sufficient for our needs. This also does not alter the way granularity affects the managerial goals.

[3] uses a combination of the number of classes and number of public methods and method parameters of those classes as a Complexity metric. As an alternative to this, for building blocks the complexity shall be measured by the cyclomatic complexity metric.

Cyclomatic complexity is a software metric used to provide an indication of the complexity of a section of source code; a building block in our case. The cyclomatic complexity equals the number of linearly independent paths through the source code [14]. Simply put, for each conditional statement
and each loop, the cyclomatic complexity increases with one. Although the metric differs to the one used in the paper, the arguments for how complexity affects the managerial goals are still valid.

Thus the table presented above holds true for Zetes building blocks and the modeling of transaction types. The only point of interest is that due to the very low amount of coupling and the fixed high cohesion of building blocks, these technical features are of less relevance. The results from this section shall support the creation of granularity guidelines for building blocks in Chapter 4.

### 3.2 Localization of Building Blocks

The second problem concerns the lack of mechanisms to locate required building blocks. Crucial to a paradigm where a system is developed by composing individual parts, is the ability to find the right part with a limited amount of effort. Having the right parts in the collection is no good if they’re not being used because of the effort required to find them or because their existence isn’t known to the user. This is true for the modeling of transaction types using building blocks as well.

Not being able to find the right building blocks, will lead to users either modeling the transaction type using an alternative, suboptimal construction or to creating a new building block with the required functionality (while such a building block already exists in the collection). In the former case, transaction types will become less compact and comprehensible and possibly would require a user executing the transaction type to perform more actions to reach the goal than is required. In the latter case, the collection would become filled with unnecessary, duplicate building blocks. This is an unwanted situation for several reasons; building blocks will become even harder to find, because of the increasing size of the collection, and different users might end up using different building blocks in similar situations which decreases the uniformity and therefore comprehensibility of transaction types. More problems ensue once functionality for one of the duplicate building blocks is added or altered. This makes previously similar building blocks different, without this being immediately clear to all SA’s; maintenance becomes increasingly difficult this way.

In the current situation at Zetes the collection of building blocks is presented to the user in the form of an alphabetically ordered list of building block names. The only mechanism that facilitates finding a certain building block is the ‘unwritten’ naming convention. With the exception of only a few (5 out of 220), all building block names start with an action, i.e. a verb, followed by an entity, i.e. a noun, to which the action is directed. Examples are `CheckArticle`, `FindLocationPerArticle`, `MergeIntoItem` and `RegisterIncomingPallet`.

This makes it relatively easy to find, using the alphabetical list, a building block if the user knows what type of action it performs. Easily finding a building block concerning a certain entity, however, is not possible. Neither is it possible, in the modeling environment, to find a building block that provides, requires or unsets a certain active registry setting. This is useful when, during the modeling of transaction types, a building block requires active registry setting A, but the user does not know which building blocks provide active registry setting A. Currently, he is forced to search through existing transaction types or the documentation files. In short, the current modeling environment provides insufficient ways to locate a wanted building block.
Especially in the light of the previous section, where it has been decided to decrease the granularity of building blocks where possible, which will undoubtedly lead to an increase of building blocks in the collection, a mechanism should be added that facilitates the search for building blocks.

### 3.2.1 Classification

A logical choice would be to classify building blocks based on the type of process they are involved in, e.g. picking, receiving or ordering of goods. Based on the type of process a user is modeling, the environment could then provide the user with relevant building blocks.

Essential is that this list of provided building blocks is of such size that the right building block is present and can be easily found. If the list becomes too large, the user is still required to search through this list and the desired effect will be lost. This means that, as long as the right building block is present, the list of building blocks should be as small as possible.

A problem with this approach is in the nature of the building blocks; they are developed with reusability in mind and are made to be broadly applicable. As a result, a single building block can, on average, be used in a large number of process types and each process type makes use of a large number of building blocks. This leads to the classification becoming diluted and, as the number of building blocks in the collection grows, even useless. This is worsened by the guidelines, aimed at reusability, given in Chapter 4. This makes using classification a poor choice to support the localization of building blocks.

### 3.2.2 Search function

Classifications that are already implicitly present in the collection are those based on the action the building block performs, e.g. scan, register or check, and on the entity to which the action is directed, e.g. article, location or stock. This classification comes forth from the way building blocks are named. Actions are grouped together, because of the alphabetical listing, but entities cannot, at the moment, be displayed in a grouped fashion. A sensible way to achieve this is by introducing a search function. After interviews with two System Architects\(^3\) it was concluded that a search function would be a logical and useful choice to implement in the modeling environment. Such a search function should at least be able to search on building block names, but ideally more building block characteristics should be searchable. The building block’s process type is an option, but this presents difficulties as explained in the previous section. Searching for a building block that provides or requires a certain active registry setting would be useful and this is certainly possible if the relevant information is stored in a suitable way. An in-depth look into this mechanism and which building block characteristics shall become searchable is presented in Section 5.1.

---

\(^3\) P. Kock and R. Sengers; the question list of the interview can be found in Appendix D: Interview Scheme 2. The results of the interview can be found in Appendix E: Interview Scheme 2 – Interview results
3.2.3 Recommender System

Different customers of Zetes make use of a number of the same, or at least similar, transaction types. This is because Zetes delivers a standard platform configured to specific customer needs, but not a custom solution per customer. For different customers therefore similar transaction types are modeled, related to the different process types that occur in many of Zetes’ customers, e.g. incoming orders, outgoing orders and replenishment. This prompted the idea to include a system to facilitate the modeling of frequently occurring transaction types. Our first look is on a recommender system.

The idea is that based on existing transaction types, it can be derived which building blocks are most often used in conjunction with each other. Building blocks that have often been used together in existing transaction types are likely to be used together in future transaction types as well. In the modeling environment the possibility could be added to select, of the building blocks that already have been added to the transaction type, one or multiple building blocks. It is then determined which building blocks are used most often in combination with the selected building blocks; those are then presented to the user. Thus, with each building block added to the transaction type, the recommendation would become more specific.

To achieve such a recommender system it is necessary to analyze the information of existing transaction types and in particular the frequency with which a building block occurs in a transaction type with any other building block. This should then be stored in an easy accessible way; for example in an adjacency matrix. Using this matrix, it can then be easily looked up which building blocks are used most often in combination with which other building blocks.

After further thought and discussions with several SA’s it became clear however that this system has serious drawbacks that make this recommender system unsuitable. The first problem concerns the need to have the information on transaction types from all previous customers. This information, at the moment, is not centrally available. The existing transaction types and the information on them is stored on servers of each individual customer. Zetes, at the moment, does not centrally store the actual transaction types of each customer; only the documentation. Thus it is not a trivial task to collect all existing transaction types, which is a necessity for the recommender to work properly.

Secondly, if a transaction type needs to be modeled for a new type of process, the recommender system provides no help to the user. This is because the recommender bases its recommendations on existing transaction types in the database and a transaction type for a new process type is per definition not yet in the database. The user would even have to model the new transaction type possibly dozens of times before any recommendation towards the new transaction type will emerge. This is caused by the fact that building blocks used in the new transaction type are most likely already used many times with many other building blocks. In other words, the recommender would take significant time to adapt to new process types.

Thirdly, there exist building blocks in the collection that are used in a very large percentage of all transaction types, e.g. the building block to set the default label printer. These building blocks are thus often used with almost any other building block. As a result of this they will always occupy the top spots of the recommendations. As long as this would only be the case for two or three building blocks it would not present such a problem. If all recommendations however continuously consist of the same set of often used building blocks, the recommender loses its usefulness.
Measure could be taken to counteract the last two phenomena. Such as the option to add extra weight to certain transaction types and filtering of often used building blocks respectively. However, this also creates new problems and adds more complexity. A new idea was thought of that achieves the same goal as the recommender system, facilitating the modeling of frequently occurring transaction types, in a better and simpler way; templates.

### 3.2.4 Templates

Templates are, like the recommender system, a way to speed up the modeling of frequently occurring transaction types. Meant here are templates for transaction types, a standard modeling for a transaction type. It consist of building blocks, their arrangement and the choice of variant codes. By opening a template in the modeling environment the user is presented with a complete transaction type which can be freely configured and adapted to make it suitable for a specific customer. This means the user has the option to remove or add building blocks, alter their arrangement and change variant codes as he sees fit, although if the template is correctly chosen large modifications should not be necessary.

Because transaction types for a certain process type are largely the same, at least one template should be designed for each process type and for other frequently recurring transaction types as well. In this way, they provide a useful starting point for users. Templates allow a user to skip the modeling of the recurring part of a transaction type and to only focus on the customer specific configuration. This approach saves time and effort.

Comparing templates to the recommender system, they have several advantages. First off, it does not necessarily require the central database containing all existing transaction types like the recommender does. Although existing transaction types could be used as starting points for the modeling of transaction types as well; more on this in the detailed view on templates in Section 5.2.

Secondly, in the case of a new process type, a new template for this process type can be created within several hours depending on its complexity, which is then immediately ready to use; contrary to the recommender system which needs a significant adaption time for new process types.

Finally, this approach is less complex than the recommender, which involves analyzing transaction types, storing the data in a convenient way and using this to compute recommendations. For templates no new implementations are needed; the modeling environment’s representation for transaction types can be used as a basis to store the templates. These advantages cause the template approach to be chosen over the recommender system. A detailed look on this approach can be found in Section 5.2.

N.B.: This part of the literature study, the localization of building blocks, contains no references to literature. There are several reasons for that; generally, the explored concepts are either well known beforehand or either customized for Zetes, such that relevant literature is not available.

The concept of classification, to begin with, is known well enough to make an initial assessment of the possibilities of classification at Zetes. After this assessment and subsequent discussions with SA’s it was clear from an early stage that classification was unlikely to be a successful approach for the localization of building blocks and that, as such, a different method needed to be found. Additional literature study was not needed to reach this conclusion.
Search functions as well are a familiar area. Enough is known on them, their possibilities, benefits and limitations to decide in favor of using them. Literature on search functions is well available, but mostly focuses on the general concepts, which needs no explanation, or focuses on implementation details, which are not addressed in this chapter.

The recommender system that has been proposed is a custom recommender specific to the situation at Zetes. It cannot be compared to a recommender following a content-based filtering or collaborative filtering approach, which are common in the area of recommenders. These approaches, although common, were not found to be suitable and corresponding literature was not applicable.

The last discussed mechanism, templates, is a very basic mechanism that should require no extra explanation. Through discussions with SA’s, it was decided that it would provide sufficient benefit to include this mechanism.

Although references are missing on the subject of localization, it has been attempted to provide sufficient arguments and comparisons to support the choice for the search function and template mechanisms.

### 3.3 Development Environment

Next let’s take a look at problems 3 and 4:

3. Building block requirements/provisions are not visible in the modeling environment
4. Variant information is not visible in the modeling environment

Both are cases of information that is not provided to the user by the modeling environment. They represent a similar type of problem and shall therefore be handled together. Since the limitation lies in the modeling environment, that shall be the starting point to be explored.

The modeling environment at Zetes is very limited and only supports the minimum of functionality needed to model a transaction type. It is therefore more often called a configuration tool; it has the ability to configure building blocks and their order, but nothing more. In order to determine if having a more complete modeling environment that more readily supports the users, the concepts that form a modeling (or more often called a development) environment shall be explored first.

Let’s start by looking at the definition of a development environment.

> “In computer program and software product development, the development environment is the set of processes and programming tools used to create the program or software product.”[4]

Applied to Zetes, the modeling environment thus encompasses all tools used during the modeling of a transaction type, i.e. at least the PC-app and the documentation files, but the Windows search function is also used extensively to search through the documentation and as such could be included as well. Although it is a development environment, it is a poor one, with all tools in separate places. A generally accepted better solution would be an Integrated Development Environment (IDE).
“An integrated development environment is one in which the processes and tools are coordinated to provide developers an orderly interface to and convenient view of the development process (or at least the processes of writing code, testing it, and packaging it for use).” [4]

IDEs are common in the field of computer programming where they provide the user with a single point for writing code, testing it and packaging it for use. For Zetes, the information on building blocks, now stored in individual documentation files, should be integrated in the development environment, as well as a search function (see Section 3.2) and a way for testing the validity of transaction types (see Section 3.4). To make this first item possible, building block information should be accessible for the development environment.

Furthermore, the data should be centrally accessible, data integrity should be ensured and it should be possible to define relationships, because of the possible influence of the variant codes of a building block on the active registry settings it provides, requires or unsets. The logical choice is to make use of a relational database, which is the most widely used choice for data storage. It ensures data integrity, allows relationships and has mature systems for managing, querying, storing and retrieving of information.

A possible, explored alternative is to make use of XML, which however is unsuitable. Although XML can be used as a database it is not designed for this purpose, but primarily for moving data between enterprises. XML files, for example, cannot be searched or updated as efficiently as relational databases and have no built-in mechanism to prevent concurrent access. In the case of large amount of data, the use of XML becomes very slow [35].

Another very important argument for choosing a relational database over an XML database is the fact that relational databases are currently used by Zetes to store both transaction type and building block information. It is therefore beneficial to remain storing the building block information in a relational database. In this way, relationships between building blocks and transaction types can be easily made.

To make information on building block requirements, provisions and variants available and visible, a new database model for building blocks (and transaction types) shall be developed; see Chapter 7.

### 3.4 Transaction Type Representation

Problem number 5 concerns the Automated Dependency Checker (ADC), which is connected to several of the other problems at hand. To be able to perform ADC certain information on building blocks should be available; namely, the provided, required and unset active registry settings (and if they are optional or mandatory) and information on variant codes (see Section 3.3), because they have the ability to influence the provided and required active registry settings. Furthermore, the ADC should take the support for conditional paths and loops, i.e. problem number 7, into account; the ADC should ensure that successful execution of the transaction type is possible for every possible path through the transaction type. Having such an ADC, it can be assured during development time that no runtime errors shall occur related to missing active registry settings. This thus eliminates the problem of unclear error messages, i.e. problem number 6, as errors should no longer occur.
Another precondition to make ADC possible is a formal description of a transaction type: which building blocks are in this transaction type in which arrangement, what variant codes are used and which active registry settings are provided, required and unset by each building block. Based on this information can an analysis be made if each building block has access to the right active registry settings at the right time. The information should be available in such a way that the analysis can be easily performed, i.e. in some form of formal specification. Two concepts shall be explored that are possible solution directions, namely Workflow Patterns and Domain Specific Languages (DSL).

### 3.4.1 Workflow Patterns

Workflow patterns are patterns describing the control-flow perspective of workflow systems, generally used for business process modeling. The workflow patterns were designed to be able to objectively determine the expressive power and the range of concepts that workflow systems are able to capture. Van der Aalst et. al. identify 43 patterns related to control-flow, resources, data and exceptions handling [27, 28]. The patterns describe the arrangement of general tasks, which can be substituted for real-life business tasks, to achieve a certain intent and function. As an example, the most basic control-flow patterns consist of the Sequence, Parallel Split, Synchronization, Exclusive Choice and Simple Merge, which should sound familiar to people experienced with process modeling.

In general, the patterns can be compared to the composition of building blocks. Building blocks can be seen as small (automated) business tasks that can be arranged to achieve a certain function, i.e. a transaction type. In this way they are similar to workflow patterns, especially after the future addition of conditional paths and loops. A specific subset of patterns is applicable to transaction types. Patterns that need to be supported by transaction types in the new situation are:

- Sequence
- Exclusive Choice
- Simple Merge
- Structured Loop
- Cancel Task
- Explicit Termination
- Transient Trigger

The specifics for each of these patterns can be found at [http://www.workflowpatterns.com](http://www.workflowpatterns.com), although most of the names should be descriptive enough to understand their functionality. The choice for the Sequence pattern speaks for itself. The Exclusive Choice and Simple Merge are the constructs that make conditional paths possible. For loops, the Structured Loop, i.e. a loop with a single entry and exit point combined with a pre or post-condition, is sufficient for our needs. Arbitrary Loops, i.e. loops with possible multiple entry and exit points, hugely increases the complexity, but the functionality is hardly ever required during the modeling of transaction types. Manual cancellation of a transaction type is always possible, so a Cancel Task construct is necessary. A Cancel Case is equal to the Cancel Task in our situation as only one task is active at any time, due to the non-existence of constructs related to parallelism in our selection of patterns. The termination of a transaction type can be indicated by an Explicit Termination construct. An Implicit Termination would be a possibility as well, but making termination explicit provides more clarity without limiting expressiveness. On
specified points in a transaction type the user can make use of hotkeys to activate certain building blocks; the Transient Trigger pattern provides this functionality.

Workflow patterns are very useful to describe the possible behavior of processes. They, however, don’t mandate a specific implementation approach [27]. Therefore, the patterns itself do not provide a usable solution for the modeling of transaction types. For this we have to look at languages that define (a subset of) the workflow patterns, often coined (Business) Process Modeling Languages. Examples include BPEL, BPMN, XPDL and UML. Specifically, the languages should support the patterns selected above, but since those are some of the most basic patterns that is usually the case.

These languages are in turn implemented by process modeling tools or workflow management systems that often allow for drag-and-drop of components, e.g. tasks, transitions or states, on a canvas to model processes. This is similar to what is needed for the modeling of transaction types.

An important point, though, is that workflow systems allow you to model a process using standard components after which the model can be executed. For Zetes, however an environment is needed in which custom components, i.e. building blocks with specific attributes like their active registry settings, can be arranged to model a transaction type. This environment needs to be able to execute code during development-time to be able to perform Automated Dependency Checking and it should be able to edit attributes of components, i.e. variants of building blocks. The environment does not need to be able to create an executable client or process; only a representation of the transaction type has to be stored in the database, which can be accessed by software on the RF-terminals that is responsible for the execution of the transaction type.

This displays an important difference between what is needed and what workflow systems provide. A way to model custom transaction types is what is wanted; not a tool that provides executable processes. It has to be noted though that numerous tools exist for workflow management; over ten years ago several hundred of these sort of products were released in several years [29]. Therefore, it is possible that a certain tool provides the necessary functionality, but none of the ones examined by me achieved this in an easy way. The described modeling languages do, however, provide inspiration for the possible representation of transaction types, but almost certainly in a customized way.

3.4.2 Domain Specific Languages

For this reason the attention shifted to Domain Specific Languages and the development of DSLs.

A DSL is as the name suggests a language specifically suited for a limited, target domain. Possible definitions include:

“*A computer programming language of limited expressiveness focused on a particular domain.*” [19]

“*A programming or executable specification language that offers, through appropriate notations and abstractions, expressive power on, and usually restricted to, a particular programming domain.*” [20]

“*A domain-specific language (DSL) is a high-level software implementation language that supports concepts and abstractions that are related to a particular (application) domain.*” [23]
The language meant here is a collection of sentences in a textual or visual notation with a formally defined syntax and semantics. A grammar or meta-model (in the case of visual notations) is used to define the structure of the sentences.

DSLs, compared to General Purpose Languages (GPL), are a lot less general, but are in turn more expressive within a limited domain. By providing the ability to use notations and constructions tailored toward a particular domain, substantial benefits in terms of expressiveness and ease-of-use are gained. Solutions to domain problems can be more easily expressed this way than would be possible in GPLs. This leads to increased productivity and reduced costs. Also, because of the higher abstraction level, less programming expertise is needed, which in combination with the familiar domain notation allows a larger group of people with less programming knowledge, i.e. domain experts, to be effective using DSLs [21, 22].

In the second of the three definitions for DSLs above, DSLs are defined to be executable; although according to several sources this is not a strict requirement. In [24] it is emphasized that DSLs need not be executable, but merely need to express something about a solution to a problem. An executability scale for DSLs is given in [22] consisting of a DSL with well-defined execution semantics, an input language for an application generator, non-executable DSLs that aid in application generation and non-executable DSLs. For Zetes a non-executable DSL would be appropriate that by providing an error-free transaction type representation aids in the generation of an application; the application here would be the transaction type stored in the database that can be executed by software on the RF-terminals.

Well known examples of DSLs are HTML (webpage markup), Excel (spreadsheets), SQL (database queries), Latex (text processing) and BNF (context free grammars), but many more exist. Other DSLs have been introduced as early as 1957. At that time a language for programming numerically controlled machine tools called APT was developed at MIT. This was then known as a ‘little language’. Other namings for DSLs that have come up over time are application-oriented, special purpose, specialized, task specific or application languages [22]. These can all be compared to DSLs and most of these terms have currently disappeared in favor of the term Domain Specific Language.

DSLs can be divided into textual and graphical languages, where a graphical language is also known as a Domain Specific Visual Language (DSVL) or a Domain Specific Modeling Language (DSML). DSVLs involve the systematic use of graphical constructs and notations to represent processes or other facets of a system, which is what is wanted for the modeling of transaction types. It is also possible to have hybrid languages that make use of both textual and graphical DSLs or DSLs that support multiple views, textual and/or graphical, on the same representation of information.

Given that an appropriate DSL can provide the right functionality for the situation at Zetes, a literature-based overview on the advantages and disadvantages of DSLs is given below to be able to more accurately determine if a DSL would be a suitable way of representing transaction types.

**Advantages**
- Because a DSL uses appropriate domain-specific notations a **higher productivity and maintainability** is reached than can be achieved by GPLs [21,23,24]. Other quality aspects
such as reliability, portability, flexibility and reusability are also improved [25,26], mainly due to the expressiveness and abstraction of DSLs.

- Using DSLs, domain experts can more easily understand, validate, modify or develop programs. This is because the DSL allows solutions to be expressed in terms of domain idioms. This also improves readability and writability for domain experts [21,24,25].
- The use of DSLs has an advantage in long-term costs compared to conventional methodologies; the higher initial development cost is offset against the reduced engineering and reengineering costs once the DSL is in use [21,26].
- DSLs are developed with and thus contain domain knowledge, which enables the conservation and reuse of domain knowledge [25,26].
- A DSL offers better possibilities for analysis, validation and optimization than would be able with GPLs; code patterns used by GPLs are too complex or not well-defined for these goals [21,22,25]. Furthermore, if the language constructs are safe, any sentence constructed using the language is safe.

**Disadvantages**

- There are costs for the design, implementation and maintenance of the DSL. The development is hard and requires a high amount of effort, brought on by the need the have expertise in both language design and the domain in question [21,22,25].
- Newly developed DSLs often lack support tools, which deliver syntax highlighting and checking, integrated compilers and advanced debuggers [21].
- Developing a new DSL implies having to educate users in the use of this DSL. Depending on the user community this can become a costly and time consuming task [21,22,26]. Having a DSL that closely matches the domain expert’s expectations mitigates this effect.
- DSLs often have lower performance compared to hand-coded software. The importance of this disadvantage depends on the performance requirements of the DSL [21,25]. In rare cases the performance is actually increased, because optimization is possible on a higher abstraction level.

The most important advantages for Zetes are the increased possibilities for analysis, because of the need to check building block dependencies, higher productivity and reduced long-term costs. From the list of disadvantages the development costs are the most important, but those are merely one-time costs that will lead to long-term cost savings. Due to the nature of the DSL the lack of support tools like syntax highlighters and compilers are of less importance. Performance is not of interest either and because of the limited size of the domain, educating users should not require a significant effort. This makes a DSL to represent transaction types a very suitable choice. Chapter 6 describes the development of the DSL.

### 3.5 Overview

The table below summarizes which solution directions have been chosen for each problem and the chapter in which these solutions are further discussed.
The database model design takes place after the design of the Domain Specific Language. The reason for this is that the new database model should incorporate all the information that is required for the Domain Specific Language.

As can be seen from the table, problem 6 and 7 have not been explicitly handled in this chapter. Problem 6 is the lack of useful information given by error messages if an active registry setting is missing, i.e. it is not made clear which setting is missing. This is a problem that is of a pure technical nature that requires changes to the source-code of either the individual building blocks or the transaction type framework. Solving this is outside the scope of this thesis. Furthermore, runtime errors caused by missing active registry settings should be eliminated as a result of the Automated Dependency Checker.

Problem 7 is the fact that it is not possible to model conditional paths and loops in a transaction type, only within the building blocks. This problem shall be solved by making sure the DSL that is to be developed supports the option of conditional paths and loops, that it is made possible to model these concepts in the proof-of-concept and that the to-be-designed database model supports these concepts as well. In other words, transaction types including conditional paths and loops can be modeled and checked in the proof-of-concept. However, the execution of transaction types on the RF-terminals containing these concepts requires more work, namely extensive code alterations to the framework that is responsible for their execution. This is also out of the scope of this thesis.

The following chapters provide an in-depth look into the solution directions that have emerged from the literature study. They consist, except for the DSL development, of first a detailed explanation of the concept, followed by the technical details pertaining their implementation. The development of the DSL is too complex and extensive to fit such a scheme and shall therefore be discussed according to several development phases, namely Decision, Analysis, Design and Implementation.
4. Granularity Guidelines

This chapter describes the granularity guidelines for the building blocks. Firstly, in the concept section it is discussed which quality attributes should be prioritized. Secondly, in the implementation section the guidelines are explained that are aimed to achieve those quality attributes.

4.1 Concept

The quality attributes explored in the section on granularity (see Section 3.1) cannot reach optimization at the same time. For example, coarse-grained components are less reusable and maintainable, but are easier to assemble. Similarly, components with high customization capability are not always easy to assemble. Because of this conflict, it is necessary to decide which quality attributes to prioritize.

For Zetes it is important to be able to model as much logistical processes, in the form of transaction types, as possible with a minimal amount of effort. The Medea platform is marketed as being a highly flexible solution able to solve most logistical problems with very little actual programming. Flexibility (i.e. high Customization), Reusability and Ease of Assembly are thus the quality attributes of choice for building blocks.

By looking at the relation between these three quality attributes and granularity, in Table 3.1, it can be seen that there is a conflict of interest here; having more fine-grained building blocks is beneficial to the customization and reusability attributes, but has a negative impact on the ease of assembly. The ease of assembly is decreased because of the need to compose more, finer-grained building blocks to reach the same level of functionality, which also means having cope with more dependencies. Both of these issues however have only limited impact.

After examination of practical business case, the average number of building blocks in a single transaction type has been found to be close to 8, with a maximum of 20. These numbers would indeed most likely increase if building blocks were to be made more fine-grained. But even if every building block now, would require two building blocks in the new situation, then the average number of building blocks would only be 16. This is still an acceptably small number of building blocks for modeling a transaction type, especially considering that manual dependency checking will no longer

---

4 Which is an unlikely situation, because a substantial amount of building blocks is not suitable to be split up in parts.
be necessary. Furthermore, the problem could be reduced by introducing container functionality for building blocks; i.e. the ability to contain a number of small building blocks so as to appear as a single large building block with a fixed interface and functionality. A container can thus, on one hand, be seen as a transaction type, by containing building blocks arranged in a certain way, and on the other hand as a building block, by being used within a transaction type and having provided, required and unset active registry settings. This would conserve the flexibility of still having the fine-grained building blocks, but also increase the ease of assembly by having coarse-grained containers.

The issue of the increased need for dependency checking will be solved by the introduction of automated dependency checking.

Since the ease of assembly would only be slightly impacted, it is clear that fine-grained building blocks are desirable. This also corresponds to answers given during the interviews that indicated that building blocks are too big and contain too much functionality limiting flexibility.

The increase in the number of building blocks in the collection, as a consequence of having more fine-grained building blocks, also has a positive impact on customization and reusability and is thus even more reason to choose for fine-grained building blocks. Having more building blocks does have a negative impact on cost effectiveness and maintainability, but these are the attributes with less priority and the negative impact is most likely offset against the positive impact on those quality attributes by reducing building block size.

It is also likely that by reducing the size of building blocks their complexity and coupling are reduced as well. This has merely positive effects as can be seen in Table 3.1.

Now that the choice to reduce building block granularity is made, it has to be decided how to achieve this. Below five guidelines are presented for the decomposition of building blocks.

### 4.2 Implementation

For each guideline an explanation is given to clarify why and how this guideline is useful, followed by an example of how this guideline can be applied at Zetes. The examples serve to further clarify the given explanation and to demonstrate that the guideline in question is indeed applicable.

1. **Construct separate building blocks for multi-choice variants**

**Explanation**

In the Zetes building block collection, building blocks exist that have a single functionality but applied to a number of different situations depending on the chosen variant code. It would be useful to separate such a building block into several building blocks by making a single building block for each of the variant codes.

In this case, reusability is not increased, because the functionality of each separate building block is still equal to the original. Maintainability, however, would be increased; changes that are only applicable in one of the situations can be applied to the corresponding building block instead of having to alter the generic building block. And in all cases the number of variants for the building blocks in
question are reduced, which helps understandability and comprehensibility of transaction types. These latter two characteristics are also improved by the ability to use specific and more expressive names for the building blocks, instead of the generic one as in the old situation.

One possible drawback of this approach occurs when the separated building blocks contain a lot of analogous, general code, i.e. common to all separated blocks. If changes are needed to this part of the code it would require editing (and testing) multiple building blocks in the new situation, instead of one in the old situation. Thus, maintainability would actually be reduced instead of increased.

If it can be estimated that, in the future, significant changes might be necessary to the general part of the code, this general part should be split off to a separate building block as well.

**Example**

Building block 2r99 is called ‘set local registry with item id’ and is used to set certain active registry settings using information stored in the registry as an item ID. Variant slot 1 can be filled with one of eight codes each of which uses the item ID to set a different active registry setting. Namely, two different types of locations, three different types of articles, an order reference, a supplier or a master item (the meaning of these settings is not of importance for this example). Variant slots 2 and 3 are only used if slot 1 indicates that an article will be set, otherwise these variant slots will be ignored.

This is a good example of a building block that could be split into several separate building blocks; either eight or five different depending if you would want to keep the location and article variants in the same building block. This would replace the generic building block name with one that would reflect which active registry setting is actually being set. Plus, the specific building blocks would no longer have any possibly unnecessary variant codes.

2. *Split up building blocks that contain control flow logic*

**Explanation**

Due to the current limitation of only being able to arrange building blocks in sequence, certain building blocks have incorporated control flow (i.e. conditional paths) into their implementation, which would normally be included in the transaction type model. This leads, first of all, to transaction types of which the logic is not comprehensible from the arrangement of the building blocks, because the internal control flow is not visible. Second of all, this leads to building blocks that contain more functionality than necessary. If, depending on some condition, one of two paths must be followed and executed, it is necessary for that building block to incorporate both paths.

![Figure 4.1: Decomposing control flow logic](image-url)
Because the formal language specification will introduce conditional paths, such building blocks can be split into multiple separate building blocks with the condition expressed in the transaction type instead of in the building block (see Figure 4.1). Obviously not every single if-statement needs to be split up in this way. An important question thus becomes ‘When to apply this decomposition?’. Blocks A, B and C are suitable to become separate building blocks if they fulfill independent business tasks. So, were block A or blocks B and C to be single statements, there is no reason for the decomposition, because a single statement can hardly be thought of as constituting a business task. Even if A, B and C were business tasks, they would also need to be independently deployable. In other words, if A, B and C can only be used in conjunction with each other they should be kept together in a single building block.

Another situation is if such a building block is already separated, but that parts B and C have additional conditions included to either execute the actual functionality or skip it completely. In this way they can be arranged in sequence. Viewing the transaction type model this construction looks very confusing, since the internal logic is not visible (see Figure 4.2). This construction can be decomposed in a similar way as can be seen in Figure 4.1.

![Figure 4.2: Candidate construction for decomposition](image)

These constructions are made necessary in the current situation because of the lack of conditional paths, but are obviously unwanted. By following this guideline the control flow logic of transaction types becomes more comprehensible and building blocks become smaller, possibly creating building blocks that can be reused more often. Through the separation the user obtains more options in the composition of functionality. In short, this guideline improves flexibility and reusability.

**Example**
The 2nd situation described above can be seen in building blocks 2r173 and 2r174, named ‘Transfer item get’ and ‘Transfer item put’ respectively. In transaction types these two building blocks are put in sequence to realize the transfer of items from the bulk location to the pick location. Both building blocks, at the start of their execution, check the value of the active registry setting ‘transfer task mode’ that indicates whether the user is in the get or put mode. Based on this, the functionality of one of the two building blocks is executed, while the other is skipped entirely. This construction should be removed from the building blocks and incorporated in the transaction type model in the form of conditional paths.

3. **Encapsulate variant functionality in separate building blocks where possible**

**Explanation**
Certain building blocks contain functionality that is executed or skipped based on a variant code; this is often the case for variable checks. Since this is based on variant codes, it is known at design-time if
it is necessary to execute these functionalities in question. By transferring these parts to separate building blocks, the designer can choose whether or not to add (some of) those building blocks to the transaction type instead of setting the appropriate variant code. It should be noted that this works best if the functionality is located at the start or at the end of the building block, so as not to have to split the entire building block in parts.

This again improves the comprehensibility of transaction types as well as creating building blocks that can possible be reused in different situations.

Example
Using building block 2r125, an ‘orderref’ (i.e. an order reference) can be set in the registry, by typing in a value on the keyboard, scanning a barcode or reading a tag. Variant slot 3 and 4 are used to indicate whether this order reference should be used to set the active registry setting for the supplier and the customer respectively.

The functionality of these two variants can be transferred to two separate building blocks; for example, a building block ‘set supplier by orderref’ and ‘set customer by orderref’ that have the active registry setting orderref as a required setting. These building blocks can now still be used in combination with the original building block (although with two variant codes less) or in combination with other building blocks that set the orderref. They can now also be executed or not depending on runtime variables. And if users mostly require the supplier and customer to be set, as in the original case, this can be captured using the container functionality. All in all, users have the choice of more, smaller building blocks with more freedom to compose them; reusability and flexibility have increased.

4. Examine building blocks with a high cyclomatic complexity

Explanation
There is no direct relation between complexity and size of components or building blocks. A high complexity can be achieved by a small component and a much larger component could have a lot less complexity. In practice, however, the complexity of a component helps to estimate the components size [13].

This makes inspecting highly complex components worthwhile for reducing component granularity. Especially since reducing complexity on its own, according to Table 3.1, already has a positive impact on cost effectiveness and maintainability. The idea is that components with high complexity contain more functionality than components with less complexity and therefore might be suitable to be split up into smaller components.

It has been argued that the cyclomatic complexity should be kept at 10 or lower for a single module [14]. This guidelines has received corroborating evidence since then. However in some cases it has been found appropriate to allow for modules with a higher complexity number. A new guideline has been introduced as:

“For each module, either limit cyclomatic complexity to [the agreed-upon limit] or provide a written explanation of why the limit was exceeded.” [15]
In the case of building blocks it is harder to maintain a low cyclomatic complexity number. This is because building blocks are more difficult to decompose, since they cannot call methods or functions of other building blocks. For this reason, a suitable cyclomatic complexity number for Zetes likely lies above 10, although a precise, suitable figure would be difficult to ascertain. A tactic could be to start inspecting building blocks with the highest cyclomatic complexity and work downwards as long as the inspections provide enough ground to continue.

Example
A building block with a high cyclomatic complexity, although not guaranteed the highest, is 2r37, called ‘Split and merge item’. The aim of this building block is to split stock from an item and divide this over other items or locations.

The cyclomatic complexity of 2r37 is well over 50. This number, first of all, indicates that a large number of test cases will be necessary to achieve a thorough test coverage of this building block (consider that there are over 50 different paths through the source code). Second of all, this number could be an indication that the building block encapsulates too much functionality and should be split up.

Looking at the source-code and documentation it can be seen that this building block is a combination of two other ones, namely 2r03 ‘Register new item through splitting’ and 2r07 ‘Merge stock into item’. And the execution of this building block consists of 14 possible steps. Although it is clear that it is not possible to separate each of these steps into independent building blocks, it does give enough reason to take a detailed look on this building block to check the benefit and possibility of a decomposition.

5. Aim to capture overlapping functionality in separate building blocks

Explanation
This guideline is one of the principles of component-based design in general. By putting functionality that is required in more than one building block in a stand-alone building block, it can be reused which eliminates the need to write the same code multiple times. In the area of SOA it is stated that:

“In other words, the range of capabilities expressed by one service contract should not overlap with the capabilities expressed by others.” [18]

The same goes for the situation at Zetes. This guideline should always be kept in mind while developing building blocks.

If currently several building blocks contain the same functionality it should be examined if the overlapping functionality can be transferred to a separate building block. This leads without question to a finer granularity.

Example
No example has been found in the Zetes building block collection, but because of my limited knowledge of the collection this is no guarantee that this situation does not occur at all in the collection. It is furthermore a guideline that should always be taken into account while developing new building blocks.
These five guidelines can serve as a basis for the decomposition of building blocks to achieve a finer granularity. It is however not a case of the-more-fine-grained-the-better. One should be careful not to make the building blocks unnecessarily fine-grained. This would require additional assembly effort and finding the right building block from the collection would become more difficult. Furthermore, once specific building blocks, through decomposition, become dependent on each other and cannot be deployed independently, then they should not have been decomposed.

Instead, a balance has to be found. The goal is to make building blocks as fine-grained as possible without them losing a well-defined, independent business function.

The guidelines presented above are based on literature, interviews and personal insights. Literature is used to determine the influence of changes in granularity on quality attributes as described in Section 3.1 and provided the basis for guidelines 2, 4 and 5. The other guidelines are based on answers and ideas provided by experienced System Architects at Zetes during interviews combined with personal insights into what would provide benefit to the modeling of transaction types. This is what supports these particular guidelines. However, they have not been validated in practice and it can therefore not be guaranteed that they are sufficiently effective.

A thorough validation would require the analysis of a number of building blocks in order to determine, for each building block, if one or more of the guidelines are applicable. If so, the guideline should be applied and the building blocks should be edited accordingly. The modeling of transaction types with the new collection of building blocks can now be compared to the old situation. The new situation should primarily improve the flexibility and freedom of modeling transaction types. Elements that can be compared are the average number of requests for new or altered building blocks by the System Architects and the need for building block configuration through variant codes, both of which should be reduced. The opinion of users comparing e.g. flexibility, comprehensibility and reusability of building blocks between the old and the new situation can be used as validation as well. Such a validation would have to span enough time to model a sufficient amount of transaction types in order to properly evaluate the new situation.

This validation process is however not possible. First of all, because conditional paths are not yet possible and thus not all guidelines can be applied. Secondly, because analyzing and updating the building blocks and the corresponding validation would require more time than is available during this thesis. A simpler form of validation can be done by asking experienced SA’s for their thought on the estimated benefits of the guidelines.
5. Localization

The decision made in the Literature Study chapter to make use of a search function and templates to localize building blocks shall be further described in this chapter. For both the search function and the templates the general concept and their implementation details shall be given.

5.1 Search function

5.1.1 Concept

A search function requires little explanation; search term(s) are matched, first of all, against the filenames of building blocks. Several issues have to be taken in mind though.

Firstly, some building blocks have more functionality than can be captured in its filename. The filename consists of a single action on a single entity which presents problems when the building block executes multiple actions on multiple entities. Extending the filenames to reflect this is a possibility, but that could lead to very long incomprehensible filenames. Especially when building blocks still exist with multi-choice variants, as described in Chapter 4, it is difficult to choose a filename that represents the (possible) functionalities. It should be noted though, that these types of building blocks, following the granularity guidelines, should be split up if possible.

Secondly, natural language always gives rise to ambiguity; natural language contains different wordings with similar meaning. In the Zetes collection, this occurs for example with the actions set, enter and assign which are used alternatively. There is no clear rule which decides which of these three alternatives the developer should choose. This causes a user to have to use a multitude of search terms or queries to find the building block he is looking for. The same goes for finish and terminate and erase and remove.

In response to these problems, the option to attach tags to a building block shall be introduced. To be able to make automated dependency checking possible several building block characteristics will need to be stored in a database, e.g. active registry settings and variant codes (see the database model in Chapter 7). In addition to this information, for each building block, a number of tags can be stored in the database.

A tag is a single word that describes (part of) a building block. The power of tags lies in the fact that any number of tags can be attached and that the developer has complete freedom in choosing which tags to attach. These tags do not have to be made visible to the user and having a high number of tags thus does not become complex for the user as having long filenames would. The search function will have access to the tags in response to a user query; which is the way tags provide benefit in finding the right building block.
The first mentioned issue can now be solved using these tags. The developer can use any number of words, i.e. actions or entities, to describe the building block functionality. If the functionality involves multiple actions and/or entities each of these can be coupled to the building block. The second mentioned issue is now less of a problem as well, since tags allow you to cover for all (generally used) variations of words describing the functionality. For example, both the words *finish* and *terminate* can be used as a tag for a single building block.

Coming back to classification based on process types as discussed in the Literature Study chapter, this now becomes possible on a smaller scale through the use of tags. Even though it is not possible to classify each building block to a process type in a meaningful way, there do exist building blocks that are only useful for a single type of process. Such a type can now be tagged to these building blocks to obtain a small scale classification. Through these tags, the ability is created to link relevant building blocks. Especially for tags concerning process types this could be useful to quickly find building blocks that can be used in conjunction with each other.

It is important to establish guidelines for the naming of building blocks and use of tags. For both, the ambiguity brought on by natural language should be minimized. Furthermore, tags should be distinctive; if a single tag is attached to (nearly) all building blocks it no longer has any value in distinguishing building blocks. If, on the other hand, a building block has a tag that is used by no other building block, its distinctiveness is optimal, however that tag is unlikely to be used.

The complete list of guidelines concerning naming of building blocks and use of tags follows below:

1. Filenames have an *ActionEntity* format, that reflects the main functionality
2. In the tag list are included at least
   - All actions and entities involved in the building block functionality
   - Generally used variations for words above
   - The process types; granted that the building block can be attributed to a limited set of process types
3. All filenames and tags shall be in English
4. Filenames and tags should be as specific as possible
5. Tag specific rules:
   - Use lowercase letters unless the tag is a proper noun
   - Use only one word per tag
   - Use singular words instead of their plural form unless it is necessary
   - Use the bare infinitive form for verbs
   - Do not use common vague words such as: of, and, the, it, or, but, etc.
   - Avoid characters and punctuations such as dashes, commas, slashes, periods, etc.

For the tagging of building blocks according to their process type, it is necessary to create a commonly accepted list of different process types available. This is to ensure that building blocks belonging to the same process type have the same tag to represent that process type. Due to the author’s limited knowledge of the domain it is not possible to present an exhaustive list of all process types here. Such a list, though, could most likely be set up by an experienced System Architect within a day.

During the modeling of transaction types it is required to align building blocks in such a way that for each building block its required active registry settings are available at the time it is executed. To
make sure this is the case, the user occasionally has to look for a building block that sets a particular active registry setting. Currently, this information is only documented in individual Word files, unsuitable to be read by computers.

However, as mentioned above, in the new situation all provided, required and unset active registry settings for each individual building block shall be stored in a database. Now that this information is electronically stored in an accessible way, it can be used as well in combination with the search function. By searching for an active registry setting, those building blocks are returned that either provide or require that active registry setting. The results are grouped in that same way as well, with for both a sub-grouping into mandatory, optional and variant-dependent active registry settings. A third group consists of building blocks that unset that active registry setting.

Combining the three search areas discussed above, the idea is to create an integrated search function, that given a search term, searches in:

- Filenames
- Tags
- Active registry settings

Building blocks that contain the search term in one of these three areas are returned to the user, grouped by these same three areas. It is also possible to specifically search the filenames, tags or (mandatory, optional or variant-dependent) active registry settings. Such a search function encapsulates the most common ways to locate a building block; based on its name, its functionality (tags), its process type (tags) or which active registry settings are involved in the building block.

5.1.2 Implementation

This section shall in short discuss how the search function shall be implemented in the modeling environment. This will be a matter of writing suitable SQL queries.

All information that can be searched for, i.e. queried, is stored in a relational database. It is therefore a logical choice to use SQL to extract the required information, as it is one of the most widely used query language for relational databases. The exact SQL queries are dependent on the way the information is stored in the database. It is not known at this point how the database model shall look like, but it is a possibility that both the filenames and tags will each have a single cell for each building block; tags in this cell shall be separated by spaces. For active registry settings the situation is more complex, because of the different types that should be accounted for, i.e. mandatory, optional and variant-dependent. Exemplary SQL queries for filenames and tags could look like this:

1. SELECT name
2. FROM buildingblock
3. WHERE (name LIKE '%find1%')
4. AND (name LIKE '%find2%')

1. SELECT tags
2. FROM buildingblock
3. WHERE (tags LIKE '%find1%')
4. AND (tags LIKE '%find2%')
This is for a situation with two search terms that both need to match. Find1 and find2 are the search terms that have been parsed from the search input field. If matching either term is sufficient to return a result then the AND in line 4 should be replaced by an OR. For more than two search terms, the query can be extended with an additional AND or OR statement similar to line 4. For only one search term, line 4 should be removed. The query for active registry settings is a bit more complex due to the way they are stored, but the technique used is the same. The exact query, however, is dependent on the design of the database model for building blocks in Chapter 7.

5.2 Templates

5.2.1 Concept

The concept for templates is, as explained in the literature study, to have a default modeling for transaction types that are often needed; be it with slight variations. These templates speed up the process of modeling reoccurring transaction types by providing the common framework, which the user only needs to configure for a specific customer. In other words, the user can open a pre-built (default) transaction type for a certain process that needs to be modeled and the user can then adapt this transaction type as necessary.

Thus a template has to be designed by an SA for each process type and other often reoccurring transaction types. This template should aim to provide a modeling that is common to all transaction types for a certain process type as much as possible.

To extend the idea of templates it was suggested that it should become easier to reuse existing transaction types. An extensive collection of transaction types is available that have been designed for specific situations and variations of a process type. In the case of a new request for such a situation it should be possible to find and use the corresponding existing transaction type. This way, more emphasis is placed on reuse, saving time and effort, whilst having the reassurance that the existing transaction type has been used in practice and is thus tried and tested.

This type of reuse can be seen as (internal) opportunistic reuse, where existing solutions happen to be available for reuse. This is different from planned reuse, where components are strategically and intentionally designed for reuse in future projects [30], e.g. building blocks.

A condition for efficient reuse, as it is with building blocks, is that the needed transaction types can be found quickly and easily. For this reason it has been decided to use tags to identify transaction types. Like building blocks, it shall be possible to attach tags to transaction types. These tags can identify the process type for which it is made and the particular variation that is used. For example, one process type is the receiving of goods. These goods can either be for perishable or non-perishable goods. This affects the transaction type, because for perishable goods the expiration date has to be taken into account. Such a transaction type would at least have the tag receiving, identifying the process type, and the tag perishable, indicating the type of goods involved, i.e. the variation. More tags could obviously be added to further specify the transaction type. In this way, existing transaction types concerned with the receiving of perishable goods can be quickly found making it easier for a user to reuse transaction types.
The default templates would still be available as well; a tag *template* can be added to those to identify them.

Tags for transaction types should follow the same general guidelines as for building blocks (see Section 5.1.1). Specifically tag guidelines 3-5 are valid for transaction types, and included should be:

- In the tag list are included at least:
  - Process type
  - Variations of the process type

For variations a common list of available tags should be documented, as for process types, to prevent the use of many different, synonymous tags for the same variation.

### 5.2.2 Implementation

The transaction types, including their tags should be centrally stored and accessible for users. It makes, therefore, sense to include the tags in the database information on transaction types. In the current situation the table *transactiontype* (see Figure 2.5) should be extended with an extra field for tags; single-word tags separated by spaces.

However, to be able to model conditional paths and loops the database model for transaction types needs to be altered. These changes have to be coordinated with the changes to the database model for building blocks to create a consistent overall database model. Therefore, the precise way in which the tags are included in that database is unclear.

Tags can be searched for in the same manner as for tags of building blocks, namely through the use of SQL queries. The exact query is dependent on the database model that is to be designed, but shall be similar in structure to the query example found in Section 5.1.2.

The templates, i.e. the default models, can be added to the database just like a regular transaction type. To identify it as a template the tag *template* should be added to its tags, besides the regular tags to identify for which process type the template is meant.

Once a template or existing transaction type has been found that the user wants to edit, the modeling environment should be able to open the model. All the necessary information about the model is present in the database; which building blocks the transaction type consists of, their arrangement including conditional paths and loops, variant codes, etc. What is not included in the database is layout information, i.e. the positioning of the building blocks and other concepts on the canvas. By adding a *pretty printer* to the modeling environment, the model can be recreated from the transaction type information in the database.
6. DSL Development Process

In Section 3.4 it was decided, based on a literature study, to make use of a DSL to formally represent transaction types in order to be able to perform Automated Dependency Checking (ADC). As a guide for the development of this DSL has been chosen to use the paper “When and How to Develop Domain-Specific Languages” [22]. This paper divides the development of a DSL into the following phases: decision, analysis, design, implementation and deployment; a scheme similar to most regular software development processes. The division provides structure to the development process and it is, for the most part, well explained what is required in each phase.

To each phase, except deployment, classes of patterns are associated. These are aimed to support the developer during that particular phase. In the decision phase the patterns identify situations for which successful DSLs have been developed. The patterns in the phases analysis, design and implementation are approaches for respectively domain analysis, DSL design and DSL implementation.

During development the phases are generally not followed in a pure sequential way; subsequent phases may have influence on decisions taken in earlier phases with the need to reconsider those decisions. This has been the case during the development of this DSL as well, but in this paper only the final decisions and the reasons for their choice are documented.

In the remainder of this chapter the development phases decision, analysis, design and implementation are successively handled. The deployment phase is out of the scope of this thesis as only a proof-of-concept of shall be constructed.

6.1 Decision Phase

As stated in [22] the most important consideration for developing a DSL is the possible reduction in costs.

“The investment in DSL development (including deployment) has to pay for itself by more economical software development and/or maintenance later on.” [22]

At Zetes there are good indications that substantial cost savings can be achieved in the process of modeling transaction types, i.e. design, implementation, testing and deployment of transaction types. The increased flexibility, due to the support of the future DSL for conditional paths and loops, reduces the need to develop custom building blocks; saving time and money. During testing especially, cost savings can be achieved; the DSL shall make ADC possible which removes the need to test if the right active registry settings are available at the right time. Some tests shall always be necessary still; to ensure the transaction type displays the required functionality for example, but the total number of tests shall decrease. The economic factor therefore speaks in favor of developing a DSL.
The patterns for the decision phase described in [22] are concerned with improving software economics on one hand and with enabling of software development by users with less or no programming experience, but with domain knowledge on the other hand. Looking at these patterns, we can see several that are applicable to the modeling and checking of transaction types. The *AVOPT* pattern states that an appropriate DSL makes domain-specific analysis, verification, optimization, parallelization and transformation possible. From these, analysis and verification (of transaction types) is of importance and can provide value. Related to this is the *data structure traversal* pattern that states that a suitable DSL makes the traversal of complicated data structures better and more reliable. Such a traversal is most likely needed for the ADC to check the entire transaction type. The ADC can be seen as a form of task automation, namely the automation of checking building block dependencies. This is an example of the *task automation* pattern. The fact that several of the patterns described in the paper match the situation at Zetes indicates that a DSL could be a good approach that serves our needs.

The above paragraphs serve, in combination with Section 3.4.2, as arguments for the development of a DSL for the modeling of transaction types.

### 6.2 Analysis Phase

#### 6.2.1 Analysis Approach

The analysis phase is meant to gather domain knowledge and to establish the requirements and functionality, i.e. the scope, of the future DSL. This is done to ensure a consistent realization, by limiting the type and amount of information to be treated in the analysis, and to prevent on-going discussions during the DSL development concerning unclear requirements [31].

An analysis of the current situation, the existing systems and documentation in combination with talks to Software Architects of Zetes serve as the source for this information. Where SA’s were inconclusive about certain factors and decisions needed to be made, personal insight, literature and thoughts on what the most appropriate choice would be guided the decision.

In the paper “When and How to Develop Domain-Specific Languages” [22] the expected output of a domain analysis is described as a *Domain Model*, which is an explicit representation of the properties of the systems in a domain; which properties are *common*, which properties are *variable* and the *dependencies* between them [32]. It consists of the following components:

- Domain Definition
- Domain Terminology
- Domain Concepts
- Feature Model

This analysis shall deviate from this slightly. The Domain Definition and Domain Concepts follow below. Domain terminology, however, is not included here; relevant domain terminology has been made sufficiently clear in the preceding chapters, in particular in Chapter 2: Current Situation, and a complete terminology list for this thesis is included in Appendix A: Terminology.
The feature model is where the commonalities, variabilities and dependencies of the Domain Concepts are documented through the use of a feature diagram [33]. For this analysis the feature model has been replaced by domain assumptions that define the commonalities and variabilities and a UML meta-model to document the dependencies and relationships of the Domain Concepts. There are several reasons for this choice:

- The properties of the concepts can be better described using commonality/variability assumptions.
- UML is becoming the de facto standard for modeling purposes.
- An UML meta-model can be used as input for our modeling tool (see Section 6.4: Implementation Phase), which the feature model cannot.
- UML is adequately suited to describe our domain.

The commonality and variability assumptions, in combination with the meta-model, can be used to informally decide whether a system in within the scope of the domain.

### 6.2.2 Domain Definition

The domain in question is the modeling and checking of transaction types. It thus consists of two parts: modeling of transaction types on one hand and the checking of the transaction type on the other hand.

The modeling of transaction types is a process which involves the selection and arrangement of Building Blocks such that execution of those building blocks in the specified order achieves a predefined goal. Building blocks can be chosen from a collection of available, pre-constructed building blocks. Each one has five variant slots that can be filled with a variant code, i.e. a single letter or number. The codes that can be used are specified per building block and the codes indicate variations in the functionality of the building block.

At moments that building blocks prompt the user for input and are thus temporarily halted, *invokable functions* or *invokables* can be called. Which invokable functions can be called is specified for each building block separately. Either individual functions can be called or a sequence of functions, after execution of which the user is returned to the position from where they were called. These invokables can be and usually are building blocks out of the building block collection. As such, invokables also have provided, required and unset active registry settings.

Next to building blocks, *Exclusive Choices* and *Merges* can be included in the transaction type. These concepts allow the modeling of conditional paths and loops. Exclusive Choices have multiple exits with a condition attached to them. A single exit is chosen during execution of a transaction type depending on which condition evaluates to true. Alternative paths can be merged together through a Merge, i.e. a modeling object with multiple entries and a single exit. A loop may be represented via an Exclusive Choice exit-connection that returns to a Building Block or Exclusive Choice that was on a path that led to the connection. In the loop, only a single modeling object provides entry to the loop, but through the use of Exclusive Choices the loop can have multiple exits. This is a relaxation of the Structured Loop mentioned during the discussion on Workflow Patterns.
A difficulty associated with the Exclusive Choice is ensuring that precisely one exit-path is chosen from the available ones. For the exits of an Exclusive Choice there are three possibilities:

- None of the associated conditions evaluate to true.
- Exactly one of the associated conditions evaluate to true.
- More than one of the associated conditions evaluate to true.

The problem of having zero of the exit-conditions evaluate to true shall be solved by allowing users to define a default exit path; a path that shall be taken if all other exit-conditions evaluate to false. In case none of the conditions evaluate to true and the modeler has not defined a default exit, an exception shall be thrown. The problem of having more than one exit-condition that evaluates to true can be solved in two ways:

- Choose one of the valid exits at random.
- Assign an evaluation sequence to the exit paths which defines the order in which the exit-conditions shall be evaluated.

In our domain, the latter option is chosen. One of the goals of modeling in general is to make choices and their consequences explicit. Therefore making random choices is not desired. Users shall be given the option to define the evaluation sequence of Exclusive Choice exit-conditions, where the default exit-path always concludes the sequence.

The last two modeling objects in the domain are the Start State and End State, which naturally indicate the start and end of the transaction type. Only a single Start State can be added to any transaction type, for which a number of active registry settings can be defined that are available at the start of the transaction type. End States can be added to a transaction type a multiple places. Other than indicating the end of a transaction type they have no function.

All the modeling objects, i.e. Building Blocks, Exclusive Choices, Merges, the Start State and End States, are connected to each other through connections; in the modeling environment visualized by arrows between the relevant objects.

An automatic dependency checker is also part of the domain. This checker verifies that during runtime for every building block the necessary active registry settings have been set. To achieve this, for each building block it is known which active registry settings they provide, require and unset. Conditional paths and loops are taken into account by this checker. This way, the checker can ensure the availability of the right active registry settings at the right time.

Building blocks, Exclusive Choices and the checker can all make use of an underlying database to retrieve and store relevant information; this database is however not a part of the domain, only an asset to the domain.

### 6.2.3 Domain Concepts

Having defined the scope for the DSL, it is now possible to identify the concepts belonging to the domain. Below, an overview of the concepts, hierarchically structured, is given, in combination with a short description of their purpose.
<table>
<thead>
<tr>
<th>Concept</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction type</td>
<td>Defines the conceptual boundary of the system</td>
</tr>
<tr>
<td>Building block</td>
<td>Provides functionality for the transaction type</td>
</tr>
<tr>
<td>Active registry settings</td>
<td>Allows information exchange between building blocks</td>
</tr>
<tr>
<td>Requires</td>
<td>Indicates which ARSs are required by the building block</td>
</tr>
<tr>
<td>Provides</td>
<td>Indicates which ARSs are provided by the building block</td>
</tr>
<tr>
<td>Unsets</td>
<td>Indicates which ARSs are unset by the building block</td>
</tr>
<tr>
<td>Variant codes</td>
<td>Slightly alters the functionality of the building block</td>
</tr>
<tr>
<td>Invokables</td>
<td>Indicates which invokables are allowed for this building</td>
</tr>
<tr>
<td>Exclusive Choice</td>
<td>Provides conditional paths to the transaction type</td>
</tr>
<tr>
<td>Conditions</td>
<td>Controls the exit path to take from the Exclusive Choice</td>
</tr>
<tr>
<td>Default exit</td>
<td>Provides a valid path if all other conditions are invalid</td>
</tr>
<tr>
<td>Merge</td>
<td>Combines multiple paths, inverse Exclusive Choice</td>
</tr>
<tr>
<td>Loop</td>
<td>Provides loops for the transaction type</td>
</tr>
<tr>
<td>Condition</td>
<td>Controls whether to execute the loop</td>
</tr>
<tr>
<td>Start State</td>
<td>Indicates the starting point of the transaction type</td>
</tr>
<tr>
<td>End State</td>
<td>Indicates an end point of the transaction type</td>
</tr>
<tr>
<td>Connection</td>
<td>Connects different modeling objects</td>
</tr>
</tbody>
</table>

Table 6.1: Domain Concepts and their purposes

The concepts listed above are extracted from the Domain Definition. Together they constitute the objects and information that is needed to model and check transaction types. More specific information regarding these concepts can be found in the Domain Assumptions.

Note that the *Include in startup*, *Include in closing* and *End transaction* markings do not appear in either the Domain Definition or the Domain Concepts. That is because they shall not be a part of the DSL that is to be designed. These markings were the result of the way transaction types are now designed; with a fixed Scan Item at the start, a fixed database commit at the end and a continuous loop of the main part of the transaction type. Thanks to the addition of conditional paths and loops the modeler now has the freedom to start with any type of building block and perform database commits and construct loops anywhere in the transaction type. This makes those markings no longer necessary.

### 6.2.4 Domain Assumptions

Below are listed the Commonality and Variability Assumptions of the domain. They, respectively, describe what is common to the systems, i.e. transaction types, in the domain and in what way these systems can be varied and distinguished. Every assumption, next to its description, are accompanied with a justification that serves as a rationale why the assumption is valid [34].

**Commonality Assumptions**

- Every transaction type has a single Start State.

  **Justification:** Naturally a transaction type needs a single starting point. It would otherwise not be possible to decide where the transaction type starts.

- The Start State has zero entries and a single exit.
**Justification:** Logically the start has no incoming connections as it is the first point in the transaction type and there is no reason to revisit the starting point using loops as it provides no functionality. It would be unclear which exit to take if multiple ones could be chosen from, therefore only a single exit is allowed for the Start State.

- An End State has zero exits.
  **Justification:** Once the transaction type reaches an End State, the transaction type is exited. No more building blocks or other objects shall be executed.

- A transaction type is fully connected and all objects are reachable from the Start State (under influence of varying active registry settings, database values and user input).
  **Justification:** An assumption to ensure that no unreachable, and thus useless, objects are added to the transaction type.

- Every Exclusive Choice exit has a well formed condition\(^5\).
  **Justification:** To decide which exit of a Exclusive Choice to choose, it is necessary that each exit has a condition that can be evaluated.

- Every Merge has a single exit.
  **Justification:** Merges combine multiple alternative paths into a single path.

- Every unfilled or invalidly filled active variant slot of a building block is treated as having the default variant code.
  **Justification:** Building blocks need to know their variant codes to execute the right functionality. In the case of unfilled or invalidly filled active variant slots the default variant code is used to ensure that the transaction type can still be executed.

- Every loop has a single entry point.
  **Justification:** This simplification of loops is chosen, because it provides enough expressiveness for transaction types whilst making the checking of transaction types easier.

- Every connection starts and ends at a modeling object.
  **Justification:** It should not be possible to have a connection for which the two endpoints are not defined. Connections serve to indicate the flow between modeling objects. A connection with unspecified endpoints has no meaning.

- After calling an invokable function and execution of this and corresponding functions, control is returned to the building block from which the invocation was called.

\(^5\) What constitutes as a well-formed condition is not specified in this thesis. See Chapter 9: Future Work.
**Justification:** Calling an invokable function is a temporary interruption from the transaction type. The predictable behavior is to return control, after execution of the invokable function(s), to the building block where the transaction type was left off.

**Variability Assumptions**
- Every transaction type has a single End State.
  **Justification:** Alternative paths can be explored through the use of Exclusive Choices, which may lead to multiple points at which the transaction type ends.

- The number of exits of an Exclusive Choice may vary. The minimum number is two (including the default exit), the maximum number is unlimited.
  **Justification:** Exclusive Choices directs the flow to one of several alternatives. Therefore at least two exits are required. There is no need to limit the number of exits.

- The number of entries to an Exclusive Choice may vary. The minimum number is one, the maximum number is unlimited.
  **Justification:** Without an entry the Exclusive Choice is unreachable. Having more entries is possible when connections loop back to the Exclusive Choice. It cannot be used to merge different alternative paths though.

- The user is free to define the evaluation sequence for the Exclusive Choice exit-conditions.
  **Justification:** The order in which the exit-conditions are evaluated can be of influence on the functionality that the transaction type displays. It is therefore important to allow the user to define the evaluation sequence. Note: the evaluation sequence is always concluded by the default exit, if one is defined.

- The number of entries of a Merge may vary. The minimum number is two, the maximum number is unlimited.
  **Justification:** Merges combine multiple alternative paths into a single path. There is no need to limit the number of entries.

- For every variant slot the variant code may be chosen from the predefined list of valid variant codes for that specific variant slot.
  **Justification:** Variant codes allow the modeler to slightly alter the functionality of the building block. For each combination of variant slot and building block a limited number of valid codes are defined.

- The number of invokables functions for a building block may vary. The minimum is zero, the maximum is unlimited.
  **Justification:** It is possible to invoke certain functions at prompts during the execution of a transaction type. If a building block has no prompt there is no moment on which to call these functions and therefore invokables should not be defined for these building blocks. There is no need to limit the number of available invokables functions.
• The user can add any number of Building Blocks chosen from the collection of available building blocks.
  
  **Justification:** The transaction type can be of any size.

• The user can add any number of Exclusive Choices.
  
  **Justification:** The transaction type can be of any size.

• The user can add any number of Merges.
  
  **Justification:** The transaction type can be of any size.

• The user can add any number of End States.
  
  **Justification:** Due to the fact that alternative paths can be taken during the course of executing a transaction type, it is also possible to have different points at which the transaction type should terminate. There is no need to limit the number of End States.

• The number of entries of an End State may vary. The minimum number is one, the maximum number is unlimited.
  
  **Justification:** An End State should have at least one entry or it is unreachable. Multiple entries are allowed so that multiple alternative paths can end in the same End State; multiple End States could be defined as well, it is up to the user to decide. There is no need to limit the number of entries.

• The user has the freedom to arrange the modeling objects in any way that is consistent with the commonality and variability assumptions.
  
  **Justification:** The ability to arrange the modeling objects in an arbitrary way is what makes it possible to use building blocks to achieve a variety of goals.

### 6.2.5 Meta-model

The meta-model is one the most important parts of a Domain Analysis as it is needed to solve several challenges, of which the ones applicable to our situation are [31]:

• Construction of Domain Specific Language: the abstract syntax of the language is described by a meta-model.

• Model Validation: the meta-model defines constraints against which models can be validated.

• Tool Integration: using the meta-model, modeling tools that are specific for the domain can be easily developed.

A meta-model is a model that describes frames, rules, constraints and models for modeling a pre-defined class of problems; in our case transaction types. More precisely, it defines the constructs of a modeling language and the way in which these can be related, through the use of constraints and modeling rules. Each model, i.e. transaction type, is an instance (or formal model) of the meta-model and adheres to the rules and constraints defined in the meta-model. A meta-model defines the so-called *abstract syntax* and *static semantics* of a modeling language.
The concrete syntax, i.e. the concrete form of the textual or graphical constructs used for the modeling, is based on the abstract syntax and should render the meta-model in an unambiguous way. The development of the concrete syntax and specification of its semantics is done separately from the abstract syntax at a later stage; see Section 6.3: Design Phase. An overview of these related concepts can be seen in Figure 6.1 (Original from [31]).

![Figure 6.1: DSL development overview](image)

A meta-model can be described using any arbitrary modeling language; examples include UML, SysML and feature models. The choice for a modeling language depends on the suitability of the language to describe the domain in question. As explained earlier, it has been decided to use UML rather than feature models, which was the modeling language of choice in the paper by Mernik et al. [22]. UML is a well known modeling language with many practically usable tools, which is well able to describe our domain. The fact that UML meta-models are often used as input for DSL development tools makes this a suitable choice for this project. Consequences of the choice for UML and whether this is also a suitable choice for the full product is discussed in Section 6.4.1.

The meta-model can be seen in Figure 6.2. Its elements and the structure are explained here; for a description of the attributes consult Section 6.3.3: Execution Semantics. At the top of the meta-model is the transaction type; the main concept. A transaction type consists of a collection of any number of objects and any number of connections, indicated by aggregation relationships. Using generalization

---

6 http://www.uml.org

7 http://www.sysml.org

8 Kyo C. Kang, et al., Feature Oriented Domain Analysis: Feasibility Study [33]
relationships an object is divided into Building Blocks, Exclusive Choices, Merges, Start States and End States. Each of these constructs can be related to each other, which is indicated by the association relationship between Object and Connection. An object can have multiple connections, i.e. outgoing and incoming connections, and each connection belongs to two objects; a source and a target object. A connection can be either a Regular connection or a Conditioned connection, where the latter is used for Exclusive Choice exits. These have additional attributes to specify the condition and sequence number.

![UML Domain model; Abstract Syntax](image)

Several constraints are associated with this meta-model, for instance those that specify the number of incoming and outgoing connections that are allowed for each type of object. These constraints are specified in the form of the Object Constraint Language (OCL) [36]; a declarative language for describing rules that apply to UML models.

**OCL Constraints; Static Semantics**

-- Building Block cardinalities

```ocl
context BuildingBlock inv:
self.incoming->size() >= 1 and
self.outgoing->size() = 1
```

-- Exclusive Choice cardinalities

```ocl
context ExclusiveChoice inv:
```
self.incoming->size() \geq 1 \text{ and } 
self.outgoing->size() \geq 2

-- Merge cardinalities
\textbf{context} Merge \textbf{inv}: 
self.incoming->size() \geq 2 \text{ and } 
self.outgoing->size() = 1

-- Start state cardinalities
\textbf{context} StartState \textbf{inv}: 
self.incoming->size() = 0 \text{ and } 
self.outgoing->size() = 1

-- End state cardinalities
\textbf{context} EndState \textbf{inv}: 
self.incoming->size() \geq 1 \text{ and } 
self.outgoing->size() = 0

-- Only one Start state allowed
\textbf{context} StartState \textbf{inv}: 
StartState.allInstances()->size() = 1

-- At least one End state required
\textbf{context} EndState \textbf{inv}: 
EndState.allInstances()->size() \geq 1

-- Start state exit cannot go to Merge
\textbf{context} StartState \textbf{inv}: 
not self.outgoing.target.oclIsTypeOf(Merge)

-- Start state exit cannot go to End State
\textbf{context} StartState \textbf{inv}: 
not self.outgoing.target.oclIsTypeOf(EndState)

-- Conditioned connections have an Exclusive Choice as source
\textbf{context} Conditioned \textbf{inv}: 
self.source.oclIsKindOf(ExclusiveChoice)

-- Regular connections do not have an Exclusive Choice as source
\textbf{context} Regular \textbf{inv}: 
not self.source.oclIsKindOf(ExclusiveChoice)

-- Objects cannot connect to themselves
\textbf{context} Connection \textbf{inv}: 
self.target <> self.source

-- No two connections from the same source to the same target
\textbf{context} Connection \textbf{inv}:
The above presented meta-model along with the rest of the Domain Analysis shall form the basis of the DSL that is to be designed.

### 6.3 Design Phase

#### 6.3.1 Design Approach

In this phase of the development the concrete syntax and execution semantics of the DSL shall be specified. This will complete the DSL development as follows from Figure 6.1.

The concrete syntax is based on the abstract syntax; it can be seen as a realization of the abstract syntax. Since a graphical modeling language is being developed, the concrete syntax shall be a graphical syntax. The concrete graphical syntax shall define how transaction types can be modeled in a graphical way. It defines the objects that can be added to the canvas, how they look and how the objects can be related using arrows. The use of a graphical notation is a goal for the newly to-be-designed modeling environment as it holds an important advantage over a textual one. Namely, graphical models are generally credited to provide a better overview and ease the understanding of models [38, Guideline 4]. They make the structure more visible and more clear to the user and are better at expressing relationships that cannot be easily extracted from textual representations [39]. On the other hand, in the paper ‘Why Looking Isn’t Always Seeing’ [40] many of the beneficial claims of graphical models are weakened and others indicate that text-based modeling is faster and has better tool support than graphical modeling [41]. A solution that combines the advantages of both is to have an interchangeable format that gives the user the ability to switch between graphics and text. Such a user-editable textual syntax is not included for this project. An extra concrete textual syntax for modeling transaction types could later be added to achieve the interchangeable format, but this is out of the scope of this thesis.

An overview of the process from modeling a transaction type to its execution is visualized in Figure 6.3. The information in parentheses indicates the technology that is used to realize the task at hand.

The user graphically models a transaction type making use of the graphical syntax. This syntax is implemented with the help of the modeling tool *Graphical Modeling Framework (GMF)*. Before the registry dependencies can be checked, it needs to be assured that the model represents a valid transaction type. Therefore, several levels of validation are introduced. During modeling, the first level of validation is performed. The tool real-time prevents the user from defining invalid connections between objects, according to the OCL constraints listed in the Analysis Phase. For example, the tool prevents the user from modeling a Start State or Building Block with multiple exits.

---

[9](http://www.eclipse.org/gmf)
At a user-initiated point in time the remaining OCL constraints are checked that are not checked in real-time. These, for example, include the remaining modeling object cardinalities and whether exactly one Start State has been added. If no invalidities are found, the 1st level of validation is finished and the 2nd level of validation can be performed. At this point validations that cannot be expressed in OCL are performed, e.g. if the model is fully connected and if the loops are properly defined. For this purpose, the graphical model is converted to an in-memory graph structure; a transaction type is basically a directed graph. The graph representation is then used to perform the validation on. There are several reasons for this intermediate step.

**The intermediate step allows the ADC to be independent of the modeling tool.**
The modeling tool that shall be chosen to implement the proof-of-concept is not necessarily the most suitable choice for the final, complete modeling environment. The proof-of-concept is meant to demonstrate the working of the DSL and ADC and is as such chosen only for this task. Therefore designing the ADC based on (the internal structure and representation of) the modeling tool would not benefit Zetes if a different modeling tool will be used for the final product. The conversion of the transaction type to a graph structure and basing the ADC on this, makes the ADC independent of the modeling tool. This way, the work on the ADC done during this thesis shall remain useful no matter what modeling tool is chosen in the future.

**The intermediate step allows the ADC to be independent of the database model.**
An analogous argument can be given with respect to the database model. The graph structure is (partly) filled using information from the database. If, for some reason, the database model requires changes, only the way in which the information is retrieved from the database needs to be altered accordingly. The graph structure and ADC, however, would require no changes as they function independent of the database model.
The graph structure is implemented with the help of a generic graph library, which in general not only supports the creation of a graph, but also provides added ways for traversal and validation. Functions to analyze the connectivity of the graph and the detection of cycles are examples of this. The graph structure of the proof-of-concept shall be implemented using the Java language, but generic graph libraries exist also for other well-known languages as C++ or C#. Using a different language would require modifications to the implementation, but the general approach remains the same.

Note that this intermediate step could also be realized using other techniques that provide the same benefits. An XML format was originally considered as a way to represent the transaction type, as certain sources indicated that XML is a good approach to represent process models [24, 42]. The most important reason to choose a graph structure over XML is that a graph remains closer to the transaction type. An XML format is a tree-based structure, whereas a transaction type is a directed graph. This caused the XML representation to be fundamentally different from the transaction type representation. A graph structure does not have this problem and this makes it easier to couple the transaction type with the graph representation.

Once a syntactically correct transaction type has been modeled by the user, the ADC is used to verify that the right active registry settings are available at the right time. For this, the ADC also makes use of the graph structure of the transaction type.

If all the building block dependencies have been satisfied and the user has finished modeling the transaction type, he can store the transaction type in the Zetes database. The graph is traversed and the necessary information, i.e. the objects, their arrangement, variant codes and the conditions, is stored in the database at the appropriate place.

Now the transaction type can be executed on the RF-terminals. This is done by launching the 2R01 building block on the terminal. The 2R01 building block is a container-like building block that is responsible for executing transaction types. When a transaction type is selected on the terminal, 2R01 retrieves the necessary information from the database and executes the building blocks in the right order. It is thus also responsible for evaluating the Exclusive Choice conditions and selecting the right execution path. To cope with the new situation of conditional paths and loops and the new database model, the current 2R01 will have to be redesigned. This part, however, is not within the scope of this thesis.

N.B.: If either one of the two validation steps or the ADC ascertains that the model is invalid, control is returned to the user so that the model can be altered. The corresponding arrows have been left out of the overview to prevent the image from becoming cluttered.

According to the paper ‘When and How to Develop Domain-Specific Languages’ [22] the DSL design can be characterized along two dimensions. On the one hand you can distinguish DSL design into formal and informal design and on the other hand there is the distinction between Language invention and Language exploitation. The latter distinction separates DSLs into ones that are designed from scratch and those that are based on an existing language respectively. In the case of Language exploitation three patterns of design are identified. The Piggyback pattern is the first, where domain-specific features are applied to part of an existing language. In other words, an existing language is partially reused. The second pattern is Specialization, where an existing language is restricted to form a domain-specific language. The third option is Extension, where as the name suggests an existing
The language is extended with new features that address domain-specific concepts. The DSL for Zetes shall make use of Specialization.

The graphical syntax shall be a Specialization of the modeling standard Business Process Modeling Notation (BPMN); a language that implements most of the Workflow Patterns described by Prof. dr. ir. W.M.P. van der Aalst [37], including the patterns we are interested in. Our graphical syntax shall be restricted to those Workflow Patterns that correspond to our domain. Reusing an existing DLS has the benefit of an easier implementation and is recommended [38, Guideline 6]. Since graphical syntaxes for transaction types at the moment not yet exist, it has been chosen to use a language that is close to the domain in question [38, Guideline 14]. Other languages in the process modeling domain could have been used to, but BPMN has been chosen as it is one the more well-known standards and the fact that it aims to provide an intuitive notation able to represent complex process semantics.

The graph structure for the representation of the transaction type is designed from scratch, although it is based on the structure of transaction types. They are similar where possible to ease the conversion from the one to the other.

The concrete syntax and execution semantics are described next, with which the DSL design is concluded.

### 6.3.2 Concrete Syntax

The graphical syntax has the following notation for the different modeling objects:

![Domain Object Notations](image)

**Figure 6.4: Domain Object Notations**

The way in which these objects can be combined has been determined in the Analysis Phase. Below can be seen how this corresponds to the graphical syntax. A graphical construct is given for the different modeling constructs accompanied by a short description.

#### 6.3.2.1 Sequence

![Sequence representation](image)

**Figure 6.5: Sequence representation**

The sequence is the basic construct for any transaction type. It connects two modeling objects in an unrestricted way. Flow is passed from the left object to the right object unconditionally. Sequence flows can occur between any two different objects with the following exceptions:
- Exits from an Exclusive Choice exist of conditional flows.
- No flow can exist from a Start State to a Merge.
- No flow can exist from a Start State to an End State.

### 6.3.2.2 Exclusive Choice

![Exclusive Choice Diagram](image)

Figure 6.6: Exclusive Choice representation

Exclusive Choices are the objects that allow for alternative paths to be chosen. They direct a single input to one of the possible exits based on their associated conditions. Exclusive Choices can occur anywhere in the transaction type. Any object can connect to an Exclusive Choice and the exits can connect to any object. Default exits are indicated as such by a diagonal slash at the beginning of the connection.

The alternative paths of an Exclusive Choice are not allowed to connect to each other directly; they can only converge at a Merge. Connections between different alternative paths are identified as *crosspaths* and are not allowed.

### 6.3.2.3 Merge

![Merge Diagram](image)

Figure 6.7: Merge representation

The Merge allows to combine several alternative paths into a single path. Any object, with the exception of the Start State can connect to a Merge and a Merge can connect to any object.
6.3.2.4 Loop

Loops are modeled using an Exclusive Choice with an exit that connects to a prior building block or Exclusive Choice. The only condition for any loop is that there is a single entry for the loop, i.e. it is a reducible loop. The figure above is an example of a valid loop. It could be extended by adding additional objects between the Building Block and the Exclusive Choice; as long as no new entries to the loop are created. Nested loops are thus, by this definition, allowed. The path from the Exclusive Choice to the Building Block, that defines the loop is identified as the loop path or loop edge.

The loop in Figure 6.9 is an example of an invalid loop. The first loop, consisting of the first two Building Blocks and the Exclusive Choice, has two entries. The loop can be entered at both the first and second Building Block due to the incoming exit of the last Exclusive Choice.

In the syntax no specific mention has been made of the container functionality. Container functionality is, as explained in Section 4.1, the ability to store a transaction type as being a building block, i.e. with the required and provided active registry settings for the transaction type as a whole. A precondition to store a transaction type as a building block is that it has a single Start State and a single End State. This container functionality enables modularity of the transaction type, which leads to managerial, flexible, comprehensible and understandable infrastructure [38, Guideline 25].

The reason that no specific mention has been made in the syntax of this container functionality is that the containers can be treated as being building blocks by the ADC. Therefore they are also treated as building blocks in the concrete syntax. When a transaction type that uses a container is stored in the database the container can be decomposed, by retrieving its information from the database. Then the transaction type can be stored in the normal way.

For the graphical syntax the semantics are only defined by the object and the connections between them; the layout of the model does not have any impact on the meaning of the model. The option to adjust the layout of the model is there to allow the user to structure the model and to simplify the
understandability for a human reader. In the conversion to the graph structure, however, this information is ignored [38, Guideline 24].

6.3.2.5 Example

To demonstrate the various modeling constructs and their graphical representation, an example is given below. The model is purely exemplary and does not represent an actual transaction type.

![Figure 6.10: Example Transaction Type](image)

The example shows a transaction type consisting of 1 Start State, 2 End States, 5 Building Blocks, 2 Exclusive Choices and 1 Merge, connected by arrows that indicate the possible paths through the transaction type. The underlined numbers indicate the sequence number of the exit paths, i.e. they indicate the order in which the exit conditions should be evaluated. The text along the Exclusive Choice exits are the conditions associated with that connection. In the case of the first Exclusive Choice the chosen path is based on user input; a press of either the ‘Y’ or ‘N’ key indicates the path. In the case of the second Exclusive Choice two expressions are defined that shall be evaluated at run-time to choose one of the paths. If both of the condition evaluate to false, the default exit shall be chosen.

6.3.3 Execution Semantics

This section defines the execution semantics of a transaction type model. The purpose of these execution semantics is to describe a clear and precise understanding of the operation of the elements and their attributes. It describes the information they represent and the way in which they can be composed.

The semantics are described informally (textually) using natural language. For each element, the execution semantics are provided through:

- A description of the operational semantics.
- A description of the attributes of the element.

The attributes that are meant are those that originate from the Abstract Syntax. After certain attributes the words Fixed or Read-only can be found between parentheses. Fixed attributes have their value set beforehand and cannot be altered. Read-only attributes cannot be altered directly by users, but can change under the influence of other attributes of the element. Attributes that are not characterized in either way are editable by the user.
6.3.3.1 Connection

A connection is used to indicate the flow through a transaction type; it indicates the order in which building blocks are to be executed. Following a connection moves the flow from the source element to the target element. Connections exiting from an Exclusive Choice have both a sequence number and a condition attached to them. Following a conditioned connection is only possible if the condition evaluates to true at run-time.

- Each connection has only one source and only one target.
- Connections exiting an Exclusive Choice (and only those) have both a sequence number and a condition attached to them.
- Following a connection leads the flow from the source element to the target element.
- The source and target must be from the following set of modeling objects: Start State, End State, Building Block, Exclusive Choice and Merge.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description / Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>source</strong>: Object</td>
<td>References the object that the connection is originating from. End States are not allowed to be a source.</td>
</tr>
<tr>
<td><strong>target</strong>: Object</td>
<td>References the object that the connection is connecting to. Start States are not allowed to be a target.</td>
</tr>
<tr>
<td><strong>Sequence</strong>: Integer</td>
<td>Indicates the order in which connections exiting an Exclusive Choice are evaluated. The first connection that evaluates to true is chosen. Connections with the ‘Default’ condition are always evaluated last and always evaluate to true.</td>
</tr>
<tr>
<td><strong>Condition</strong>: String</td>
<td>Specifies the condition that is to be evaluated. Consists of either an expression based on active registry settings or database values, or input that can be given by the user (a key press).</td>
</tr>
</tbody>
</table>

Table 6.2: Attributes for the Connection element

6.3.3.2 Start State

The Start State is the starting point of any transaction type. It has no functionality of itself other than to indicate the starting point and which active registry settings are provided and/or required at that point.

- A Start State has no incoming connections.
- A Start State has only one outgoing connection.
- The outgoing connection can only connect to a Building Block or Exclusive Choice.
- A transaction type can have only one Start State.
<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description / Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>outgoing</strong>: Connection</td>
<td>References the outgoing connections of this element.</td>
</tr>
<tr>
<td><strong>incoming</strong>: Connection</td>
<td>References the incoming connections of this element.</td>
</tr>
<tr>
<td><strong>Requires</strong>: String[0..N] (Read-only)</td>
<td>A list that contains the active registry settings that need to be set before arriving at this element. For Start States this is only used if the transaction type is a container.</td>
</tr>
<tr>
<td><strong>Provides</strong>: String[0..N] (Read-only)</td>
<td>A list that contains the active registry settings that are set upon exiting this element, i.e. active registry settings that are set before execution of the transaction type.</td>
</tr>
</tbody>
</table>

Table 6.3: Attributes for the Start State element

Even though the Start State has no incoming connections, the attribute is listed here. This is because all attributes are listed that originate from the Abstract Syntax. Looking at the Abstract Syntax by itself, a Start State can have incoming connections. This is, however, constrained by the Static Semantics. The same occurs for End States and outgoing connections.

The incoming and outgoing attributes are the same for each of the modeling objects and shall therefore be excluded from the attributes list of the other objects.

### 6.3.3.3 End State

The End State is the final point of any transaction type. It has no functionality of itself other than to indicate the final point and which active registry settings are provided and/or unset at that point. This is determined by looking at the paths that end in the End State and which active registry settings are provided and/or unset by building blocks on those paths. Once a transaction type reaches an End State the transaction type is completed and terminated.

- An End State has no outgoing connections.
- An End State has any number of incoming connections.
- A transaction type can have any number of End States.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description / Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Provides</strong>: String[0..N] (Read-only)</td>
<td>A list that contains the active registry settings that are provided by building blocks on the path that leads to this element. This is only used if the transaction type is a container.</td>
</tr>
<tr>
<td><strong>Unsets</strong>: String[0..N] (Read-only)</td>
<td>A list that contains the active registry settings that are unset by building blocks on the path that leads to this element. This is only used if the transaction type is a container.</td>
</tr>
</tbody>
</table>

Table 6.4: Attributes for the End State element

### 6.3.3.4 Building Block

Building Blocks are the functional parts of a transaction type. They are the only object that can manipulate active registry settings and database values. If the flow of a transaction type reaches a
building block through one of its incoming connections the building block is executed. After execution
the flow is continued along the building block’s outgoing connection.

- A Building Block has any number of incoming connections.
- A Building Block has only one outgoing connection.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description / Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: String (Fixed)</td>
<td>The name of the building block.</td>
</tr>
<tr>
<td>Function code: String (Fixed)</td>
<td>A unique code to identify the building block.</td>
</tr>
<tr>
<td>Description: String (Fixed)</td>
<td>A general description of the building block’s functionality.</td>
</tr>
<tr>
<td>Comments: String</td>
<td>Allows the user to add comments to the building block, e.g. to explain design decisions.</td>
</tr>
<tr>
<td>Variant Slot 1-5: Char</td>
<td>Allows the user to specify the variant codes for this element, through the five variant slots. Only single characters can be entered. Changes to the variant codes lead to appropriate changes to the required, provided and unset active registry settings.</td>
</tr>
<tr>
<td>Requires: String[0..N] (Read-only)</td>
<td>A list that contains the active registry settings that need to be set before arriving at this element.</td>
</tr>
<tr>
<td>Provides: String[0..N] (Read-only)</td>
<td>A list that contains the active registry settings that are set by this element.</td>
</tr>
<tr>
<td>Unsets: String[0..N] (Read-only)</td>
<td>A list that contains the active registry settings that are unset by this element.</td>
</tr>
<tr>
<td>Invokables: Array&lt;Array&lt;BuildingBlocks&gt;&gt;</td>
<td>An array that specifies arrays that specify a sequence of building blocks. Each sequence of building blocks is eligible to be invoked.</td>
</tr>
<tr>
<td>CanInvoke: Boolean (Fixed)</td>
<td>A Boolean value to indicate if it is possible to invoke building blocks from this element.</td>
</tr>
<tr>
<td>Container: Boolean (Fixed)</td>
<td>A Boolean value to indicate if this building block is a container for a (sub) transaction type.</td>
</tr>
</tbody>
</table>

Table 6.5: Attributes for the Building Block element

6.3.3.5 Exclusive Choice

Using the Exclusive Choice object, conditional paths and loops can be defined. Exits originating from an Exclusive Choice have a condition and sequence number attached to them, which allows for alternative paths to be chosen during run-time depending on user input or expressions that evaluate active registry settings or database values. By directing an exit path to a prior building block a loop can be defined.

In order to determine which path to follow the conditions are evaluated in order. The first condition that evaluates to true determines the path to follow. The default exit is always evaluated last and
always evaluates to true. In case all conditions evaluate to false and no default exit has been specified, an exception is thrown.

- An Exclusive Choice has any number of outgoing connections.
- An Exclusive Choice has any number of incoming connections, where all but one are loop edges.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description / Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type:</strong> [Expression, UserInput]</td>
<td>An enumeration value that indicates whether the conditions for the exit paths of this element are based on user input or an expression.</td>
</tr>
</tbody>
</table>

**Table 6.6: Attributes for the Exclusive Choice element**

### 6.3.3.6 Merge

The Merge object allows to merge alternative paths that have originated from an Exclusive Choice. Multiple incoming paths are combined into a single exit path. At the same time, the Merge takes the intersection of the active registry settings that are available for each of the alternative incoming paths. These are the active registry settings that are set after the Merge no matter which of the alternative paths was taken to reach the Merge. In other words, these are the only registry settings of which it can be assured that they are available after the Merge. Only these active registry settings will be used to verify if subsequent building blocks have their required settings at their disposal. More on the details of the Automated Dependency Checker can be found in Section 6.4.4.

- A Merge has any number of incoming connections.
- A Merge has only one outgoing connection.

A Merge has no attributes that need clarification.

### 6.4 Implementation Phase

The DSL is implemented in a proof-of-concept. The proof-of-concept is meant to demonstrate the suitability of the DSL and show that the visual modeling and validation of transaction types is feasible. A graphical development environment is implemented that exhibits this functionality. In other respects the development environment is limited in functionality compared to full-scale development tools.

This phase of the DSL development describes the implementation details of the proof-of-concept. First the consequences of the choice for UML to describe our abstract syntax is discussed. This is related to the proof-of-concept as the UML model is used as input for the modeling tool to construct the proof-of-concept. Secondly, the graph structure is described, that shall be used to base the 2nd level validation steps on. Thirdly, a choice is made for a modeling tool with which to implement the proof-of-concept. Lastly, the way the chosen modeling tool is used to implement the proof-of-concept is presented.
6.4.1 UML

To describe the meta-model of our domain it has been decided to use UML. A consequence of this choice is the need to use UML not only now, but UML is also required in the future might changes to the DSL be necessary. This could require the use of UML modeling tools, which may involve licensing costs, and UML modeling knowledge to use them.

In our case licensing costs are not required, since we can, and will, make use of Ecore for describing our model. This is a model that is aligned on the Object Management Group’s (OMG) Essential Meta Object Facility (EMOF)\(^\text{10}\). This in turn is a DSL for defining meta-models. Since EMOF originated in UML, Ecore still possesses concepts as classes, attributes and association, generalization and aggregation relationships; like UML does. Ecore models can be defined using the Eclipse Modeling Framework (EMF)\(^\text{11}\), an open-source Eclipse-plug-in, which is free of licensing costs and eliminates the need for other meta-modeling tools.

Furthermore, Ecore provides interoperability with all EMF-based tools and applications and can also be imported, albeit requiring more effort, by other well known DSL development tools as MetaEdit+ and Microsoft DSL Tools. This provides Zetes with the freedom to choose a DSL development tool, while using the defined meta-model.

Although UML is hardly used within Zetes and therefore UML modeling knowledge is lacking, this will unlikely present a problem. Only a limited subset of UML is used to describe our meta-model and the required knowledge on UML is thus limited as well. The required UML modeling knowledge can be easily acquired.

This makes UML a suitable choice for this project. This does not make it certain that UML shall be used for the final product. Zetes could opt to choose a language that better aligns with their current used modeling languages. They could also choose a modeling tool that does not require an UML model as input, which would remove one of the advantages of using UML to describe our meta-model.

6.4.2 Graph Structure

The graph structure, like any graph, consists of vertices and edges. For our structure custom vertices and custom edges have been designed to represent the information that corresponds to transaction types. Every vertex and edge is an instantiation of the class Vertex or Connection respectively. These classes, their variables and their functions are as follows:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description / Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID: int</td>
<td>Used to uniquely identify each vertex.</td>
</tr>
<tr>
<td>Type: [BuildingBlock, ExclusiveChoice, Merge,</td>
<td>An enumeration value indicating the type of</td>
</tr>
</tbody>
</table>

\(^\text{10}\) http://en.wikipedia.org/wiki/Meta-Object_Facility/

\(^\text{11}\) http://www.eclipse.org/modeling/emf/
In the graph structure each modeling object is represented by a vertex and the connections between them by edges. Information that is not of importance validating the transaction type structure or building block dependencies have been excluded from the graph structure, e.g. building block names and variant codes.

An important piece of information (for the ADC) is the aggregate variable of edges. This variable contains an aggregate of the provided active registry settings so far. In other words, it indicates which active registry settings are available on that edge. By comparing this to the active registry settings that are required by the target of the edge, one can decide if the necessary settings are available at that point. Note that due to loops the aggregate active registry settings are not necessarily the same on subsequent passes of a certain edge, but this is taken into account by the ADC.

A coupling between the internal representation of the modeling tool and the graph structure exists by references between the corresponding elements. Vertices contain a reference to the model element it belongs to and vice versa for model elements. This is needed for the ADC to be able to provide feedback to the model.

### Table 6.7: Vertex details

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description / Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartState, EndState</td>
<td>model element this vertex represents.</td>
</tr>
<tr>
<td><strong>Element</strong>: ObjectReference</td>
<td>A reference to the model element to which this vertex belongs.</td>
</tr>
<tr>
<td><strong>Provides</strong>: String[0..N]</td>
<td>A list that contains the active registry settings that are provided by the model element to which this vertex belongs.</td>
</tr>
<tr>
<td><strong>Requires</strong>: String[0..N]</td>
<td>A list that contains the active registry settings that are required by the model element to which this vertex belongs.</td>
</tr>
<tr>
<td><strong>Unsets</strong>: String[0..N]</td>
<td>A list that contains the active registry settings that are unset by the model element to which this vertex belongs.</td>
</tr>
</tbody>
</table>

### Table 6.8: Edge details

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description / Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1: Vertex</td>
<td>A reference to the source vertex of this edge.</td>
</tr>
<tr>
<td>v2: Vertex</td>
<td>A reference to the target vertex of this edge.</td>
</tr>
<tr>
<td><strong>Type</strong>: [Regular, UserInput, Expression]</td>
<td>An enumeration value indicating the type of connection this edge represents.</td>
</tr>
<tr>
<td><strong>Connection</strong>: ObjectReference</td>
<td>A reference to the model connection to which this edge belongs.</td>
</tr>
<tr>
<td><strong>Sequence</strong>: int</td>
<td>The sequence number of the connection to which this edge belongs.</td>
</tr>
<tr>
<td><strong>Condition</strong>: String</td>
<td>The condition of the connection to which this edge belongs.</td>
</tr>
<tr>
<td><strong>Loop</strong>: Boolean</td>
<td>Indicates whether this edge loops back to a prior vertex.</td>
</tr>
<tr>
<td><strong>Aggregate</strong>: String[0..N]</td>
<td>A list that contains which active registry settings are at least available on this edge.</td>
</tr>
</tbody>
</table>

In the graph structure each modeling object is represented by a vertex and the connections between them by edges. Information that is not of importance validating the transaction type structure or building block dependencies have been excluded from the graph structure, e.g. building block names and variant codes.

An important piece of information (for the ADC) is the aggregate variable of edges. This variable contains an aggregate of the provided active registry settings so far. In other words, it indicates which active registry settings are available on that edge. By comparing this to the active registry settings that are required by the target of the edge, one can decide if the necessary settings are available at that point. Note that due to loops the aggregate active registry settings are not necessarily the same on subsequent passes of a certain edge, but this is taken into account by the ADC.

A coupling between the internal representation of the modeling tool and the graph structure exists by references between the corresponding elements. Vertices contain a reference to the model element it belongs to and vice versa for model elements. This is needed for the ADC to be able to provide feedback to the model.
The graph is implemented using the generic graph library JgraphT\textsuperscript{12} for Java.

6.4.3 Modeling Tool

To enable the use of the developed DSL a software tool is needed that allows the creation of models according to the DSL. In [31] it is even claimed that MDE and the use of DSLs does not make sense without tool support. The minimal features that such a software tool must possess are [43]:

- A modeling environment that allows the creation and editing of visual models for the DSL.
- An Artifact Generator responsible for generating development artifacts based on the constructed models. Source-code is often the development artifact in question. In our case, however, the storage of correct transaction types in the database is the development artifact.

A list of additional useful features is given in the same paper of which Model Validation is especially applicable to our case. During analysis and design, constraints have been defined to which the models must adhere and the tool should therefore provide the means to validate models against these constraints. Other features which might proof useful are Model Transformations, Model Analysis and a Model Debugger. These features have a lower priority though.

Developing the software tool, i.e. a model-driven IDE for our DSL, can be done in different ways, classified into three categories:

- Development from scratch – No libraries or frameworks are used that are external to the implementation language.
- Development with reuse – One or more libraries or frameworks are reused, e.g. artifact generators or diagramming libraries.
- Development with meta-modeling tools – A meta-modeling tool is used that, based on a formal definition of a DSL, generates an IDE for that DSL.

Of these three categories, development using meta-modeling tools is found to be the cheapest and is the most suitable choice in the majority of situations. An exception to this rule can be made if the DSL describes a domain that has a sufficiently large (potential) user base and/or market [43]. As the DSL for Zetes is only meant for internal use this is obviously not the case.

The development approach for the proof-of-concept shall thus make use of a meta-modeling tool. The tool used to develop the proof-of-concept needs to fulfill certain added requirements which may not be applicable to the final IDE that is to be developed:

- Easy to learn – Only limited time is available for the development of the proof-of-concept. Because of this it becomes important that as little time as possible is spent on understanding the tool.

\textsuperscript{12} http://www.jgrapht.org
- License free – The proof-of-concept is not meant to be a complete solution, but merely serves to demonstrate the feasibility of the visual modeling of transaction types. It therefore is unwanted to choose a tool that incurs licensing costs. This holds, granted that open-source alternatives are available that meet the other requirements.

The most commonly used meta-modeling tools are Microsoft DSL Tools, MetaEdit+, Graphical Modeling Framework (GMF) and the Generic Modeling Environment (GME). Of these, GMF is chosen as the tool to develop the proof-of-concept. GMF meets the stated requirements and is compared to the other options the most suitable for our purpose. Microsoft DSL Tools is said to have a steep learning curve [44] and as such is not a valid option. MetaEdit+ requires licensing costs and furthermore has more restricted constraint definition possibilities than GMF [44]. The problem with GME is that it is mainly suitable for simple prototyping of DSL editors. Although our proof-of-concept does not aim to provide a full solution, GME could proof to be a too limited tool. GMF on the other hand, through its integration with Eclipse, provides the user with complete freedom to edit or extend the generated code and implement any IDE requirements.

The choice for GMF to develop the proof-of-concept does not necessarily make it the most suitable choice to develop the full IDE with. For the full IDE that is to be developed other requirements may be of concern, while the requirements of being easy to learn and free of licensing costs may no longer be applicable. Additional research shall be necessary to decide on the fittest meta-modeling tool for the full IDE.

6.4.4 Proof-of-Concept

6.4.4.1 Concept

Given the limited time available for implementation during the execution of this project, it is unfeasible to implement a fully functioning IDE. For this reason, it has been chosen to implement a proof-of-concept that serves to demonstrates the suitability of the DSL and the feasibility of the visual modeling of transaction types and checking the validity of the constructed models. This represents the most important part of the IDE that is to be developed. The following features of the IDE are implemented by the proof-of-concept:

- The visual modeling of transaction types according to the DSL
- The Automated Dependency Checker

This excludes other features of which the most important are:

- Database connection – No connection to the database is implemented. For this proof-of-concept, several building blocks shall be hardcoded so they can be used during modeling. Building blocks however, shall not be retrieved from the database, nor shall modeled transaction types be stored in the database.
- Localization – The implementation of a search function and the support for templates, as detailed in Chapter 5: Localization, shall not be available in the proof-of-concept.
- Formalized conditions – Conditions are implemented as a text string; they are not formally structured.
These features are separate from the main functionality of the IDE, which is the modeling and validation of transaction types. In order to remain as close as possible to the full IDE, the hardcoded building blocks contain only that information which would normally be retrieved from the database and the transaction types contain all the information that is needed to store them in the database. Implementing the database connection would thus not change the way in which transaction types are modeled and validated. The localization features are only for the convenience of the user in speeding up transaction type modeling. They, again, do not alter the way transaction types are modeled and validated.

In this way, the results presented in the proof-of-concept concerning the DSL, model creation and model validation are still applicable to the full IDE.

6.4.4.2 Implementation

In this section first the general outline of the proof-of-concept is presented and the way GMF is used for the development of it. After this, pseudocode is given for important handwritten parts of the proof-of-concept. Namely, the conversion of the transaction type model to a directed graph, the validation of the graph structure and the validation of the registry settings, i.e. the ADC. These are the parts that are independent of the modeling tool that is used; therefore, the approach given in pseudocode below can also be used for the final IDE.

An overview of the GMF process can be seen in Figure 6.11. Using GMF for software development starts with the construction of an Ecore file (.ecore), an UML-like model that defines the abstract syntax; also known as the Domain Model. In our case, this corresponds to the abstract syntax given in Section 6.2.5. Based on this Ecore file, three more files are created, partly generated by GMF, partly manually created by the user.

![Figure 6.11: GMF process overview](image)
The graphical definition (.gmfgraph file) defines the visual appearance of objects and connections that can be dragged on the canvas. This is a combination of standard shapes, e.g. rectangles and ellipses, and custom user-defined polygons. The user can also set visual properties for these shapes as color, size, border width, etc. In the tooling definition (.gmftool file) the user defines the GMF palette. The palette forms a menu for the IDE that contains the objects and connections that can be dragged onto the canvas. The mapping definition (.gmfmap file) brings together all of the above. Here the graphical and tooling definitions are mapped to the domain model. This is also the place at which OCL constraints can be defined. These can be link constraints, limiting the source and target ends of a certain type of connection, and model constraints, which are constraints applied to the model as a whole.

From these three files a generator model (.gmfgen file) is automatically created, which allows the user to adjust certain generation parameters. Once this is done, the generator model is used to generate the actual source code for the IDE; in our case the proof-of-concept. To modify the generated IDE to satisfy specific user needs, the generated source code can be freely altered and extended.

Our proof-of-concept contains three validation steps, OCL validation, graph validation and registry validation (ADC), following the structure from Figure 6.3: Transaction Type Process Overview.

The OCL validation is a generated functionality by GMF based on the OCL constraints defined in the mapping definition. Therefore, this validation step is discussed no further. The other two validation steps, on the other hand, are handwritten functionalities based on the directed graph that is derived from the transaction type model. Below, an explanation of the workings of constructing the graph and the two validation steps is given. This is done by providing pseudocode accompanied by a brief description of it.

**Graph construction**

```plaintext
Function followOutgoing(processed, source) do
  for E in outgoingConnections[source] do
    if target[E] not in processed then
      if target[E] is BuildingBlock then
        vertex := newBuildingBlockVertex()
      else if target[E] is Merge then
        vertex := newMergeVertex()
      else if target[E] is ExclusiveChoice then
        vertex := newExclusiveChoiceVertex()
      else if target[E] is EndState then
        vertex := newEndStateVertex()
      end if
      add vertex to graph
      add edge from source to target vertex
      add source to predecessors[target]
      followOutgoing(processed, target[E])
    else
      add edge from source to target vertex
      add source to predecessors[target]
    end if
  end for
end function
```

The graph is constructed by going through the transaction type in a recursive way. For a given source object (BuildingBlock, ExclusiveChoice, Merge, StartState or EndState) each of its outgoing
connections are followed. For each of those connections the type of the connection’s target is
determined and an appropriate vertex is added to the graph – granted that the connection’s target was
not yet processed. After this an edge is added from the source vertex to the target vertex. The target
object is added to the list of processed objects and serves as the source for the next recursive call. If an
object is encountered that has already been processed, it is only necessary to add the edge
corresponding to the followed connection to the graph. In both cases the source vertex is added to the
predecessor’s list of the target vertex.

Initially, the Start State is added to the processed list and a vertex for the Start State is added to the
graph. Then followOutgoing is called with the vertex of the Start State as the source parameter. This
way, each object reachable from the Start State and all of its outgoing edges are processed, such that
the graph represents the full transaction type. Parts that are unreachable from the Start State are not
included in the graph, nor should they be. This occurrence is handled during the validation of the
graph structure.

Graph structure validation
The validation of the graph structure is needed to assure that a valid transaction type has been
modeled. This is a prerequisite for the validation of the building block dependencies. For this purpose
the algorithm needs to perform the following:

- Determine if the transaction type is connected
- Check if all objects can reach an End State
- Detection of (in)valid loops
- Detection of crosspaths

The first two points are easily handled. To check whether the graph is connected and all objects are
reachable from the Start State, the number of vertices in the graph is compared to the number of
objects in the transaction type. Because the graph only includes objects that are reachable from the
Start State, the size of the graph is smaller than the size of the modeled transaction type if the
transaction type is not fully connected.

To check if all objects can reach an End State it is only necessary to check whether Exclusive Choices
that are the source of a loop edge can reach an End State. If this is the case than automatically all
objects can reach an End State. By using any pathfinding algorithm this can be checked.

The detection of loops and crosspaths is a more complicated problem. To achieve this, the method by
Sreedhar et. al. is used [45]. This method describes Dominator-Join graphs and how they can be used
to detect reducible and irreducible loops. In a DJ graph the edges are divided into either Dominator,
Crossjoin or Backjoin edges. A dominator edge \( x \rightarrow y \) indicates that \( x \) immediately dominates \( y \) in the
original graph, i.e. the transaction type. Any edge \( x \rightarrow y \) where \( x \) does not immediately dominate \( y \) is a
join edge. Such a join edge is a backjoin edge if \( y \) dominates \( x \) and a crossjoin edge otherwise. [45]
provides a more in-depth look into DJ graphs.
Following from the theory of DJ graphs it follows that a backjoin edge is the loop edge of a reducible loop and that a crossjoin edge should always target a Merge or End State object. Loop edges should furthermore originate from an Exclusive Choice and are not allowed to target a Merge. These conditions are checked in the algorithm for which the pseudocode is given above.

The only additional check is necessary for reducible loops that in our case are illegal. An example of such a loop can be seen in Figure 6.9: Un-allowed loop. Such a loop is considered reducible, and thus allowed, according to the theory of DJ graphs. For transaction types, however, it is an unwanted construct as it complicates the ADC.

Registry settings validation
The registry validation works in a similar way as the way the graph is filled, namely by recursion. In this case, for a certain vertex it is decided which settings are available after execution of the corresponding object. Each outgoing edge of the vertex has that aggregate set as being the settings that are available on that edge. Then for each outgoing edge it is checked whether that edge satisfies all registry requirements of its target. Where required settings are missing, this is indicated through error markings.

The aggregate is formed by the intersection of the aggregates of the incoming edges of the vertex, minus the settings that are unset by its invokables, plus the settings that are provided by the corresponding model element, minus the settings that are unset by the model element. The required settings of a vertex are formed by the required settings of the model element plus the required settings by its invokables.
It is necessary to assure that on each iteration of a loop all required settings are available. For this, it suffices to traverse the loop one additional time. If all required settings are satisfied for building blocks inside a loop during the first iteration, then the only way that this can change if certain required settings are unset inside the loop. One additional traversal of the loop detects this. This is what is done for Exclusive Choices in the algorithm. Once for an Exclusive Choice all loop exits have been processed, the algorithm continues with the remaining exits.

A vertex that represents a Merge is only traversed, once for all incoming edges it is known what their aggregate is. Otherwise it cannot be determined which settings are certainly available for the outgoing edge of the Merge.

The proof-of-concept is the result of the implementation of the pseudocode above in combination with the IDE generated by GMF. A demonstration of the proof-of-concept was given to five SA’s at Zetes, who completed a survey afterwards. The results of this survey is used as part of the validation of the DSL, which can be found in Chapter 8.

```java
Function checkOutgoing(processed, source)
switch source.type
  case StartState:
  case BuildingBlock:
    aggregate[outgoing] := determineAggregate(source)
    checkEdge(outgoing)
    add outgoing to processed
    checkOutgoing(processed, target[outgoing])
    break
  case ExclusiveChoice:
    aggregate := determineAggregate(source)
    outgoingSet := outgoingEdgesOf(source)
    loops := getLoopExits(source)
    for L in loops do
      if not L in processed then
        aggregate[L] := aggregate
        checkEdge(L)
        add L to processed
        checkOutgoing(processed, target[L]
        break case
      end if
    end for
    nonLoops := outgoingSet \ loops
    for E in nonLoops do
      aggregate[E] := aggregate
      checkEdge(E)
      add E to processed
      checkOutgoing(processed, target[E])
    end for
    break
  case Merge:
    incomingSet := incomingEdgesOf(source)
    if incomingSet in processed then
      aggregate[outgoing] := determineAggregate(source)
      checkEdge(outgoing)
      add outgoing to processed
      checkOutgoing(processed, target[outgoing])
    end if
    break
  end case
end function
```
A new database model is required that stores the information on registry settings and variant codes for building blocks. This allows a user to view this information in the development environment. Furthermore, the information is necessary for the ADC to function. The database model also needs to be altered to be able to cope with conditional paths and loops. In the concept section below it is specified which additional information needs to be stored, followed by the design of the new database model in the implementation section.

7.1 Concept

In the new modeling environment there are several additional demands for the database. It should be able to store more information regarding building blocks and transaction types and in particular should support the representation of transaction types that incorporate conditional paths and loops. Information that is not yet stored in the current database, but should be in the new situation consists of the following:

Concerning building blocks:

- Active registry settings
  - Requires
  - Provides
  - Unsets
- Allowed variant codes, including variant descriptions
- Influence of variant codes on active registry settings
- Tags

Concerning transaction types:

- Conditional path construct
  - Paths
  - Conditions
  - Sequence number
  - Type
- Loops\(^{13}\)
- Invokables per building block of a transaction type
- Comments per building block of a transaction type

\(^{13}\) Loops shall be implemented as a conditional path to a prior building block.
In order to achieve this the existing database model needs to be modified and extended. The starting point remains the current database model and this will only be altered where necessary. As a reference the current database model is included here once more.

In the next section all of the above additions shall be included in the new database model through additions to existing tables and extra database tables.

### 7.2 Implementation

The model in Figure 7.2 shows the new database model. On the basis of this an explanation for the model and the choices that were made are given. To clarify, building blocks are named *optional_functions* in the database model. Building blocks that are in use in a transaction type are identified as *activated_functions*.

The three tables *Transactiontype*, *Optional_function* and *Activated_function* which constituted the old database model are still present in the new database model, albeit in a slightly altered form. The attributes *transactiontype_name*, *version*, *creationdate* and *author* are added to the *Transactiontype* table on one hand. The *instruction_on_screen* attribute has been removed on the other hand; it is obsolete in the new situation. In the old situation, the *transactiontype_description* was used to name the transaction type. Now, however, it is possible to specify a name and a short description in separate attributes. The same has been done for the *Optional_function* table. A *can_invoke* attribute has been added to this table as well to indicate whether the building block is capable of invoking other building blocks.
The *Optional_function* table, as well as the *Activated_function* table, furthermore no longer has the *hotkey* attribute. This was an attribute that was no longer used and could be removed without any consequences. In the *Activated_function* table the attributes *include_in_startup*, *include_in_closing* and *end_transaction* have also been removed. As explained in Section 6.2.3, these marking for building blocks are no longer necessary, due to the addition of conditional paths and loops, and could
therefore be removed. The function_sequence_number has been replaced by the internal_ID attribute. Its function remains largely the same, to identify building blocks within the transaction type, but since a transaction type is no longer necessarily sequential, its name was changed. Besides the comments attribute, which speaks for itself, invokable_ID and target_ID have been added to this table. These serve to support the invokables for building blocks and the conditional paths respectively and shall each be discussed along with the other additions below.

The database model uses the Crow’s foot notation to indicate the cardinality of the relationships. Colors are used to group relationships that are related, but have no functional meaning. The black relationships correspond to the relationships in the original database model. The other colors are explained below.

7.2.1 Active registry settings & Variant codes

The new database should be able to store, for each building block, which variant codes are allowed for each variant slot, which active registry settings are provided, required and unset and the influence of variant codes on active registry settings. This is achieved by the four tables Optional_function, Active_registry_setting, Variant_code and Variant_to_ars. The corresponding relations are colored blue.

In the Active_registry_setting table, for each building block, identified by its optional_function_code, an active registry setting, an action and an optionality can be stored. The action attribute can be set to either ‘required’, ‘provided’ or ‘unset’ to indicate in which way the active registry setting is used. The optionality attribute indicates if this is either mandatory or optional. Note that this does not limit an active registry setting from being both required and unset.

In the Variant_code table is stored what the allowed variant code values of a variant slot are, for each building block. A slot number between one and five and a corresponding value and description of its functionality can be coupled to a certain building block in this table.

The table Variant_to_ars finally couples a specific variant code of a specific slot to a certain active registry setting to indicate that that variant value is needed for the information in the active registry setting table to be applicable. A constraint, thus, for this table is that the active_registry_setting_ID and the variant_code_ID belong to the same building block, i.e. optional_function_code. Default active registry settings that are not dependent on any variant codes are not coupled through the Variant_to_ars table; all others are.

7.2.2 Conditional paths

Transaction types are in the new situation no longer necessarily sequential. Thus, this needs to be represented in the database. This is done by coupling to each building block in a transaction type, a target_ID. This ID in turn can point, in the Target table, to several internal_IDs, i.e. building blocks within the transaction type, that can be targeted. For both regular transaction types and containers, these relationships are indicated in green. In this way, the outgoing connections of a building block can be stored, possibly with a condition and sequence number attached to them.
In this database model, the condition is stored as a string value. For expressions the condition could be further formalized. For example, by dividing the condition into two values and an operator. The values could then be either a constant, an active registry setting or a database value. Possible operators could include smaller than, greater than and equals. However it is at this point now clear what the requirements for the conditions are. The need for arithmetic for example would already complicate the above division. By representing the condition as a string value, the option is left open to formalize this at a later point. Note that the future modeling environment should ensure that only valid expressions are stored in the database.

For two building blocks in sequence with internal_IDs 1 and 2 respectively, the target_ID of the first building block would in the Target table be coupled with internal_ID 2. The type would now indicate that this is a sequential flow.

### 7.2.3 Invokables

Those building blocks that have their can_invoke attribute set to true, can have a number of invokables attached to them when they are used in a transaction type. These invokables are usually single building blocks, but can also be a sequence of multiple building blocks. The Invokable and Invoke_sequence tables are responsible for supporting this. The corresponding relations are colored orange.

The invokable_ID that is associated with a building block refers to several invokable_sequence_codes through the Invokable table. In the Invokable_sequence table in turn, one such code is associated with a sequence of building blocks, along with appropriate variant codes. Multiple invokable sequences of building blocks can be stored in this way for a single building block in a transaction type.

### 7.2.4 Containers

As a way to make large transaction types more comprehensible or to enable reuse of an oft used collection of building blocks, container functionality is added in the new situation. This is the ability to specify a small (sub) transaction type and use this as if it were a building block. This container would thus contain properties of both transaction types and building blocks. On one hand it is a transaction type, because it contains building blocks, which are related to each other in a certain way as a transaction type. On the other hand, the container has provided, required and unset active registry settings associated with it like a building block and it can be used as such in a transaction type.

For this reason, the Activated_function table is included once more in the database model; now as the table Contained_function. This table is a copy of the Activated_function table, with the exception that it no longer is a collection of optional functions belonging to a transaction type, but a collection of optional functions belonging to an optional function. Corresponding relationships are colored red.

A (sub) transaction type can be stored in the Contained_function table in the same way as for the Activated_function table, but it also has properties of a building block since it is associated with an optional function. In this way, a container can have active registry settings associated with it and can be used as a building block in a transaction type.

In the case that a container is used in a transaction type, the attributes of the optional function concerned with the variant codes and the invokables are not applicable.
7.2.5 Tags

In correspondence to decisions made in Chapter 5, the ability has been added to store tags for both transaction types and building blocks to facilitate the localization of them. Two separate tables have been added for this: the table Transactiontype_tags to store multiple tags per transaction type and the same for building blocks in the table Optional_function_tags. These relationships for both transaction types and building blocks are colored purple.

7.2.6 Normalization

In the design of a relational database, normalization is the process of organizing data to minimize redundancy, which in turn reduces the chances of inconsistencies and anomalies in the database. This is achieved by decomposing tables into smaller, less redundant tables and defining relationships between them. Normalization or decomposition is done according to several available Normal forms. These normal forms consist of a set of rules which, when followed, reduce the potential for logical inconsistencies and anomalies. Namely, by isolating data so that insertions, updates, and deletions of a field can be made in just one table and then propagated through the rest of the database via the defined relationships. The higher the normal form, the less vulnerable the database is to inconsistencies and anomalies. Of the available normal forms, the Third Normal Form (3NF) is usually a minimal and sufficiently restricted normal form. Tables in 3NF are typically free of insertion, update and deletion anomalies.

Because of the above and the fact that 3NF is sufficiently restrictive for the situation at Zetes\(^\text{14}\), the database has been designed in 3NF.

\(^{14}\) As indicated by database adepts within Zetes.
8. Validation

In the introduction of this thesis a research goal was stated (see Section 1.2), which represents the most important goal of this thesis. After analysis of the current situation regarding the modeling of transaction types, a number of problems were identified as well (see Section 2.3). These problems are related to the research goal and solving them are sub-goals. In this chapter the usefulness and feasibility of the presented solutions are validated, i.e. the granularity guidelines, the localization mechanisms, the DSL and the database model.

This is done in two ways. For some goals it can be objectively verified that they are achieved. For these, arguments are provided to show that those goals have indeed been achieved. To validate goals and corresponding solutions that are of a more subjective nature, a survey is held among five System Architects of Zetes to determine the quality of the solutions. These are the people who eventually have to execute and work with the solutions provided in this thesis and are thus preeminently suited to validate them. The survey can be found in Appendix F: Thesis Presentation Survey. First the validation of the result of the research goal is discussed, followed by the validation of the results of the sub-goals.

8.1 Research Goal

The research goal is as follows:

- Design a formal language specification for the modeling of transaction types using the standard building blocks.

Objectively, it can be seen that such a formal language specification has been designed. Namely, in the form of a Domain Specific Language. This is a formal language consisting of an abstract syntax, static semantics, a concrete syntax and corresponding semantics. The DSL developed in this thesis is specific for the modeling of transaction types using the standard building blocks.

The DSL can also objectively be seen to satisfy the requirements of the language:

- It should support the modeling of sequential flow.
  - This is supported through the use of the Sequence construct (see 6.3.2.1).

- It should support the modeling of conditional paths.
  - This is supported through the use of the Exclusive Choice and Merge constructs (see 6.3.2.2 & 6.3.2.3).

- It should support the modeling of loops.
  - This is supported through the use of the Loop construct (see 6.3.2.4).
- It should make automated dependency checking of building blocks possible.
  - The proof-of-concept demonstrates the feasibility of implementing an automated dependency checker on the basis of the DSL (see 6.4.4).

- It should be suitable to be used as a basis for a graphical development environment that allows for visual modeling of transaction types using the standard building blocks.
  - The proof-of-concept demonstrates the feasibility of using the DSL as a basis for a graphical development environment. The DSL can actually directly be used as input for certain meta-modeling tools to generate such a graphical development environment. This is also the way in which the proof-of-concept was constructed (see 6.4.3).

However, the fact that a formal language specification has been designed does not indicate anything about the quality and suitability of the design. For this we have to look at the results of the survey. The survey contained questions about the DSL regarding the amount of freedom it provides, its ease-of-use, speed of modeling and usefulness. Usefulness is one of these the most important quality attributes. On a 5-point scale ranging from Useless to Very useful, all five survey participants indicated the usefulness of the DSL as Useful or Very useful. The ease-of-use and speed of modeling were rated as Good and Faster respectively by the majority.

The amount of freedom the DSL provided regarding transaction types was rated to be only sufficient. This can be explained by the restrictions placed on loops and crosspaths by the DSL, which was necessary to restrain the complexity of the ADC. In the future these restrictions can be lifted, although the DSL already suffices for their needs at the moment. The clarity of the error indications is another point that was rated only sufficient and could use work. Currently, several errors with different causes are combined under the error ‘Illegal connection’, with a reference to the connection in question. The clarity of the errors could be improved by indicating why the connection is illegal. Due to time constraints this is not yet fully included. The usefulness and ease-of-use of the ADC was rated as good or above.

### 8.2 Sub-goals

Solving the identified problems of the way in which transaction types are currently modeled constitute the sub-goals. For each of the problems it is argued that sufficient solutions have been presented to solve the problem.

#### 1. Granularity guidelines for building blocks are missing

In response to this problem, five granularity guidelines have been presented. The usefulness and feasibility of each of these guidelines is validated through the survey.

For each of the guidelines, the participants were first asked to estimate on what number of building blocks the guideline would be applicable. This gives a first impression on the usefulness of the guideline. For all guidelines it was found it is likely applicable to over twenty building blocks. This is more than enough to warrant the existence of each of the guidelines.

Furthermore, ratings were given on the usefulness of reducing a single building block’s granularity on the one hand and on the usefulness of improving the clarity of the building block’s functionality on the
other hand. For both, the ratings were either **Useful** or **Very useful** for each of the guidelines, except for guideline number 4. This is the guideline that advocates to check building blocks with a high cyclomatic complexity and was rated as **Somewhat useful** by the majority. As this was the most abstract guideline of the five and it is quite unclear to the results this guideline will provide, this is not unexpected. The rating however does not warrant the removal of the guideline.

No problems were identified that would inhibit the feasibility of any of the guidelines.

2. **The modeling environment lacks mechanism for locating building blocks**

Two mechanisms to facilitate the localization of building blocks have been presented. Namely, the integrated search function and templates. The usefulness and feasibility of each of these mechanisms is validated through the survey.

The participants were asked to rate both mechanisms on their usefulness in localizing building blocks and speeding up the modeling of transaction types. It was found that the templates were not likely to help in localizing specific building blocks, but would proof to be useful in speeding up the modeling process. The search function was found to be useful in both respects.

With respect to the feasibility of the search function, it was noted that with an increasing collection of building blocks the performance might become an issue. This however is unlikely to occur as the building block collection would have to increase enormously for it to be an issue. Even then it would be an implementation issue.

3. **Building block requirements/provisions are not visible in the modeling environment**

4. **Variant information is not visible in the modeling environment**

For both of these problems it was necessary to add additional information to the database, such that the modeling environment is able to retrieve this information and present it to the user. Looking at the new database model, it can objectively be seen to include the necessary information on provided, required and unset settings per building block as well as the necessary information on variant codes per building block. The tables Active_registry_setting and Variant_code respectively are responsible for this.

The survey does not include questions on the new database model, since not all the System Architects are database adepts. The database model was, however, constructed in cooperation with a database adept within Zetes and the correctness of the database model is as such validated.

5. **The modeling environment delivers no automated dependency checking**

The validation of the automated dependency checker is handled above during the validation of the research goal.

6. **Error messages do not provide useful information**

This problem, as stated in Section 3.5, is not handled in this thesis.

7. **Conditional paths and loops are not possible within a transaction type**

The validation of conditional paths and loops, which are included in the DSL, is handled above during the validation of the research goal.
9. Future Work

Several steps remain in order for Zetes to fully profit from the work contained in this thesis. These steps consist, one the one hand, of applying the recommendations and designs in this thesis and, on the other hand, of creating a fully functioning modeling environment according to the DSL. In this chapter the different steps are presented in the preferred order of execution.

1. Analyze the building blocks
There are two reasons to look at each building block in the collection. First of all, each building block needs to be checked against the granularity guidelines and should, if necessary, be altered according to the guidelines. This is not a strict prerequisite for making use of the DSL or the modeling and validation of transaction types in general, but it is recommended. Following the guidelines improves the flexibility and reusability of building blocks, while the container functionality and ADC ensure that the ease of assembling a transaction type remains intact. The potential benefit of the DSL is improved if the building blocks are of the proper granularity.

Second of all, to optimally profit of the integrated search function, tags should be attached to each building block.

Given the number of building blocks in the collection, this is a labor-intensive and time-consuming task. This task is furthermore not a necessity to be able to profit from the options that the DSL provides, i.e. the graphical modeling and validation of transaction types. Therefore, this task is the prime candidate to be skipped if time-saving and cost-saving choices have to be made.

2. Implement and fill the database
In order to present building block information in the modeling environment and make use of the ADC, it is necessary to implement the database model and fill it with the appropriate building block information, which is now stored in individual documentation files. This task should be completed before being able to utilize the DSL. It can be executed simultaneously with the following two tasks though.

3. Choose modeling tool
An important decision that still needs to be made is which meta-modeling tool to use to construct the IDE. As detailed in Section 6.4.3, various meta-modeling tools are available. Which one of these is the most appropriate to implement the IDE is not studied in this thesis. A choice was made to use GMF for the proof-of-concept. However, the complete IDE may have different requirements for the modeling tool. A study shall have to be performed to determine the most suitable choice. A useful article for this study is “Model-Driven Engineering and its introduction with metamodeling tools” by Lukman et. al. [43]. This article documents the project of procuring the fittest meta-modeling tool for the implementation of an IDE. The documented process can be used a method for choosing the fittest meta-modeling tool in our case.
4. Implement Integrated Modeling Environment

An IDE for the modeling of transaction types has to be developed and implemented. A possibility is to extend the proof-of-concept to a full IDE, which is a realistic option if GMF is going to be the modeling tool of choice. Extending the proof-of-concept, the following points will have to be addressed to make it deliver all the necessary functionality.

- **Database connection** – The proof-of-concept lacks a connection to the database in order to, for example, retrieve building block information or to store transaction types. Implementation changes need to be made such that required information is retrieved from the database and appropriate transaction type information is stored to the database.

- **Formalized conditions** – Conditions for Exclusive Choice exits have to be formalized, so that it can be ensured that the conditions can be correctly evaluated at runtime. Currently, no form of formalization has been introduced.

- **Localization mechanisms** – The localization mechanisms presented in this thesis have not been implemented in the proof-of-concept.

- **Container functionality** – Only part of the container functionality is implemented in the proof-of-concept. Containers can be added to a transaction types and containers can be opened from within a transaction type. However, changes to a container are not propagated to the transaction types that use that container.

- **Variant codes for invokables** – In the current form, the proof-of-concept does not offer the possibility to set the variant codes for invokables.

- **Improve ease-of-use** – During the construction of the proof-of-concept not much attention was given to the ease-of-use of the modeling environment. The interface could be improved by, for example, adding tooltips and keyboard shortcuts.

- **Clean up code** – Although the code has been designed and written in a structured way, it is worth spending more time on this subject. Due to the time pressure and last minute changes, the code can be illogical or inefficient in places. Future maintenance and additions to the code become easier if this is addressed.

In case a different meta-modeling tool is chosen than GMF the development has to start anew. The following point can be reused independently of the modeling tool:

- **The DSL**
  - Abstract syntax
  - Static semantics
  - Concrete syntax
  - Semantics
- **Implementation of**:
  - Graph construction
  - Graph validation
  - Registry validation
The DSL remains the same regardless of the modeling tool. For the validation of the structure of the transaction type and the validation of the registry, this is not entirely so. Certain GMF-specific calls can be found in the code which shall have to be altered. And in the case that a different programming language than Java is used, the current source code cannot be used at all. The overall approach that is taken however, can be reused. The construction of a DJ-graph to detect reducible and irreducible loops, for example, was found to be very effective. The article “Identifying Loops Using DJ Graphs” by Vugranam et. al. [45] is a very useful source for this.

5. **Redesign 2R01**

The 2R01 building block is currently responsible for retrieving transaction types from the database and executing them (see Section 6.3.1). Before being able to do this for transaction types designed according to the DSL, the 2R01 building block needs to be redesigned. For example, it has to take into account conditional paths and loops, the new database model and the evaluation of the Exclusive Choice conditions.
The main focus of this thesis was the development a Domain Specific Language for the visual modeling of transaction types using the standard building blocks. This DSL supports the modeling of conditional paths and loops; constructs that are not available in the current situation. Before arriving at this point however, several related issues were addressed.

It had been identified that building blocks contain too much functionality. As a result of which they became harder to compose, which increased the need for custom building blocks. To counteract this problem, five granularity guidelines were presented that aim to reduce building block granularity as a result of which the flexibility and reusability of building blocks is improved.

A consequence of the guidelines is that the building block collection will contain more and smaller building blocks. Due to the lack of localization mechanisms this increases the chances that users cannot find the required building block in a reasonable time. To speed up the localization of building blocks and the modeling of transaction types in general, two mechanisms have been introduced. An integrated search function that works on the basis of building block’s filenames, tags and active registry settings makes sure that required building blocks can be found quickly. Templates are meant to speed up the modeling of an entire transaction type by allowing the user to make use of default transaction types. These defaults can be altered to fit specific customer needs.

Another consequence of the granularity guidelines is that it requires a larger number of building blocks to reach the same level of functionality. To prevent the user from having to arrange numerous small building blocks, the DSL supports container functionality. By grouping coherent building blocks in a container, the ease of assembly of the large building blocks is maintained, whilst having the individual parts of the building blocks available as well.

A newly developed Automated Dependency Checker ensures that for all building blocks in a transaction type the required registry settings are available at the time the building block is executed. Hereby freeing the user from the need for extensive testing.

To make use of an ADC possible, it was required to include additional building block information in the database. Which already required changes to the way transaction types are stored in order to support conditional paths and loops. A new database model has been designed that includes all the information necessary for the DSL and ADC to function.

The above described concepts were combined in a proof-of-concept that demonstrates that using a graphical development environment for the modeling and validation of transaction types is feasible and provides benefit compared to the current situation. Moreover, it demonstrated that the developed DSL is a suitable way to achieve this.
Although the DSL and ADC are only suitable for internal use at Zetes, the general idea of this thesis is applicable to other domains as well. Other domains that require visual modeling will likely benefit by designing a suitable DSL and may find inspiration in this thesis. For Zetes the DSL means going from having limited expressiveness to being able to express a wider range of transaction types in a more easily, less error-prone way.
References


Feature Oriented Domain Analysis: Feasibility Study, Carnegie Mellon University, November  
1990


[35] Rich Rollman, XML Tips and Tricks: Determining whether XML is appropriate,  


2010

[38] Gabor Karsai, Holger Krahn, Class Pinkernell, Bernhard Rumpe, Martin Schindler and Steven  
Vökel, “Design Guidelines for Domain Specific Languages”, In 9th OOPSLA Workshop on  
Domain-Specific Modeling, 2009

International Conference on System Sciences, 2002

[40] Marian Petre, “Why Looking Isn’t Always Seeing: Readership Skill and Graphical  

[41] Hans Grönniger, Holger Krahn, Bernhard Rumpe Martin Schindler and Steven Vökel, Text-  
based Modeling, Technische Universität Braunschweig, Germany

[42] Alf Inge Wang, Experience Paper: Using XML to implement a workflow tool, Department of  
Computer and Information Science, Norwegian University of Science and Technology,  
November 2002

[43] Tomaž Lukman and Marjan Mernik, “Model-Driven Engineering and its introduction with  
metamodeling tools”, In 9th International PhD Workshop on Systems and Control, October 2008

[44] Sanna Sivonen, “Domain-specific modelling language and code generator for developing  
repository-based Eclipse plug-ins”, In VTT Publications 680, 2008

Graphs”, In ACM Transactions on Programming Languages and Systems, Vol. 18, No. 6, pages  
649-658, November 1996
# Appendices

## Appendix A: Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2R01</td>
<td>2R01 is the identification number of the building block that is responsible for the execution of transaction types.</td>
</tr>
<tr>
<td>Automated Dependency Checker</td>
<td>The mechanism responsible for checking, during modeling-time, if the building block dependencies are satisfied.</td>
</tr>
<tr>
<td>Building Block</td>
<td>Building blocks are the executable components of a transaction type.</td>
</tr>
<tr>
<td>Cohesion</td>
<td>The extent to which classes of a component are interrelated. [3]</td>
</tr>
<tr>
<td>Component-Based Software Engineering/Development (CBSE/CBSD)</td>
<td>CBSDO advocates the acquisition, adaptation and integration of reusable software components, including commercial-off-the-shelf (COTS) products, to rapidly develop and deploy complex software systems with minimum engineering effort and resource cost. [5]</td>
</tr>
<tr>
<td>Coupling</td>
<td>The extent to which classes within a component relate to the other classes, which are not in that component. [3]</td>
</tr>
<tr>
<td>Customization</td>
<td>The ability to fit and alter solutions based on commonalities and variability across customers.</td>
</tr>
<tr>
<td>Cyclomatic Complexity</td>
<td>A complexity measure; it equals the number of linearly independent paths through the source code. [14]</td>
</tr>
<tr>
<td>Ease of Assembly</td>
<td>The ease with which components can be assembled.</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Efficiency is the quality of completing a given number of work items in satisfactory, according to some measure, time frame.</td>
</tr>
<tr>
<td>Functionality</td>
<td>The totality of essential functions that the software product provides.</td>
</tr>
<tr>
<td>Graphical Modeling Framework</td>
<td>The meta-modeling tool used for the implementation of the proof-of-concept.</td>
</tr>
<tr>
<td>Granularity</td>
<td>The extent to which a system is broken down into small parts.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>The ease with which software features can be added, removed or modified.</td>
</tr>
<tr>
<td>Medea PC-application</td>
<td>The application used by Zetes for the modeling of transaction types.</td>
</tr>
<tr>
<td>Medea Platform</td>
<td>A hard- and software combination used for the development and execution of logistical process support.</td>
</tr>
<tr>
<td>Performance</td>
<td>The number of steps needed by a methodology, an algorithm, to find a solution.</td>
</tr>
<tr>
<td>Portability</td>
<td>Refers to how well the software can adopt to changes in its environment or with its requirements.</td>
</tr>
<tr>
<td>Reliability</td>
<td>Characterizes the capability of the system to maintain its service provision under defined conditions for defined periods of time.</td>
</tr>
<tr>
<td>Reusability</td>
<td>The extent to which a component can be (re)used in developing various business applications. [3]</td>
</tr>
<tr>
<td><strong>RF-Terminal</strong></td>
<td>The handheld devices used by warehouse personnel on which transaction types are executed.</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>The ability to either handle growing amounts of work in a graceful manner or to be enlarged.</td>
</tr>
<tr>
<td><strong>Service Oriented Architecture/Computing (SOA/SOC)</strong></td>
<td>SOA presents well-defined business functions as services, which are made available to multiple applications through standard protocols. Using SOA, institutions can integrate business functions into new and interesting applications.</td>
</tr>
<tr>
<td><strong>Transaction Type</strong></td>
<td>Transaction types represent processes to support warehouse management.</td>
</tr>
<tr>
<td><strong>Usability</strong></td>
<td>Usability only exists with regard to functionality and refers to the ease of use and learnability for a given function.</td>
</tr>
</tbody>
</table>
Appendix B: Interview Scheme 1

Vragenlijst bouwblokken & transactie constructie

Doel: een beeld krijgen over de huidige situatie en problematiek rondom het construeren van transacties met behulp van de standaard bouwblokken.

Naam:………………………………………………………

Functie:…………………………………………………….

Ervaring met Medea:…………………………………..jaar

1. Wat is uw algemene mening over Medea en de manier van het construeren van transacties?

2. Hoe gaat u te werk bij het construeren van transacties? Vanaf nul of op basis van bestaande transacties?

3. Welke moeilijkheden en/of beperkingen ervaart u tijdens de constructie van transacties?

4. Op welke manier zouden eerdergenoemde moeilijkheden en/of beperkingen opgelost kunnen worden?

5. Welke opties/mogelijkheden zou u willen zien om de constructie van transacties te vereenvoudigen?

6. Wat vindt u van de huidige structuur, of het gebrek daaraan, van de bouwblokken? Is er reeds een impliciete structuur?

7. Wat zou u graag zien om de structuur van de bouwblokken te verbeteren/verduidelijken?

8. Hoe wordt op het moment bepaald welke bouwblokken op elkaar aansluiten en welke niet? Hoe worden de dependencies bepaald? En op welk moment?


10. Hoe zit de implementatie van het geheel (in grote lijnen) in elkaar? Gebruikte talen, software, representatie, opslag, etc.
Appendix C: Interview Scheme 1 – Interview results

Resultaat interviews bouwblokken & transactie constructie (gecombineerd)

Geïnterviewde medewerkers:

<table>
<thead>
<tr>
<th>Naam</th>
<th>Functie</th>
<th>Ervaring met Medea</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charles van Liempd</td>
<td>System Architect</td>
<td>+/- 4 jaar</td>
</tr>
<tr>
<td>Harm Donkers</td>
<td>System Architect</td>
<td>+/- 4 jaar</td>
</tr>
<tr>
<td>Niki Schuitvlot</td>
<td>System Architect</td>
<td>+/- 9 jaar</td>
</tr>
</tbody>
</table>

1. Wat is uw algemene mening over Medea en de manier van het construeren van transactie types?
   - Bewerkelijk (zeker zonder ervaring)
   - Veel zoekwerk vereist voor het vinden van de juiste bouwblokken en de vereisten voor bouwblokken
   - Steile learning curve

2. Hoe gaat u te werk bij het construeren van transactie types? Vanaf nul of op basis van bestaande transactie types?
   - 80-90% van de transactie types worden gekopieerd van/zijn gebaseerd op bestaande transactie types
   - Verschillende klanten op gebied van magazijnbeheer vereisen grotendeels dezelfde transactie types
   - Indien een transactie type vanaf nul wordt opgebouwd:
     - Controleer of de benodigde bouwblokken aanwezig zijn
     - Controleer wat de varianten van de bouwblokken inhouden (in de documentatie)
     - Controleer of deze bouwblokken precies het juiste gedrag vertonen
       - Zo niet, construeer een variant
     - Construeer het transactie type
     - Test het transactie type

3. Welke moeilijkheden en/of beperkingen ervaart u tijdens de constructie van transacties?
   - De voorwaarden aan een bouwblok zijn enkel terug te vinden in de documentatie
   - Het is gedurende design-time onduidelijk of aan de voorwaarden van elk van de bouwblokken is voldaan; hier is geen software matige ondersteuning voor
   - Het ontwikkelen van transactietypes lijkt dynamisch
     - Echter beperken de bouwblokken de flexibiliteit
     - Voor kleine afwijkingen van de functionaliteit moet een nieuwe variant geschreven worden
     - Belangrijke oorzaak hiervan is dat de bouwblokken te groot zijn, i.e. teveel functionaliteit bevatten
   - Waardes opgeslagen in de active registry zijn geen afspiegeling van de objecten in de database
o Enkel een identifier wordt in de active registry opgeslagen van een object
o Dit vereist continue terugkoppeling naar de database
o Voor het aanvragen van specifieke informatie van een object moet een variant gemaakt worden die die informatie uit de database haalt

- Bouwblokken zijn slechts eenmaal te gebruiken binnen een transactie type
  - Dit vereist het kopiëren van bouwblokken of het aanpassen van de constructie
- Variant codes zijn de enige manier om invloed uit te oefenen op de variabelen in het active registry
- Foutmelding niet altijd specifiek over wat er fout is
  - Bij het ontbreken van een waarde in het active registry wordt slechts vermeld in welk bouwblok het fout gaat en niet welke waarde ontbreekt
  - In nieuwe bouwblokken is dit verbeterd
- Variant codes zijn niet consequent; de ene keer cijfers, de andere keer letters
  - Functionaliteit van de variant codes moet in de documentatie opgezocht worden
- Er is geen mogelijkheid voor conditionele paden of loops binnen een transactie type

4. Op welke manier zouden eerdergenoemde moeilijkheden en/of beperkingen opgelost kunnen worden?

  - Weergave van de voorwaarden aan een bouwblok in de ontwikkelomgeving
  - Softwarematige controle van de dependencies tussen bouwblokken; zijn alle vereiste active registry waarden aanwezig?
  - Splits bouwblokken met meerdere functionaliteiten op in meerdere aparte bouwblokken
    - Vb. Een bouwblok dat een article code inleest en daarna (afhankelijk van de variant code) wel of niet een controle uitvoert op deze code
    - Beter zou zijn om één bouwblok te hebben dat de code inleest en één bouwblok dat de controle uitoefent. Dit controle bouwblok kan dan naar inzicht wel of niet toegevoegd worden aan het transactie type
    - Hierdoor wordt automatisch ook het aantal varianten per bouwblok verminderd
  - Sla de volledige objecten op in het active registry in plaats van enkel de identifier
    - Via afzonderlijke bouwblokken kan specifieke informatie uit dit object opgevraagd worden
  - Maak het mogelijke bouwblokken meerdere malen te gebruiken binnen een transactie type
    - Dit is misschien implementatietechnisch niet mogelijk
  - Werk alle bouwblokken bij om meer specifieke foutmeldingen te geven
  - Geef een domeinlijst en de bijbehorende functionaliteit van varianten weer in de ontwikkelomgeving
  - Dwing de structuur van het construeren van transactie types af in de ontwikkelomgeving om fouten te voorkomen
  - Voeg de mogelijkheid tot conditionele paden en loops binnen een transactie type toe

5. Welke opties/mogelijkheden zou u willen zien om de constructie van transacties te vereenvoudigen?

  - Zie vraag 4

6. Wat vindt u van de huidige structuur, of het gebrek daaraan, van de bouwblokken? Is er reeds een impliciete structuur?
• Enige vorm van structuur is vanuit bestaande transacties; van hieruit kan worden bekeken welke bouwblokken vaak samen worden gebruikt
• Er is geen expliciete structuur aanwezig

7. Wat zou u graag zien om de structuur van de bouwblokken te verbeteren/verduidelijken?

• Een categorisering van de bouwblokken (lastig!)
  o Bouwblokken worden gebruikt in verschillende toepassingsgebieden; een bouwblok kan zowel gebruikt worden voor inkomende en uitgaande orders als bijvullen, verplaatsen, etc.
  o Categorisering op basis van entiteiten (article, location, database etc.) misschien mogelijk
  o Categorisering op basis van functionaliteit (user input, database access, scanning, etc.) misschien mogelijk
• Maak het mogelijk te zoeken binnen en te filteren en sorteren op bouwblokken mogelijk
• Door het selecteren van een active registry setting, alle bouwblokken weer te geven die die active registry setting enerzijds vereisen of anderzijds leveren

8. Hoe wordt op het moment bepaald welke bouwblokken op elkaar aansluiten en welke niet? Hoe worden de dependencies bepaald? En wanneer?

• Dependencies worden:
  o Opgezocht in de documentatie
  o Opgezocht in bestaande transacties
  o Gevonden na testen
• Er is geen foutdetectie gedurende het construeren van transactie types
  o Er zijn geen beperkingen voor het combineren van bouwblokken tot een transactie
  o Bij de uitvoering worden ontbrekende waarden in het active registry gedetecteerd


• Er zijn geen specifieke bouwblokken waar veel problemen mee zijn
  o Deze problemen zouden dan reeds opgelost zijn

10. Hoe zit de implementatie van het geheel (in grote lijnen) in elkaar? Gebruikte talen, software, representatie, opslag, 2R01, etc.

• Alle code is geschreven in Delphi met behulp van de ontwikkelomgeving Delphi
• De werking van bouwblok 2R01:
  o Login op userdatabase
  o Geef beschikbare RF-functies weer (op basis van user rechten)
  o Wacht op input (user kiest een functie)
  o Haal de bouwblokken op behorende bij deze functie
  o Voer de 'on-startup' bouwblokken uit
  o Voer standaard 'scan item' uit
Loop sequentieel door de bouwblokken heen
‘end’ bouwblok zorgt ervoor dat er een database commit plaatsvindt

- Een probleem met 2R01 is dat het standaard ‘scan item’ uitvoert; dit is niet altijd vereist
- Bepaalde bouwblokken zorgen ervoor dat ‘scan item’ ook gebruikt kan worden voor locaties e.d.
- Logischer zou zijn om ‘scan item’ weg te halen uit 2R01 en dit een apart bouwblok te maken
- Dit geldt ook voor het registreren van de transactie op het einde van 2R01
- Dit zou van 2R01 een puur framework maken dat slechts de bouwblokken in het transactietype uitvoert
- Dit zou ook de noodzaak voor ‘on startup’ bouwblokken weghalen
Appendix D: Interview Scheme 2

Vragenlijst classificatie

Doel: er wordt gezocht naar een manier om het vinden van het juiste / gezochte bouwblok makkelijker te maken.

Naam: .................................................................

Functie:..............................................................

Ervaring met Medea:.............................................jaar

11. Wat is uw huidige manier van zoeken om een bepaald bouwblok te vinden? Bijvoorbeeld, een bouwblok dat een bepaald active registry setting levert.

12. Welke problemen ondervindt u bij deze manier van werken?

13. Heeft u ideeën en/of wensen om dit proces te vereenvoudigen?

14. In hoeverre zou een zoekfunctie op namen van bouwblokken toegevoegde waarde hebben? Waarom wel/niet?

15. In hoeverre zou een zoekfunctie op geleverde en vereiste active registry settings van bouwblokken toegevoegde waarde hebben? Waarom wel/niet?

16. Wat vindt u van een recommender systeem (op basis van samenhangende bouwblokken)? Nuttigheid? Haalbaarheid?

17. Ziet u andere manieren om de bouwblokken te ordenen / classificeren / structureren?
Appendix E: Interview Scheme 2 – Interview results

Resultaat interviews classificatie (gecombineerd)

Geïnterviewde medewerkers:

Naam: Pascal Kock
Functie: System Architect
Ervaring met Medea: +/- 9 jaar

Naam: Rudi Sengers
Functie: System Architect
Ervaring met Medea: +/- 2 jaar

18. Wat is uw huidige manier van zoeken om een bepaald bouwblok te vinden? Bijvoorbeeld, een bouwblok dat een bepaald active registry setting levert.

   1. Op basis van ervaring, kennis van de bouwblokken
   2. Collega’s raadplegen
   3. Zoeken op relevante termen binnen bestandsnamen
   4. Zoeken in documentatie

19. Welke problemen ondervindt u bij deze manier van werken?

   - Geen ideale situatie
   - Tijdrowend

20. Heeft u ideeën en/of wensen om dit proces te vereenvoudigen?

    Er zouden categorieën aan de bouwblokken moeten worden gegeven. Op basis van:
    a. RF
    b. Voice
    c. Mobility
    En hierbinnen op bijvoorbeeld procestype:
    o Picking
    o Staging
    o Inventory
    o Inslag
    o Uitslag
    o Etc.

Het probleem hierbij is dat wanneer de bouwblokken fijnmaziger worden, elk bouwblok inzetbaar is bij veel verschillende proceestypes wat de classificatie doet verwateren. Overlap.
Vergelijkbaar met het bovenstaande zou er een treestructure moeten worden gemaakt, waarbij door dieper in de tree te gaan steeds een steeds specifieker verzameling bouwblokken moet overblijven. Onduidelijk is echter op basis van wat deze treestructure zou moeten worden geconstrueerd.

Een andere mogelijkheid is om tags te hangen aan bouwblokken waarop gezocht kan worden. Met behulp van tags kun je de verschillende raakvlakken van het bouwblok vastleggen (i.e. de acties die het uitvoert, op welke entiteiten, binnen welke proceстypes, rekening houdend met verschillende vormen van naamgeving).

De 1e stap omvat sowieso het vastleggen van de kenmerken (functionaliteit, active registry settings, tags, etc.) van een bouwblok. Op basis hiervan kan gezocht / geclassificeerd worden.

De documentatie zou bondiger moeten zijn. Zodat in een oogopslag duidelijk is wat de functionaliteit is en welke active registry settings worden gemanipuleerd.

21. In hoeverre zou een zoekfunctie op namen van bouwblokken toegevoegde waarde hebben? Waarom wel/niet?

Nuttig, maar het probleem hierbij is dat de ge bruiker de naamgeving van het gezochte bouwblok moet weten. Naamgeving is niet overal uniform; hier zijn geen regels voor, maar slechts gebruiken. Soms zijn zelfs niet alle namen in het Engels.

Regelgeving hieromtrent zou dit slechts deels kunnen oplossen; sommige bouwblokken bevatten meer functionaliteit dan is vast te leggen in de naam. Tags zijn wat dat betreft meer geschikt; meerdere tags mogelijk voor functionaliteit, meerdere tags mogelijk voor verschil in naamgeving.

22. In hoeverre zou een zoekfunctie op geleverde en vereiste active registry settings van bouwblokken toegevoegde waarde hebben? Waarom wel/niet?

Nuttig, belangrijk is de mogelijk om specifiek te zoeken op verplicht, optionele, vereiste en geleverde active registry settings.

23. Wat vindt u van een recommender systeem (op basis van samenhangende bouwblokken)? Nuttigheid? Haalbaarheid?

Nuttig. Een soortgelijke mogelijkheid zo zijn om op basis van een geselecteerd bouwblok A, een lijst van bouwblokken weer te geven die mogelijk active registry settings leveren die vereist worden door A. Zelfde verhaal voor active registry settings die geleverd worden door A.

24. Ziet u andere manieren om de bouwblokken te ordenen / classificeren / structureren?

Samenvattend, een algemene zoekfunctie die naar de zoekterm zoekt binnen de bestandsnamen, functiebeschrijving, active registry settings, tags, etc. Waarbij de resultaten ook op basis hiervan geordend worden (vergelijkbaar met zoeken binnen Windows 7; de zoekterm wordt gezocht binnen muziek, bestanden, foto’s, internetpagina’s, etc.).

En daarnaast de mogelijkheid om specifiek op één van bovenstaande gebieden te zoeken.

Verder, blijkt uit ervaring dat klanten niet bezig zijn met het modelleren van transactie typen en hierin ook niet geïnteresseerd zijn.
Appendix F: Thesis Presentation Survey

Name: ........................
Function: ........................

1. Goal
Below a survey is presented concerning the thesis presentation at Zetes of May 14th. The goal of this survey is to determine the usefulness and feasibility of several of the presented topics. Namely, the granularity guidelines, the localization mechanisms and the visual modeling and validation of transaction types, i.e. the proof-of-concept. The results of this survey shall be used as part of the validation of these topics.

Wherever a multiple-choice question is given, only a single answer is allowed. For open questions, please provide a short and concise answer. In case you are unsure of an answer do not provide one.

2. Granularity Guidelines
During the presentation five granularity guidelines were presented. As a reminder, these were the presented guidelines:

1. Construct separate building blocks for multi-choice parameters
2. Split up building blocks that contain control flow logic
3. Encapsulate parameter functionality in separate building blocks where possible
4. Examine building blocks with a high cyclomatic complexity
5. Aim to capture overlapping functionality in separate building blocks

A more detailed description of these guidelines can be found after the survey questions at the end of this document [not included in this appendix]. These guidelines are aimed at reducing building block granularity and improving the clarity of the building block’s functionality.

First, for each guideline a few specific questions are asked, followed by some general questions on the guidelines.

2.1. Guideline specifics
1. Construct separate building blocks for multi-choice parameters

Is it clear how this guideline should be applied?

☐ Not at all  ☐ Not really  ☐ Somewhat  ☐ Very  ☐ Completely

Looking at the building block collection, how often do you estimate that this guideline is applicable?

☐ Unknown  ☐ 0  ☐ 1-10  ☐ 11-20  ☐ > 20
Do you foresee any problems applying this guideline to the building block collection? If so, explain which ones.

………………

Looking at a building block for which this guideline is applicable, how do you rate the usefulness of this guideline in reducing the building block’s granularity?

☐ Useless  ☐ Not really useful  ☐ Somewhat useful  ☐ Useful  ☐ Very useful

Looking at a building block for which this guideline is applicable, how do you rate the usefulness of this guideline in improving the clarity of the building block’s functionality?

☐ Useless  ☐ Not really useful  ☐ Somewhat useful  ☐ Useful  ☐ Very useful

2. Split up building blocks that contain control flow logic

Is it clear how this guideline should be applied?

☐ Not at all  ☐ Not really  ☐ Somewhat  ☐ Very  ☐ Completely

Looking at the building block collection, how often do you estimate that this guideline is applicable?

☐ Unknown  ☐ 0  ☐ 1-10  ☐ 11-20  ☐ > 20

Do you foresee any problems applying this guideline to the building block collection? If so, explain which ones.

………………

Looking at a building block for which this guideline is applicable, how do you rate the usefulness of this guideline in reducing the building block’s granularity?

☐ Useless  ☐ Not really useful  ☐ Somewhat useful  ☐ Useful  ☐ Very useful

Looking at a building block for which this guideline is applicable, how do you rate the usefulness of this guideline in improving the clarity of the building block’s functionality?

☐ Useless  ☐ Not really useful  ☐ Somewhat useful  ☐ Useful  ☐ Very useful

3. Encapsulate parameter functionality in separate building blocks where possible

Is it clear how this guideline should be applied?

☐ Not at all  ☐ Not really  ☐ Somewhat  ☐ Very  ☐ Completely

Looking at the building block collection, how often do you estimate that this guideline is applicable?

☐ Unknown  ☐ 0  ☐ 1-10  ☐ 11-20  ☐ > 20
Do you foresee any problems applying this guideline to the building block collection? If so, explain which ones.

Looking at a building block for which this guideline is applicable, how do you rate the usefulness of this guideline in reducing the building block’s granularity?

☐ Useless   ☐ Not really useful   ☐ Somewhat useful   ☐ Useful   ☐ Very useful

Looking at a building block for which this guideline is applicable, how do you rate the usefulness of this guideline in improving the clarity of the building block’s functionality?

☐ Useless   ☐ Not really useful   ☐ Somewhat useful   ☐ Useful   ☐ Very useful

4. Examine building blocks with a high cyclomatic complexity

Is it clear how this guideline should be applied?

☐ Not at all   ☐ Not really   ☐ Somewhat   ☐ Very   ☐ Completely

Looking at the building block collection, how often do you estimate that this guideline is applicable?

☐ Unknown   ☐ 0   ☐ 1-10   ☐ 11-20   ☐ > 20

Do you foresee any problems applying this guideline to the building block collection? If so, explain which ones.

Looking at a building block for which this guideline is applicable, how do you rate the usefulness of this guideline in reducing the building block’s granularity?

☐ Useless   ☐ Not really useful   ☐ Somewhat useful   ☐ Useful   ☐ Very useful

Looking at a building block for which this guideline is applicable, how do you rate the usefulness of this guideline in improving the clarity of the building block’s functionality?

☐ Useless   ☐ Not really useful   ☐ Somewhat useful   ☐ Useful   ☐ Very useful

5. Aim to capture overlapping functionality in separate building blocks

Is it clear how this guideline should be applied?

☐ Not at all   ☐ Not really   ☐ Somewhat   ☐ Very   ☐ Completely

Looking at the building block collection, how often do you estimate that this guideline is applicable?

☐ Unknown   ☐ 0   ☐ 1-10   ☐ 11-20   ☐ > 20
Do you foresee any problems applying this guideline to the building block collection? If so, explain which ones.

Looking at a building block for which this guideline is applicable, how do you rate the usefulness of this guideline in reducing the building block’s granularity?

☐ Useless  ☐ Not really useful  ☐ Somewhat useful  ☐ Useful  ☐ Very useful

Looking at a building block for which this guideline is applicable, how do you rate the usefulness of this guideline in improving the clarity of the building block’s functionality?

☐ Useless  ☐ Not really useful  ☐ Somewhat useful  ☐ Useful  ☐ Very useful

2.2. General
Do you think that additional guidelines are necessary to further reduce the building block granularity? If so, provide an example if possible.

Any additional remarks / comments concerning the granularity guidelines can be included below.

3. Localization
It was identified that mechanisms are needed to speed up the modeling of a transaction type; primarily by new mechanisms for localizing building blocks. For this purpose, a search function and templates were presented. As a reminder, a short description of both techniques can be found after the survey questions in Appendix B. [not included in this appendix]. Similar as before, first questions specific for each technique are given, followed by general questions concerning building block localization.

3.1. Search function
Is the concept of the search function clear?

☐ Not at all  ☐ Not really  ☐ Somewhat  ☐ Very  ☐ Completely

Do you foresee any problems for the implementation of the search function? If so, explain which ones.

How do you rate the usefulness of this technique in localizing required building blocks?

☐ Useless  ☐ Not really useful  ☐ Somewhat useful  ☐ Useful  ☐ Very useful

How do rate the usefulness of this technique in speeding up the modeling of transaction types?

☐ Useless  ☐ Not really useful  ☐ Somewhat useful  ☐ Useful  ☐ Very useful
3.2. Templates
Is the concept of the templates clear?

☐ Not at all  ☐ Not really  ☐ Somewhat  ☐ Very  ☐ Completely

Do you foresee any problems for the implementation of the templates? If so, explain which ones.

………………

How do you rate the usefulness of this technique in localizing required building blocks?

☐ Useless  ☐ Not really useful  ☐ Somewhat useful  ☐ Useful  ☐ Very useful

How do you rate the usefulness of this technique in speeding up the modeling of transaction types?

☐ Useless  ☐ Not really useful  ☐ Somewhat useful  ☐ Useful  ☐ Very useful

3.3. General
Do you think that additional techniques are necessary to facilitate the localization of building blocks? If so, provide an example if possible.

………………

Any additional remarks / comments concerning building block localization can be included below.

………………

4. Proof-of-Concept
The proof-of-concept demonstrated the feasibility of the visual modeling and validation of transaction types. The modeling of transaction types was done according to the Domain Specific Language that has been developed. The DSL was specified using a UML meta-model, with corresponding constraints and a concrete syntax. This in total determines the graphical model elements and the way in which they can be composed.

First questions concerning the actual modeling of a transaction type using the proof-of-concept are given. After this, questions related to the automated dependency checker are presented.

4.1. Visual modeling of transaction types
Is it clear how the DSL can be used to model transaction types?

☐ Not at all  ☐ Not really  ☐ Somewhat  ☐ Very  ☐ Completely

How do you rate the amount of freedom for the user to model transaction types the way he sees fit?

☐ Very weak  ☐ Weak  ☐ Sufficient  ☐ Good  ☐ Very good

How do you rate the ease-of-use of the DSL concerning the way transaction types are modeled?

☐ Very weak  ☐ Weak  ☐ Sufficient  ☐ Good  ☐ Very good

How does the speed of modeling transaction types according to the DSL compare to modeling transaction types in the current situation? (Without taking into account the automated dependency checker)
Much slower  Slower  Equal  Faster  Much faster

How do rate the usefulness of the DSL for modeling transaction types?

Useless  Not really useful  Somewhat useful  Useful  Very useful

4.2. Automated dependency checker

Is it clear how the automated dependency checker can be used to validate building block dependencies?

Not at all  Not really  Somewhat  Very  Completely

How do you rate the clarity of the error indications?

Very weak  Weak  Sufficient  Good  Very good

How do you rate the ease-of-use of the automated dependency checker?

Very weak  Weak  Sufficient  Good  Very good

How do you rate the usefulness of the automated dependency checker in the process of modeling transaction types?

Useless  Not really useful  Somewhat useful  Useful  Very useful

4.3. General

Do you foresee any problems for the implementation of the full visual modeling environment? If so, explain which ones.

............... 

Any additional remarks / comments concerning the visual modeling and validation of transaction types can be included below.

............... 

Thank you for taking the time to fill out this survey!