



**AGH University of Science and Technology**

**Computer Science Laboratory**

Department of Automatics

Al. Mickiewicza 30

30-059 Kraków, POLAND

## **Analysis of UML Representation for XTT and ARD Rule Design Methods**

**Krzysztof Kluza**

*AGH University of Science and Technology*

*Department of Automatics*

*Kraków, POLAND*

*kluza@agh.edu.pl*

**Grzegorz J. Nalepa**

*AGH University of Science and Technology*

*Department of Automatics*

*Kraków, POLAND*

*gjn@agh.edu.pl*

*Published online: 10.12.2009*



## **Analysis of UML Representation for XTT and ARD Rule Design Methods\***

**Krzysztof Kluza**

*AGH University of Science and Technology*

*Department of Automatics*

*Kraków, POLAND*

*kluza@agh.edu.pl*

**Grzegorz J. Nalepa**

*AGH University of Science and Technology*

*Department of Automatics*

*Kraków, POLAND*

*gjn@agh.edu.pl*

**Abstract.** The report concerns the UML representation for XTT and ARD Rule Design Methods. In the report an overview of the Software and Knowledge Engineering background is given and the HeKatE methodology is explained. Furthermore, the UML representation for XTT and ARD representation is introduced and the metamodel for this representation is discussed. The translation between UML models, serialized to XMI, and the HeKatE representation is considered and suitable translation algorithms for XSL Transformations are presented.

**Keywords:** UML, MOF, XMI, visual design, rule-based systems, HeKatE

---

\*The paper is supported by the Hekate Project funded from 2007–2009 resources for science as a research project.

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Background . . . . .	4
1.2	The Goal of the Report . . . . .	4
1.3	Structure . . . . .	5
<b>2</b>	<b>Selected Aspects of Software Engineering</b>	<b>5</b>
2.1	Software Engineering Process . . . . .	5
2.2	UML . . . . .	6
2.2.1	Origins of UML . . . . .	6
2.2.2	Models and Diagrams . . . . .	6
2.2.3	Use of UML . . . . .	8
2.3	OCL . . . . .	8
2.4	MOF . . . . .	10
2.4.1	Metamodel . . . . .	10
2.4.2	MOF standard . . . . .	10
2.4.3	MOF Layers . . . . .	11
2.5	MDA approach . . . . .	12
2.5.1	Models in MDA . . . . .	12
2.5.2	MDA Layers . . . . .	12
2.5.3	Transformations . . . . .	13
2.6	Business Rules . . . . .	14
<b>3</b>	<b>Knowledge in the HeKatE Design Process</b>	<b>15</b>
3.1	Knowledge and Software Engineering . . . . .	15
3.1.1	UML in Knowledge Engineering . . . . .	16
3.2	Hybrid Knowledge Engineering . . . . .	17
3.2.1	HeKatE Phases . . . . .	17
3.2.2	HaDEs . . . . .	18
3.3	Conceptual Design Phase . . . . .	19
3.3.1	Conceptual Modelling Using ARD . . . . .	19
3.3.2	ARD Syntax . . . . .	19
3.3.3	ARD Transformations . . . . .	19
3.3.4	Hierarchy of ARD . . . . .	20
3.3.5	ARD Tools . . . . .	20
3.4	Logical Design Phase . . . . .	21
3.4.1	EXtended Tabular Trees . . . . .	21
3.4.2	XTT Visual Representation . . . . .	21
3.4.3	Logical Design Phase . . . . .	22
3.5	Physical Design Phase . . . . .	22
3.6	HeKatE vs Software and Knowledge Engineering . . . . .	23
<b>4</b>	<b>UML Representation of ARD</b>	<b>23</b>
4.1	Preliminary Approaches of Modeling ARD in UML . . . . .	23
4.1.1	Approach with Activity Diagrams . . . . .	23
4.1.2	Approach with Component Diagrams . . . . .	23
4.2	The Proposed ARD Model . . . . .	24
4.2.1	Syntax and Semantics of Artifacts . . . . .	24
4.2.2	UML Model . . . . .	25
4.2.3	Evaluation . . . . .	26
4.3	MOF Metamodel of the Proposed UML Model . . . . .	26

4.3.1	Metamodel for ARD Diagrams . . . . .	27
4.3.2	Metamodel for TPH Diagrams . . . . .	27
<b>5</b>	<b>UML Representation of XTT</b>	<b>28</b>
5.1	Preliminary Approaches of Modeling XTT in UML . . . . .	29
5.2	The Proposed XTT Model . . . . .	29
5.2.1	Used Artifacts . . . . .	30
5.2.2	UML Model . . . . .	30
5.2.3	Evaluation . . . . .	31
5.3	Metamodel of the Proposed Model . . . . .	32
5.3.1	Metamodel for XTT Diagrams . . . . .	32
<b>6</b>	<b>Model Translations</b>	<b>34</b>
6.1	XMI Representation . . . . .	34
6.1.1	Syntax of an XMI Document . . . . .	34
6.1.2	XMI Representation of UML Model for ARD . . . . .	35
6.1.3	XMI Representation of UML Model for XTT . . . . .	35
6.2	HeKatE Markup Language . . . . .	36
6.3	Translation Algorithms . . . . .	36
6.3.1	Translation Algorithm from XMI to ARD . . . . .	36
6.3.2	Translation Algorithm from ARD to XMI . . . . .	37
6.3.3	Translation Algorithm from XMI to XTT . . . . .	38
6.3.4	Translation Algorithm from XTT to XMI . . . . .	40
6.4	Implementation . . . . .	42
<b>7</b>	<b>Closing remarks</b>	<b>42</b>
7.1	Summary . . . . .	42
7.2	Future Work . . . . .	43
<b>A</b>	<b>OCL Constraints for ARD - Appendix</b>	<b>50</b>
A.1	OCL Constraints for the Metamodel of ARD Diagrams . . . . .	50
A.2	OCL Constraints for the Metamodel of TPH Diagrams . . . . .	50
<b>B</b>	<b>OCL Constraints for XTT - Appendix</b>	<b>52</b>
B.1	OCL Constraints for the Metamodel of XTT Diagrams at the Lower Level of Abstraction	52
B.2	OCL Constraints for the Metamodel of XTT Diagrams at the Higher Level of Abstraction . . . . .	54
<b>C</b>	<b>Thermostat Case Study - Appendix</b>	<b>58</b>

# 1 Introduction

## 1.1 Background

There is an old Chinese proverb "A picture is worth a thousand words.". Indeed, pictures are meaningful for people. A single visualization can replace a long description. A visual representation is superior, because using it one can quickly absorb huge amounts of data, and it is easier to grasp relationships in data.

Nowadays, software is becoming more complex. It is difficult to describe it only textually. However, its substance can be visualized. Visual modeling is already an essential part of the Software Engineering process [64]. UML (the Unified Modeling Language) [58] is a graphical modeling language. It has become the dominant graphical notation for software modeling. It uses diagrams to visualize complex software. Therefore, it could be useful in solving these problems.

In Knowledge Engineering the key problem is the knowledge representation [66]. The rule-based approach [14] is one of the most common knowledge representations. Rules are intuitive and easy to understand for humans. Each rule represents a small piece of knowledge. Therefore, it can be easily added or subtracted from a larger knowledge base.

The design of large knowledge bases is non trivial neither. A visual representation, such as the decision tables [66], allows to deal with this complexity. There are also ongoing efforts to develop a UML-based representation for rules [22, 57]. However, there has been no coherent proposal so far.

## 1.2 The Goal of the Report

This report is dedicated to the presentation and summary of the results of the thesis [29] and related papers concerning UML representation for the XTT rule design method [43, 30] and MOF-based metamodel for this representation [31]. XTT (EXtended Tabular Trees) [44] is a hybrid knowledge representation and design method, aims at combining decision trees and decision tables. It has been developed in the HeKatE (Hybrid Knowledge Engineering) [49, 41] research project that regards Software Engineering based on Knowledge Engineering.

The major issue in the HeKatE methodology is that it requires dedicated tools to design systems. Such tools have been developed, but this approach makes users use only this software. There was a need to allow users a wider choice of tools, especially if they have already used certain tools. This is possible if some popular visual language is used (e.g. UML), instead of the original ARD and XTT notations.

The UML representation of ARD and XTT supports the separation of the conceptual and logical design, which is assumed in the HeKatE process. The representation describes the structured rulebase, and is both transparent and hierarchical. As the chosen language of the representation is UML, the whole process can be supported by many tools. Moreover, the HeKatE methodology uses a declarative model specification and UML also is a declarative language, so this provides paradigm compatibility.

Some approaches of visual rule representation have already been described in [30]. One of them is [rwg2009www-urml](http://www.urml.org) [33], which extends UML syntax and semantics. It is also possible to redefine semantics of UML using custom UML profiles. However, using representation which extends UML syntax one is not able to use existing UML tools, and profiles, in some cases, can redefine the semantics so much, that the model can be incomprehensible.

The representation discussed in the report does not introduce new UML artifacts, but is based on the original UML syntax and use the original language efficiently. UML models can be formally defined using a narrowed UML metamodel. Such a metamodel allows to develop tools for validating UML models of XTT. However, the raw UML diagrams (which are just graphics) can not be easily processed. They have to be serializable to XMI, which is their XML-based representation. XMI is (in theory) portable between various UML tools supporting XMI serialization. Such models, serialized to XMI, can be transformed into a custom XTT representation. This transformation can be done using

translators based on proposed algorithms.

### 1.3 Structure

This report is organised as follows:

- The Section 2 introduces the background of Software Engineering. Basic concepts of software modeling in Software Engineering (UML, OCL, MOF, MDA and BR) are introduced.
- The Section 3 describes the Hybrid Knowledge Engineering Project (HeKatE) and its relation to Software Engineering. The main emphasis is on the description of the XTT and ARD design methodology.
- The Section 4 presents a UML representation of ARD and its MOF-based metamodel.
- The Section 5 contains the description of a UML representation of XTT and its MOF-based metamodel.
- The Section 6 describes the XMI representation of the models described in the previous two sections and provides information about the model translations.
- The Section 7 summarizes the main contributions of this report and discusses open issues and future work.
- The appendix contains items which for the quantitative reasons could not be included in the main part of the report, such as OCL expressions and additional information, and the case study of a UML representation for the selected XTT thermostat case.

## 2 Selected Aspects of Software Engineering

### 2.1 Software Engineering Process

Generally speaking, Software Engineering (SE) deals with construction of large programs. The aim of Software Engineering is to produce effectively high quality software. sommerville:2004 in [64] defines Software Engineering through software production. It is concerned with all aspects of software production. Thus, it is concerned with both the technical process of software development and all activities around this.

Therefore, Software Engineering is not only about the software itself. It is worth emphasising that Software Engineering also includes the concerns the process of software manufacturing itself. Software systems are very complex and so is the process of manufacturing them. So, teamwork is necessary for such projects. And wherever there is a team, there is a need for management [3].

Software development consists of several different phases. The sequence of them is called the software life cycle [16]. There are many various life cycle models. The best known is the traditional model, named the *waterfall model*. This model, shown in Fig. 1, assumes a standard sequence of tasks. There are no feedbacks in this model, except feedbacks to direct previous phase.

Other models are usually, more or less, modifications of the traditional model. For example, the *prototyping model* [3] builds a throwaway version (or prototype) of the system, to test concepts and requirements. After customer agreement to such a system, the software development usually follows the traditional model.

Another type of life cycle model is the *spiral model* introduced by boehm1998 [4]. It is a combination of the *prototyping model* and the *waterfall model*. Its aim is to combine advantages of both concepts. The model is shown in Fig. 2. It is a spiral that starts in the middle and continually revisits the basic phases, including communication with customer.

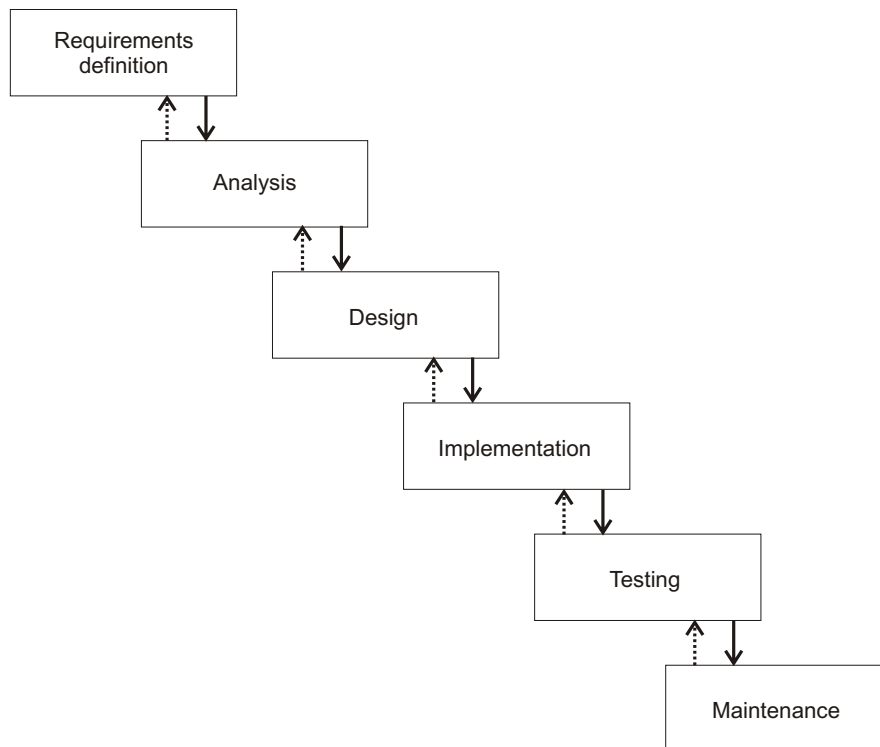


Figure 1: Waterfall life cycle model – based on [64]

## 2.2 UML

When it comes to the practical software design, the Unified Modeling Language (UML) is *de facto* the standard for modeling software applications [21].

UML attempts to provide diagrams to capture requirements (use case diagrams), collaboration between parts of software that realize them (collaboration diagrams), the realization itself (sequence or statechart diagrams) and diagrams which show how everything fits together and is executed (component and deployment diagrams) [60].

### 2.2.1 Origins of UML

UML was introduced in the nineties as an incorporation of the best of the notations in use at the time (the Booch Method [21] by Grady Booch, the Object Modeling Technique [21] coauthored by James Rumbaugh, and Objectory by Ivar Jacobson), attempting to be a unifying notation. The influence of the different notations is shown in Fig. 3 [21].

In 1997, UML was released as UML version 1.1. With version 2.0, UML has evolved. Number of the original diagram types have been retained and extended. Furthermore, some new diagram types have been introduced [17].

However, the fundamental difference between UML 1 and UML 2.X is the enhancement of definitions of its abstract syntax rules and semantics. The language has a modular structure. The formal definition of the language is described in the UML Superstructure document [59]. There is also the Diagram Interchange Specification [56] which provides a way to share UML models between different modeling tools.

### 2.2.2 Models and Diagrams

Although UML is intrinsically relevant to diagrams, modeling is not just about diagrams. It is about capturing a system as a set of models [60].



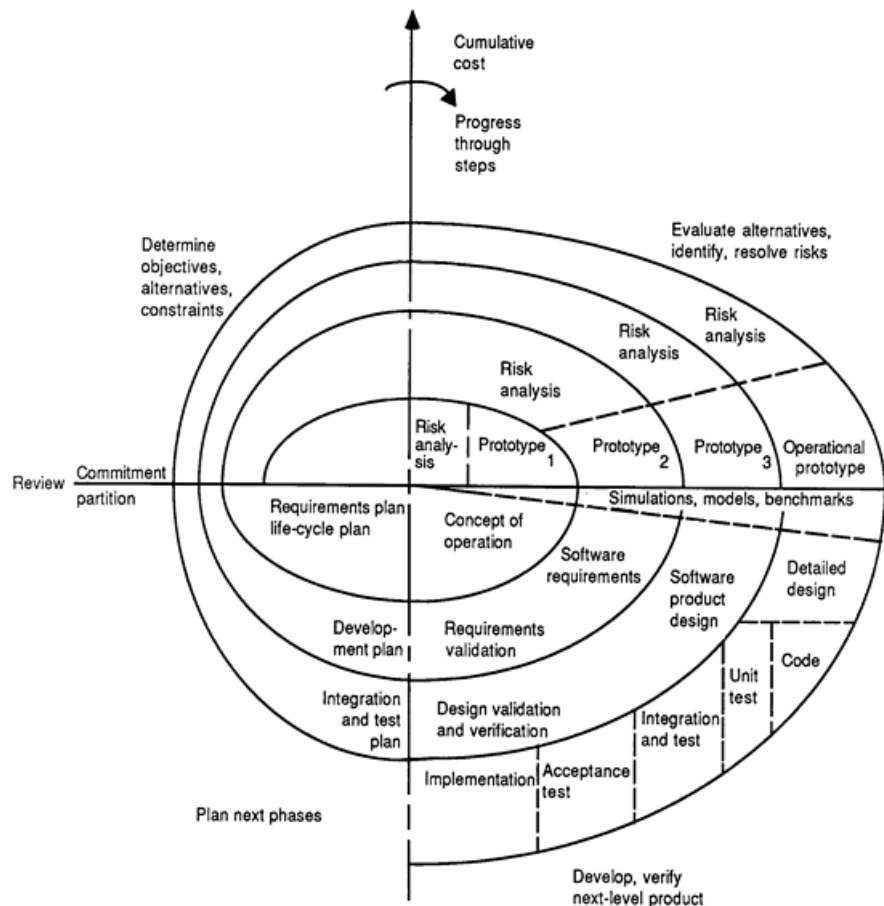


Figure 2: Spiral life cycle model [4]

There is no single definition of a model [28]. A model may mean many different things because there are many different types of models. But most of the definitions have something in common. They emphasize that a model is an abstraction. It differs from the thing it models. The thing exists in reality while a model is its abstraction. The feature of a model is that it can be used to produce things existing in reality. A model describing a system can be used to produce similar systems. Such systems are not the same, they might differ in details. It is so, because details are omitted in the model [28].

In UML, a model is a set of diagrams [28]. Such diagrams describe the system (or only a part of it). The complete system can be described by a number of models. Each one describes the system from a different perspective than another, often on another level of abstraction.

UML has incorporated some of the best of the mentioned notations. By design, each UML diagram should be consistent with any other diagram representing the same model. But inconsistency is highly likely to occur in models. It is so, because UML possesses a huge set of different artifacts and diagram types are very complex. In addition, as fowler2003:umldistilled remarked [9]:

*Diagram types are not particularly rigid. Often, you can legally use elements from one diagram type on another diagram. The UML standard indicates that certain elements are typically drawn on certain diagram types, but this is not a prescription.*

Today UML 2.0 defines thirteen types of diagrams divided into two main categories:

- *structure diagrams* – contain 6 types of diagrams representing the structure of a modeled application,
- *behavior diagrams* – contain 7 types of diagrams representing general types of behavior (including 4 that represent different aspects of interactions).

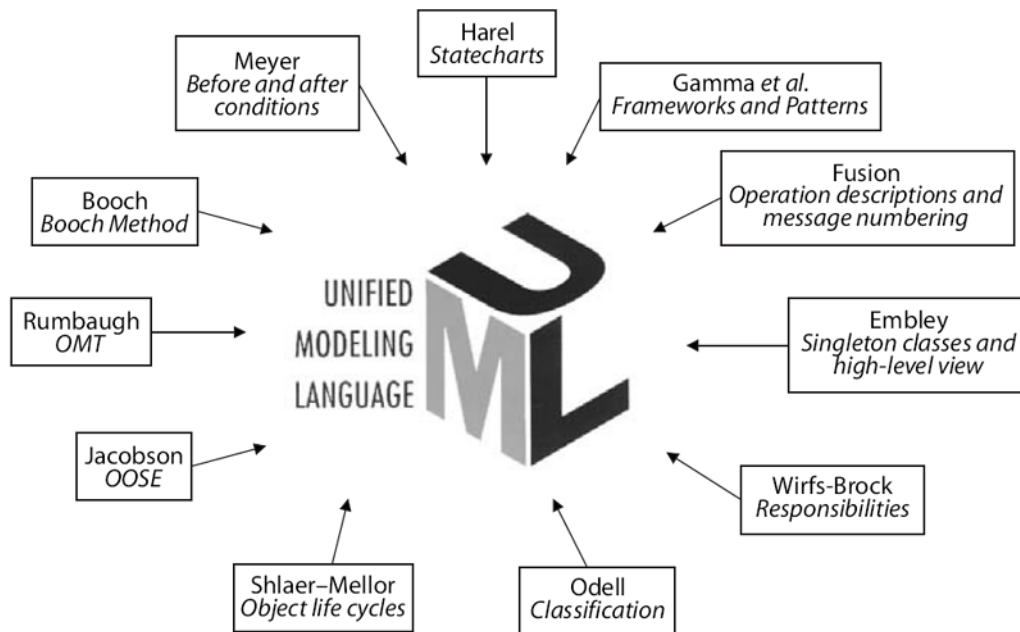


Figure 3: Influences on UML [21]

A hierarchically categorized tree showing the classification of UML diagram types is shown in Fig. 4 [9].

Unfortunately, there is no strict definition of the relationship between modeled behavior and structure, which is discussed by Nalepa and Wojnicki in [50].

Furthermore, in [9] there are some diagrams discussed that can help to define applications and have no UML equivalent, like a screen flow diagram. The decision tables are also mentioned.

### 2.2.3 Use of UML

UML itself is not a design method or a software process. It is only a notation which can be useful within a software process or designing. But even though it is only a notation, its use can be various. fowler2003:umldistilled describes UML as a different thing to different people. He gives three common ways in which people use UML [9]:

- as a sketch – to make simple sketches showing only key points of the model, this approach emphasizes selective communication rather than complete specification,
- as a blueprint – to provide a detailed specification of a system and a detailed design documentation for programmers to code up,
- as a programming language – to execute code, so it requires a complete model of the system.

Another issue is a methodology which indicates how to apply a design. UML itself does not require any specific method, but it is mostly used with an object-oriented design method.

## 2.3 OCL

UML diagrams are typically not detailed enough to provide for every aspect of a specification. Not every relevant aspect can be expressed in pure UML. One can describe some additional constraints in natural language, but it probably would be inaccurate and ambiguous.

The Object Constraint Language (OCL) [34] is a formal language. One can write expressions for UML models in it. It has been developed in order to avoid ambiguous constraint expressions.

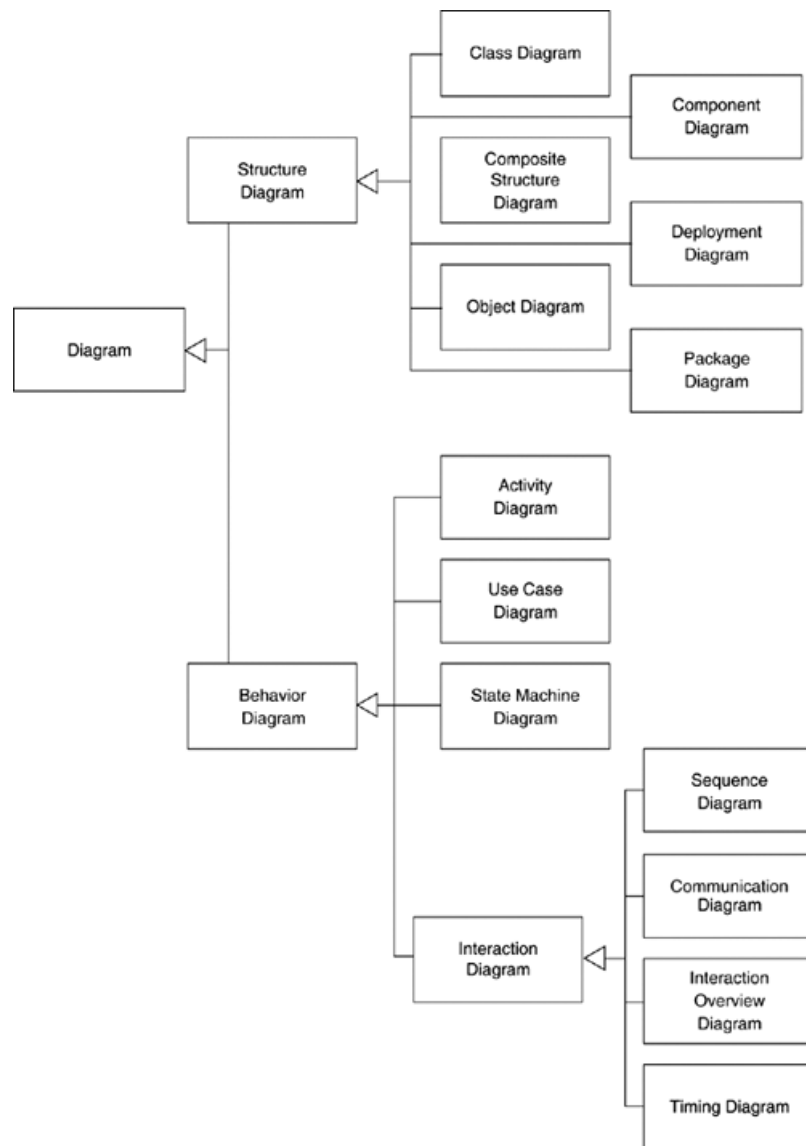


Figure 4: Hierarchy of UML diagram types [9]

Although formal languages mostly have complicated syntax, OCL attempts to be suitable both for business or system modelers and programmers. After all, it has been developed as a business modeling language within the IBM Insurance division [34].

OCL expressions are simple and easy to read and write. The language is used for writing expressions on elements in a model. Such an expression cannot have side effects and change anything in the model. It simply returns a value after being evaluated.

OCL specification gives many different purposes of using an OCL expression. It can be used to specify [34]:

- invariants on classes and types in the class model,
- type invariant for stereotypes,
- pre- and post conditions for operations,
- guards,
- target for messages and actions,

- constraints on operations,
- derivation rules for attributes for any expression over a UML model.

Constraints supported by OCL can be divided into four types [5]:

- invariants – constraints that must be true for all instances of a particular element,
- preconditions – constraints that must be true when the execution of a particular operation is about to begin,
- postconditions – constraints that must be true when the execution of an operation has just ended,
- guards – constraints that must be true before firing a state transition.

OCL provides some built-in types like integers, strings or several kinds of collections. But it also supports types constructed from the entities of a UML model. So, a modeler can introduce new types corresponding to diagram artifacts.

OCL expressions can be expressed on the UML diagram within braces (`{ }`) or notes. An exemplary OCL expression is shown in Fig. 5. They can also be presented separately. In that case, the context of a certain model element is needed. However, it does not mean that the expression can only refer to elements from that context. OCL allows to navigate between elements in the model, starting from the context of one of them.

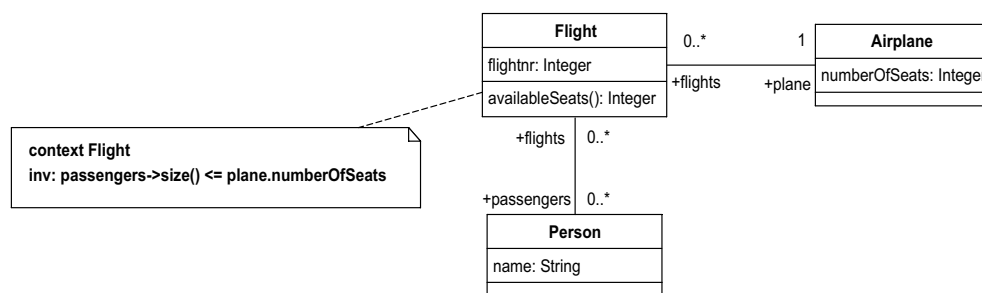


Figure 5: An example of an OCL expression [62]

## 2.4 MOF

### 2.4.1 Metamodel

Languages are often defined using a grammar in Backus Naur Form (BNF) [65]. Such a grammar describes how to create correct expressions in a certain language. However, BNF is a text-based notation and is suitable for text-based languages. Visual languages, like UML, have graphical syntax. Therefore, they require a visual mechanism for defining their syntax [28].

A language used to define another language is called a meta-language. By analogy, a language (or strictly speaking a model) used to define models is called a metamodel. It is also often called "a model of a model" [18]. The abstract syntax of UML is defined by the UML metamodel. This metamodel is defined using MOF [59].

### 2.4.2 MOF standard

MOF (Meta Object Facility) [55] is an OMG standard as well as an international standard ISO/IEC 19502:2005 [24]. It defines the language to define modeling languages and is a universal way of describing modeling constructs [23]. MOF is defined using MOF itself. So, MOF is at the highest abstraction level [18]. A fragment of the MOF metamodel is shown in Fig. 6 [10].

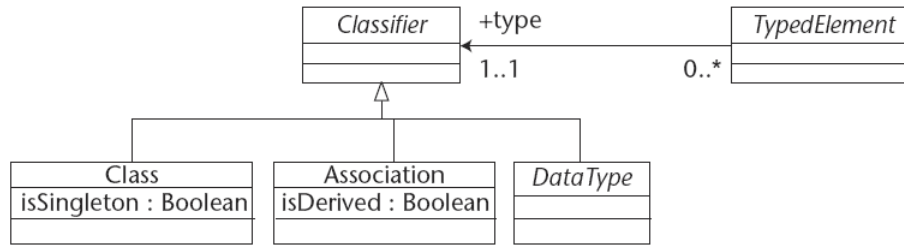


Figure 6: A fragment of the MOF metamodel abstract syntax (for MOF 1.4) [10]

MOF syntax is based on UML class diagrams. With MOF, one can model artifacts as classes and their properties as attributes of the class. The relationships between artifacts can be modeled as associations between classes representing these artifacts. Furthermore, to increase precision of the metamodel one can use OCL expressions [23].

### 2.4.3 MOF Layers

OMG defined an architecture for MOF. Traditionally, it is a four-layered architecture. These conceptual layers are called a meta-metamodel layer, a metamodel layer, a model layer, and an information layer. However, in theory, MOF 2.0 can be used with as many levels as the user needs [55].

The MOF meta-metamodel is an abstract language used to define metamodels. For instance, to define the UML metamodel, a metamodel can be described as an instance of the MOF meta-metamodel. It is a language used to define models. Models, in turn, are comprised of metadata that describe data in the information layer.

Each element in a certain layer describes elements in the layer below it. Fig. 7 captures relationships between these four layers.

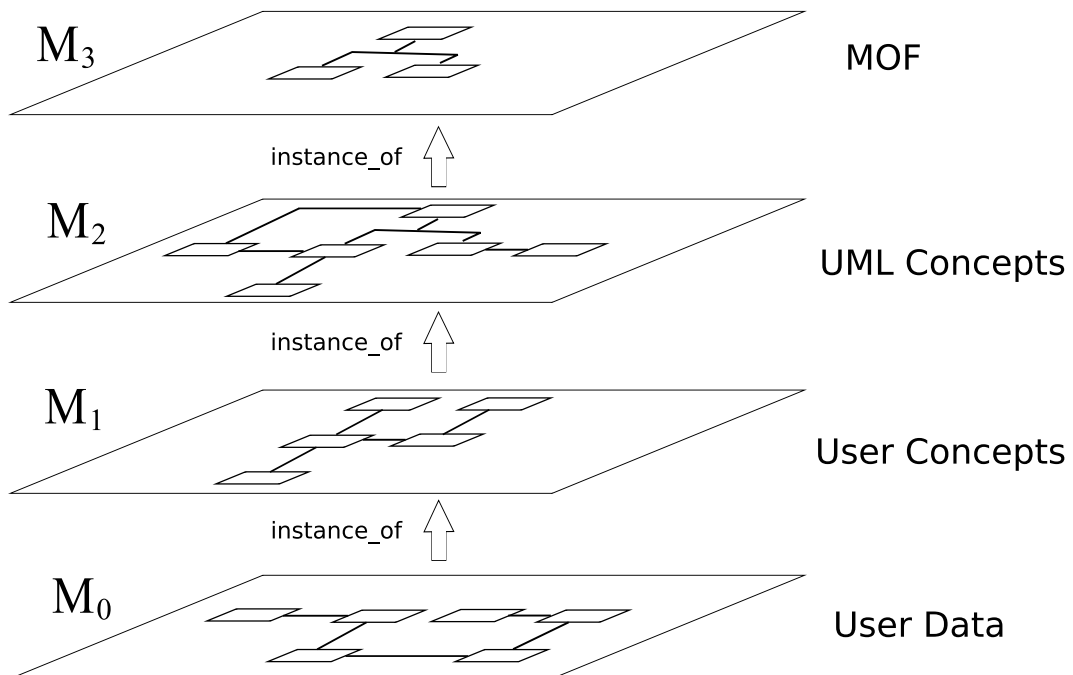


Figure 7: Traditional OMG Modeling Infrastructure [18]

## 2.5 MDA approach

The Model Driven Architecture (MDA) [36] is an approach to system development. It increases the power of models in this process. The software development process is to be driven by the modeling of the software system. That can be an evolutionary step in the software development industry.

MDA, developed by OMG, defines an architectural framework. It supports detailed specification based on models. According to OMG [36], these specifications will allow to produce interoperable, reusable, portable software components and data models. The main feature of MDA is that it completely separates the design specification from the specific technology platform. It is made possible, because MDA uses formal models. Such models can be processed by a computer. In such a development process a modeler is becoming a programmer, as well as a modeling activity is becoming a programming activity [10].

### 2.5.1 Models in MDA

As it was said in Section 2.2.2, a model has no single definition. However, from the MDA point of view a strict definition of a model is needed. It is so, because there is a need for automatic transformations within the MDA framework. For that reason in [28] such a definition is given:

*A model is a description of (part of) a system written in a well-defined language.*

A well-defined language means a language with a well-defined syntax and semantics, suitable for an automated computer interpretation. The definition does not specify a particular language. Models used with MDA can be expressed using the UML. But MDA is not restricted only to UML [28].

However, UML can help to enable the model-driven approach. It has some great strengths, such as [5]:

- providing a MOF metamodel,
- allowing to define extensions,
- allowing to raise the abstraction level of modeling.

### 2.5.2 MDA Layers

In the MDA development life cycle one can identify the same phases as in the traditional software life cycle [28]. However, there is a difference between them. In the traditional life cycle transformations from model to model, or from model to code, are done manually by human. An MDA framework supports processing and relating models. So, MDA models can be understood and processed by the machine. Such models can be on different levels of abstraction. Models on certain layer focus on particular perspectives of the system. At the core of MDA are model abstraction layers [36]:

- *Computation Independent Business Models (CIM)*

CIM is on the highest-level of business model [28]. It shows a system from the computation independent viewpoint. This viewpoint does not show structure details of the system. It focuses on the system requirements and environment [36]. In [12] CIM is compared to a domain model specification and to an ontology.

In practice, CIM often uses a specialized business process language [28]. So, specialized computer knowledge is unnecessary. Only the understanding business and business processes is needed.

- *Platform Independent Models (PIM)*

PIM shows a system from the computation-dependent perspective [12]. Such a perspective hides the details necessary for a certain platform. PIM defines a model that is independent

of any implementation technology. It focuses on the operation of a system [36]. That model ignores programming languages, OS, hardware etc. PIM is mostly described in UML or some UML derivative [60].

- *Platform Specific Models (PSM)*

PSM shows a system from the platform specific perspective [36]. This perspective is based on the platform independent one. It focuses on the usage of specific platform by a system. PSM specifies a system in terms of the certain implementation constructs. PSM can use different UML profiles to define the PSM for each target platform [60].

- *Code model*

One can distinguish also a code model layer. This represents the code, usually in a high-level language [60].

Fig. 8 shows MDA models on different model abstraction layers. Abstraction increases right to left.

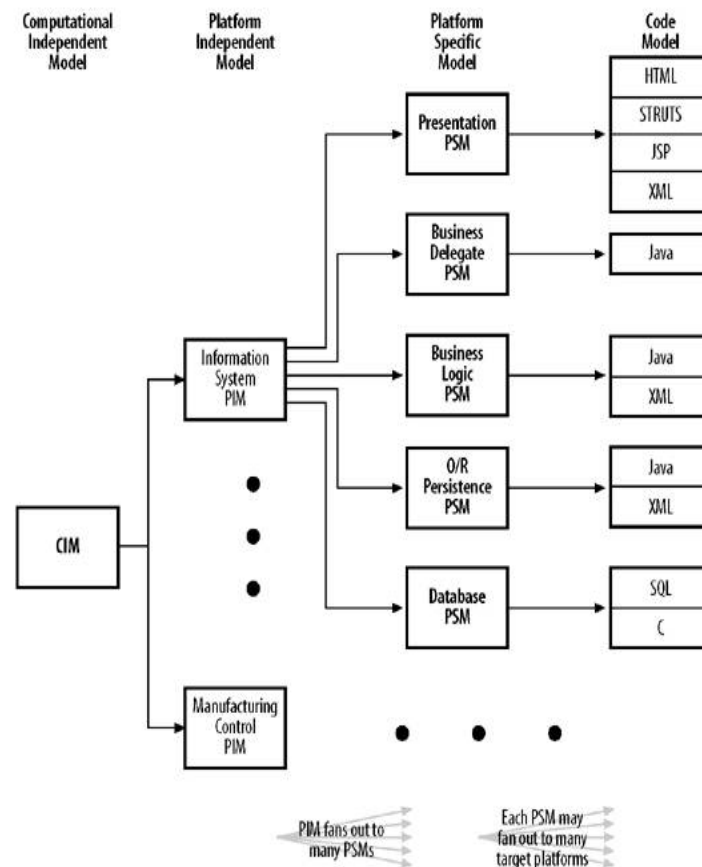


Figure 8: An example of MDA models and their relationship [60]

### 2.5.3 Transformations

In addition to models on different abstraction layers, MDA defines how these relate to each other [28]. It specifies model transformations, one model is converted into another model of the same system [36]. The main goal is to be able to transform a high level PIM into a PSM, and then each

PSM to code. These transformations are essential in MDA. The possibility of automatic transformations increases the abstraction level of the design. The MDA transformations with input and output models are shown in Fig. 9 [28].

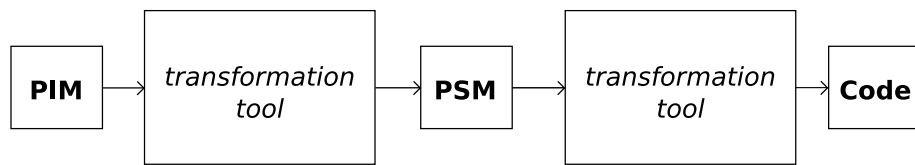


Figure 9: Major steps in the MDA development process [28]

## 2.6 Business Rules

As it was said, rules are omnipresent in our lives. Nowadays, the same is with business rules in complex information systems. A very general definition of a business rule from [6] says that:

*A business rule is a statement that defines or constrains some aspect of the business.*

Such an influence on some aspect of the business can either assert business structure or influence the behavior of the business [1].

Business Rules Group (BRG) distinguishes four categories of business rules [6]:

- *Definitions of business terms*

Definitions of business terms are itself business rules. They describe how people think and talk about things. They organize the vocabulary and allow business users to speak the same language.

- *Facts relating terms to each other*

This category contains facts that relate to the relationships between business terms. They may be specified either in natural language sentences or modeled graphically.

- *Constraints*

Business Rules are often behavior constraints. For example, they constrain updating certain data or executing specific actions.

- *Derivations*

Such rules define derivation relationships between knowledge in different forms. They specify transformations for such relationships.

In 2002, BRG published the Business Rules Manifesto [15]. It contains principles of the business rule approach. In this statement there are commonsense observations relevant to business rules, such as: "Rules should be expressed in plain language." or "Rules should be single-sourced." [63].

Business Rules manifesto states that rules shall be expressed in plain language. It is a very general statement. Thus, today business rules can be expressed at different levels of formalization. They may be described in formal language, but there is no obstacles to expressing them in informal way [1].

Rules can be also declared at different levels of complexity. They may be expressed very simply, with a single sentence. But sometimes they require a more detailed specification. Table 1 shows an example of the business rule detailed description [1].

Some business rules can be directly represented on the UML diagram (e.g. a rule: *each student is assigned to exactly one class of the school*, can be presented as in Fig. 10) or expressed in a simple manner by means of OCL expressions (e.g. *Student age can not be less than 6 years*).

Possibilities of representing business rules by UML and OCL models and their applicability in modern development processes are investigated in [53].



<b>Name:</b>	Tenured professors may administer student grades
<b>Identifier:</b>	BR123
<b>Description:</b>	Only tenured professors are granted the ability to initially input, modify, and delete grades students receive in the seminars that they and they only instruct. They may do so only during the period a seminar is active.
<b>Example:</b>	Dr. Bruce, instructor of Biology 301 Advanced Uses of Gamma Radiation, may administer the marks of all students enrolled in that seminar, but not those enrolled in Biology 302 Effects of Radiation on Arachnids, which is taught by Dr. Peters.
<b>Source:</b>	University Policies and Procedures, Doc ID: U1701, Publication date: August 14, 2000.
<b>Related rules:</b>	BR12 Qualifying For Tenure, BR65 Active Period for Seminars, BR200 Modifying Final Student Grades.

Table 1: Detailed description of the business rule [1]

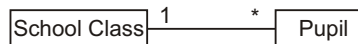


Figure 10: A simple business rule represented on the UML diagram

### 3 Knowledge in the HeKatE Design Process

#### 3.1 Knowledge and Software Engineering

The basic difference between Software and Knowledge Engineering is that Software Engineering (SE) tries to model how the system works, while Knowledge Engineering (KE) tries to capture and represent what is known about the system. In the KE approach information about how the system works can be inferred and this can be done automatically from what is known about the system [49].

In the SE approach modeled systems are becoming more complex. In order to deal with this complexity and to model the system in a more comprehensive way, design has become more and more declarative. However, programming itself remains mostly sequential, which is relevant to the way in which the program is executed on a computer (based on the Turing Machine concept). Unfortunately, there is no direct bridge between declarative design and sequential implementation [49]. The set of problems concerning an efficient and integrated Software Engineering process are commonly described as semantic gaps [35, 61, 19, 48]. Several types of semantic gaps can be distinguished. Fig. 11 shows three examples of such gaps in the waterfall life cycle model of software development process:

- **Requirement-Analysis Gap**, the so-called Analysis Specification Gap [61, 19]. It is the gap related to proper formulation of requirements and their appropriate transcription in the analysis/design model.
- **Analysis-Design Gap**, which can be considered in the field of UML. It concerns especially the distinction between diagrams that model software structure and software behavior.
- **Design-Implementation Gap**, the so-called Specification-Implementation Gap [61, 19]. It is a semantic gap between design, which is becoming more and more declarative, and its sequential implementation.

To get rid of semantic gaps in SE a more formal approach is needed. A proposal for solving these problems is based on the idea of using Knowledge Engineering methods in software design. In

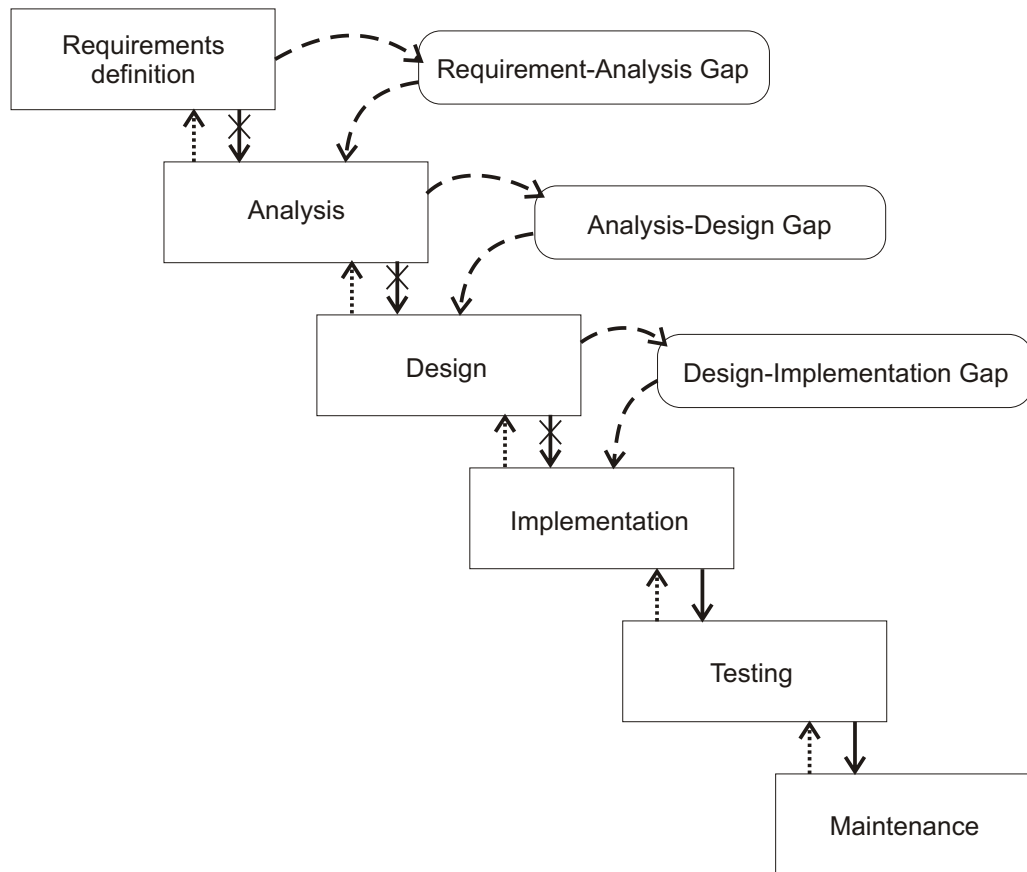


Figure 11: Semantic gaps in the waterfall life cycle model

the case of Knowledge-Based Systems there is no single modeling approach as UML in SE. Different classes of KBS may require specific approaches and use various knowledge representation methods. However, the common feature of the representations is their declarative nature [49].

Because the software design process is declarative, the application should be declarative as well. It implies that execution of the application must be provided through a declarative language, such as Lisp, Prolog or Haskell [48].

According to [49], these days SE becomes more knowledge-based, while KE is more about SE. SE can adopt advanced conceptual methods from KE, such as:

- declarative knowledge representation,
- knowledge transformation based on existing inference strategies,
- verification, validation and refinement.

Model-Driven Architecture (MDA) [36], presented in Section 2.5, may be an example of this trend.

### 3.1.1 UML in Knowledge Engineering

For the purpose of this report it is worth considering how UML can be used in Knowledge-Based Systems. Several possible approaches are described in [48]:

- *Model system with a knowledge-based approach*

This approach uses some classic knowledge representation methods, such as decision trees. However, the design of the software implementation uses UML. At last, an object-oriented code is generated. This is an easy solution, but it exposes the semantic gap.

- *Model rule-based knowledge with UML diagrams, and then generate the corresponding OO code.*

This approach extends or redefines the original semantics of UML. Unified Rule Modelling Language (rwg2009www-urml) [33] is an example of using this approach.

- *Incorporate a complete rule-based logic core into an OO application, implementing I/O interfaces, including presentation layer, in an OO language.*

The last approach is possibly the most complicated one. It relies on the incorporation of the knowledge-based component into an OO application. In this way the semantic gap between SE and KE is minimized.

Next subsection introduces the HeKatE project, where a solution similar to the last one, but more complete is developed.

## 3.2 Hybrid Knowledge Engineering

HeKatE<sup>1</sup> (Hybrid Knowledge Engineering) [49] is a research project regarding Software Engineering based on Knowledge Engineering. It tries to incorporate some well established KE tools and paradigms into the domain of SE [40]. This approach aims at providing a new software development methodology [51].

The project is based on experiences with the MIRELLA [46] project. It develops and refines the integrated design process for Rule-Based Systems (proposed in Mirella). The integrated design process is based on the idea of a meta-level approach to the design process [49]. It is a top-down hierarchical design methodology consisting of three phases: conceptual, logical, and physical.

HeKatE provides a separation of the design phases. However, it does not cause semantic gaps between them. It is so, because the phases are integrated. The design procedure takes place at the abstract, logical level [49]. Then, this design specification is automatically translated into a low-level code (Prolog and XML).

The generated code is to be a prototype implementation of the system. Thanks to automatic code generation, system can be formally verified. Formal properties of the system can be automatically analyzed still during the design [49].

The project uses the XTT (EXtended Tabular Trees) to model, represent, and store the business logic [40]. It is a hybrid knowledge representation, combining decision trees and decision tables. XTT uses a hierarchical visual representation of the decision tables linked into a tree-like structure [48]. It offers transparent knowledge representation. The XTT method is supported by a Prolog-based interpretation. So, the logic of the system is encoded with the use of a Prolog-based representation. Such a declarative, rule-based logic core can be integrated into an OO application as a logical model (as in the MVC [11] design pattern). Other parts of an application can be developed with classic programming techniques. Thus, the HeKatE project may provide a coherent runtime environment for the combined Prolog and Java/C applications [48].

### 3.2.1 HeKatE Phases

This subsection describes the HeKatE design phases. In each phase another design method is used [51]. Fig. 12 shows the top-down hierarchical design methodology of the HeKatE project [26]. It consists of three phases:

1. *Conceptual design.* Conceptual design is the most abstract phase. During this phase both system attributes and their functional relationships are identified [45]. This phase uses ARD (Attribute-Relationship) [51] diagrams as the modeling tool. ARD method uses a visual representation to specify relationships between attributes. It allows to design the logical XTT structure [49].

---

<sup>1</sup><http://hekate.ia.agh.edu.pl/>

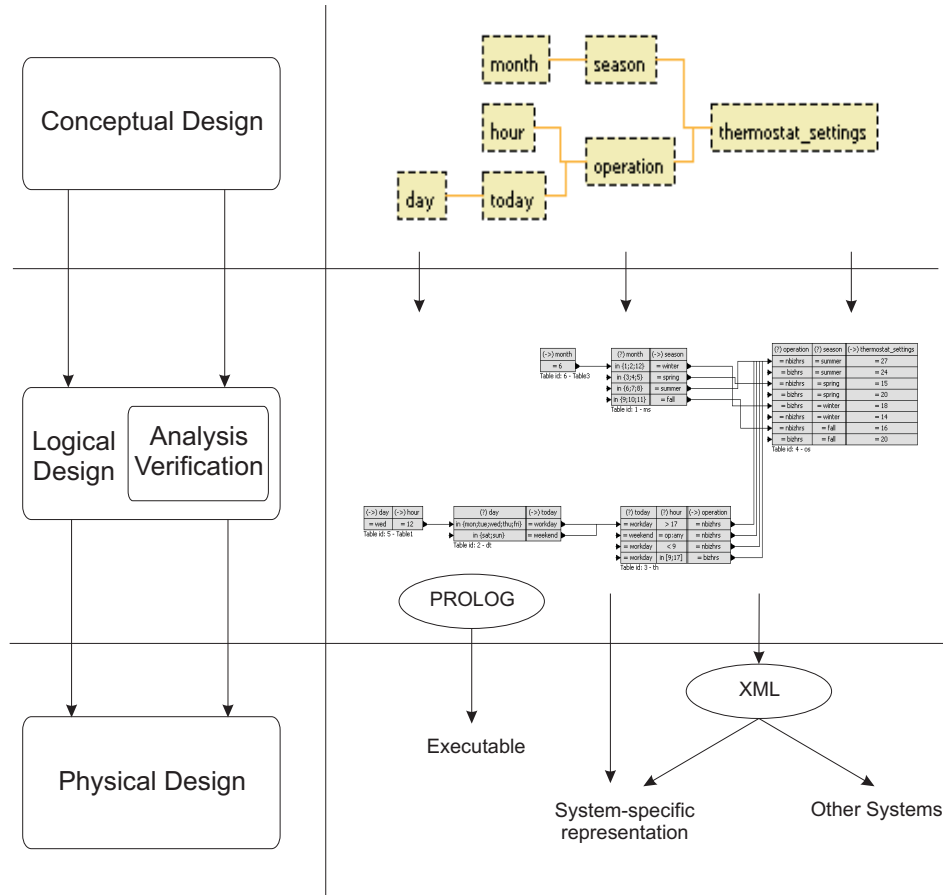


Figure 12: HeKatE methodology phases [26]

2. *Logical design.* In the logical design phase system structure is represented as a XTT hierarchy [49]. The preliminary model of XTT can be obtained as a result of the previous phase. This phase uses the XTT representation as the design tool. The logical design phase allows on-line analysis, verification, and also revision and optimization (if necessary) of the designed system properties, using Prolog [49].
3. *Physical design.* During the physical design phase the preliminary Prolog-based implementation of the system is generated from the XTT model [49]. Generated code can be compiled, executed and debugged.

Next subsection presents the integrated design environment from the HeKatE project.

### 3.2.2 HaDEs

HaDEs is integrated design environment from the HeKatE project [37, 27]. It supports:

- ARD design with Visual ARD Rapid Development Alloy (VARDA) and HeKatE Java Editor Project (HJEd),
- XTT modeling with HQEd,

HQEd (HeKatE Qt Editor) [26] is a CASE tool developed for modeling in HeKatE methodology. It supports ARD and XTT visual design methods. The tool also provides additional functionalities, such as formal verification of the designed system properties, code generation of a prototype implementation and execution of the generated application.

- runtime with Hekate RunTime (HeaRT) [47, 42],

HeaRT provides the unified run-time environment. It is an inference engine supported with optional sequential (C/Java) runtime.

- knowledge translation with HeKatE Translation Framework (HaThoR).

The XTT model is encoded in the HeKatE Meta Representation (HMR) [39]. This provides a high-level textual human-readable rule representation and can be directly execute by the HeaRT engine. The model can be also serialized to XML-based HeKatE Markup Language (HML) [38]. HaThoR is an XML-based framework for knowledge translation from/to HML to other formats.

### 3.3 Conceptual Design Phase

#### 3.3.1 Conceptual Modelling Using ARD

The ARD method [45, 32, 51] supports designing at a very general design level. It poses a kind of a requirements specification method. ARD is used during the conceptual phase of HeKatE methodology.

On its input ARD requires a general system description. It can be written in the natural language. On its output ARD gives a model which captures knowledge about relations between attributes describing system properties. This model can be used in the next phase for designing the logical XTT structure [49].

#### 3.3.2 ARD Syntax

Attributive Logic [32] uses attributes for denoting certain properties in a system [51]. The ARD method aims at capturing relations between such attributes. An ARD Diagram is a set of properties and dependencies among them. An exemplary ARD diagram, shown in Fig. 13, describes ARD concepts. Depicted labels in the diagram point out several concepts related to ARD. The detailed and formal description of them one can find in [51]. Simplistic definitions of such concepts are as follows [26]:

- A *Conceptual Attribute* is an attribute describing some general aspects of the system. Such an attribute must be then specified and refined. Its name begins with a capital letter: *Thermostat*.
- A *Physical Attribute* is an attribute describing well-defined atomic aspects of the system. Such an attribute cannot be further specified or refined. Its name begins with a lower case: *theThermostatSetting*.
- A *Property* is a set of attributes, which describes some feature on some abstraction level.
- A *Simple Property* is a property described by a single attribute.
- A *Complex Property* is a property described by multiple attributes.
- A *Dependency* is an ordered pair of properties, in which the first one is an independent property and the second is dependent one.

In the next subsection ARD transformations are introduced.

#### 3.3.3 ARD Transformations

The most important issue in the ARD method it to identify as many property functional dependencies as possible. However, it is a very complex task [51]. To manage this task, transformations are introduced. Transformations transform the diagram to a more detailed and less abstract form.

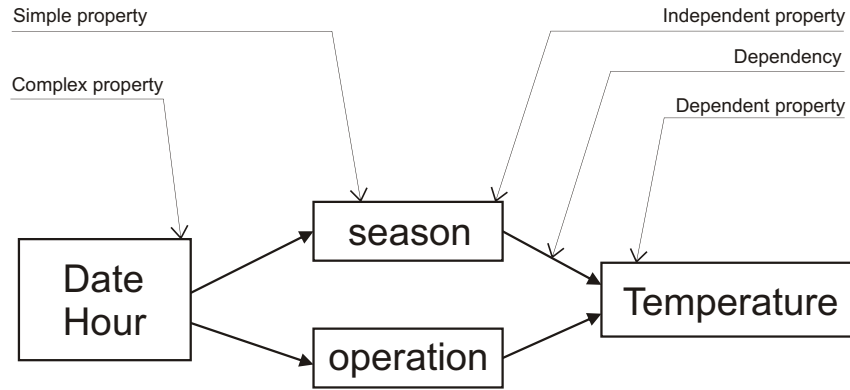


Figure 13: An exemplary ARD diagram describing ARD concepts

ARD supports two types of transformations [51]. Each one creates a more detailed diagram either introducing new attributes (finalization) or defining functional dependencies (split). An example of finalization is shown in Fig. 14.

With the use of these two transformations one can gradually refine properties, attributes and functional dependencies in the designed system. The whole process ends with all of the properties described only by physical attributes and all of the functional dependencies [51] defined.

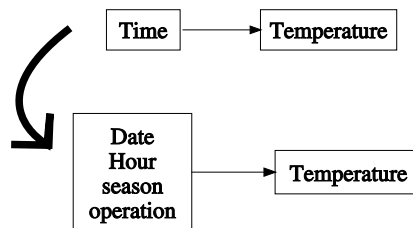


Figure 14: An example of ARD finalization

### 3.3.4 Hierarchy of ARD

During the design process the ARD diagram grows. With making the design more and more specific new diagrams are created. These diagrams are at different levels of abstraction. Such consecutive levels of more and more detailed diagrams establish a hierarchical model [51].

For the purpose of gradual refinement of a designed system the hierarchical model is stored. It is done in order to be able to identify the origin of given properties or to get back to previous diagram levels for refactoring purposes. The information about changes at consecutive diagram levels are captured in TPH (Transformation Process History) [51] diagrams. A TPH diagram forms a tree-like structure and denotes how particular property has been split or finalized. An exemplary TPH diagram is shown in Fig. 15.

### 3.3.5 ARD Tools

The ARD design process is supported by VARDA, a purely declarative tool written in the Prolog language [51]. It supports automated visualization at any diagram level. HQEd, mentioned in Section 3.2.2, allows the user only to display the ARD model, loaded from a ARDML file [26].

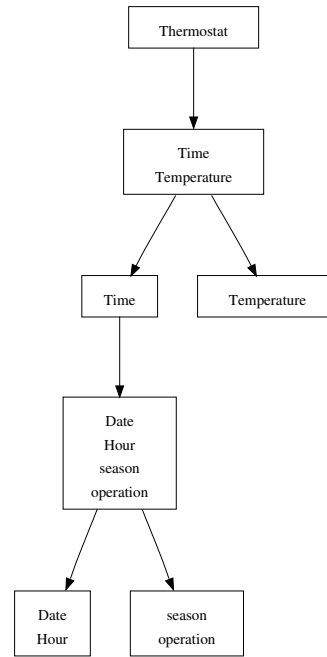


Figure 15: An example of TPH diagram

### 3.4 Logical Design Phase

#### 3.4.1 EXtended Tabular Trees

EXtended Tabular Trees (XTT) [44] is a knowledge representation and a design method which aims at combining decision trees and decision tables. Tables are linked into a tree-like structure and they constitute a hierarchical visual representation. In [43] three important representation levels are distinguished:

1. *visual* – the model is represented by a hierarchical structure of linked extended decision tables,
2. *logical* – tables correspond to sequences of extended decision rules,
3. *implementation* – rules are processed using a Prolog representation.

Such a representation has the power of the decision table representation. It is both a very intuitive way of knowledge representation and a highly efficient way of visualization with high data density [44].

#### 3.4.2 XTT Visual Representation

From the Rule-Based System design point of view in the XTT method a visual representation is crucial. The key related concepts are [26]:

- rules, which give logical representation of the system knowledge base,
- decision tables, which store knowledge,
- decision trees, which serve as knowledge components.

An XTT tree is a structure composed of two or more tables connected with at least one connection [26]. An exemplary XTT tree is shown in Fig. 16.





### 3.6 HeKatE vs Software and Knowledge Engineering

At the starting point in HeKatE the ARD design method is used. It solves the Analysis Specification Gap problem [48]. It is so, because ARD is a top-down methodology allowing gradually consecutive refinement. It is possible thanks to the introduction of many different detail levels containing system components and dependencies among them.

Analysis-Design Gap is solved by providing the integration of conceptual and logical phases. A result from the first phase may be a starting point for the second phase – logical design phase.

In turn, declarative design methods for the business logic solve the Design-Implementation Gap problem [49]. It is so, because there is no translation from the declarative design into the implementation language. It is thanks to the knowledge base, which is specified and encoded in the Prolog language. So, the knowledge base becomes an executable application [48].

In HeKatE there is also no evaluation problem, because the whole model and the transformations at every stage are formal and can be verified. With regards to the latter issue mentioned in Section 3.1 – the so-called Separation Problem – in HeKatE it is also solved. In HeKatE there is a strong separation between the software logic model, and the presentation layer [48]. The logic of the system is encoded with the use of a Prolog-based representation and other parts of an application can be developed with classic programming techniques.

Thus, thanks to consistency and formal form of knowledge in HeKatE, the Semantic Gaps in the design process are decreased or even eliminated [49].

The HeKatE approach may seem similar to the MDA approach [49], especially to the formalized transition from the PIM to the PSM model. However, in HeKatE there is no platform specific model, because both XML and the Prolog code are portable. The unified run-time environment is provided by HeaRT [47].

## 4 UML Representation of ARD

The key assumption of ARD design is that the attributes are functionally dependent. This is similar in any Rule-Based System design with knowledge specification in *Attributive Logic* [32]. ARD allows for specification of functional dependencies between system attributes using a visual representation [51]. UML representation of ARD has been proposed in [43, 30] and described in the thesis [29].

### 4.1 Preliminary Approaches of Modeling ARD in UML

#### 4.1.1 Approach with Activity Diagrams

Looking at an ARD diagram the first thought that comes to mind is the similarity to activity diagrams. It is so, because of functional dependencies in ARD diagrams which seem to look very similar to flows in UML activity diagrams.

An ARD diagram as in Fig. 12 can be presented in UML as in Fig. 17. In such a case, all flows go simultaneously until they get to the join node. The activity waits till all flows reach the join node, and after that an action (occurring after the join node) is performed. This is, indeed, the correct behavior of the model.

However, the original ARD does not provide behavior description, but only the structure with functional dependencies. So, such a model, based on activity diagram, is not compliant with ARD.

#### 4.1.2 Approach with Component Diagrams

As it was said in Section 3.3.1, the ARD method aims at capturing relations between attributes describing system properties. Complex properties are described by two or more attributes. This compositions can be easily presented in UML component diagrams. Indeed, a component can be composed

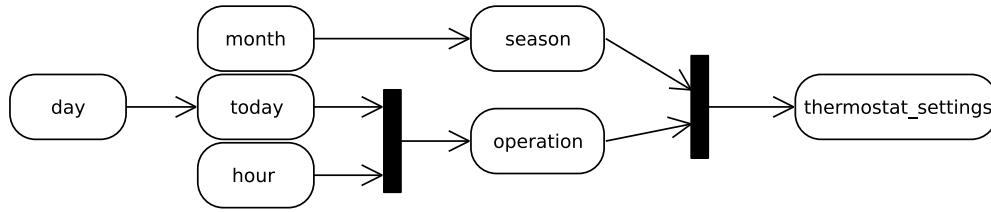


Figure 17: An example of ARD representation using activity diagrams

of other components or classes, as in Fig. 18. Therefore, in such an approach, using component diagrams, a UML model of ARD would use a static structure diagram to show functional dependencies.

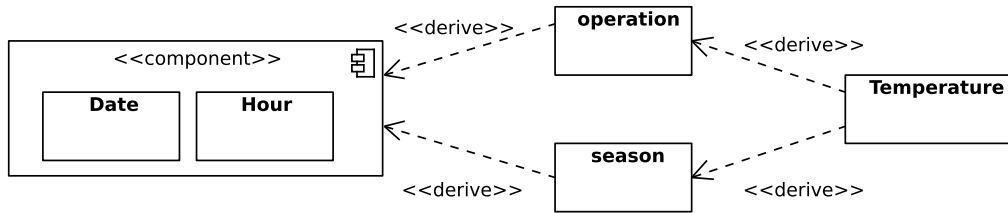


Figure 18: An example of an ARD representation using component diagrams

However, a UML model of ARD shall be also capable to represent TPH relations. An example of this is presented in Fig. 19. Although it seems to be a correct UML diagram, in fact UML 2.0 does not provide the relation of composition in component diagrams, as in class diagrams. So, this approach had to be also abandoned.

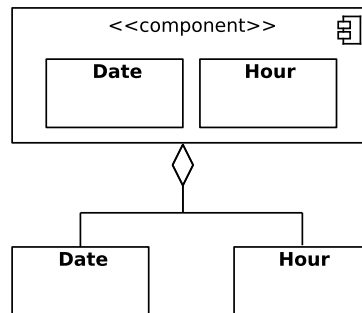


Figure 19: An example of a TPH representation using component diagrams

## 4.2 The Proposed ARD Model

This subsection introduces a representation of ARD in UML. The representation uses class diagrams to convey ARD. Next subsections present syntax and semantics of used artifacts, UML model of ARD and its evaluation.

### 4.2.1 Syntax and Semantics of Artifacts

In UML classes describe sets of objects that share the same specifications of features, constraints, and semantics [59]. Attributes are one of class features. Same is with properties in ARD which are described by attributes. According to the UML Superstructure, classes have following purposes [59]:

- to specify a classification of objects,
- to specify the features that characterize the structure and behavior of those objects.

Because ARD does not describe the behavior, there is no need to use operations in the UML representation of ARD. The Table 2 lists the artifacts used in UML diagrams representing ARD.

Table 2: UML artifacts [59] used in the proposed representation




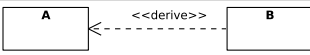
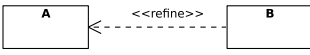
UML artifact	Name	Description
	Class	Specifies an entity type which is some classification of objects.
	Abstract Class	Same as class, specifies an entity type, however this is an entity type of which no instances exist.
	Aggregation	Specifies a form of association in which a whole is related to its part.

Table 3: Types of used dependencies [59]

Dependency	Stereotype	Description
	«derive»	Specifies a derivation relationship among model elements, where one of them can be computed from the another.
	«refine»	Specifies a refinement relationship among model elements at different levels of development. Refinement can be used to model transformation from one to another phase of a sequential model development.

In UML not everything is strictly specified. For example, in Superstructure ver. 2.2 [59] it is claimed that:

*Precise semantics of shared aggregation varies by application area and modeler [59, p. 39].*

*Attempting to create an instance of an abstract class is undefined - some languages may make this action illegal, others may create a partial instance for testing purposes [59, p. 218].*

In addition to aforesaid artifacts, the proposed model uses the stereotyped dependencies from Standard Profile L2 [59, see: Annex C]. The Table 3 lists the types of stereotyped dependencies used in diagrams.

#### 4.2.2 UML Model

Elements of the proposed UML model correspond one-to-one with the properties in ARD (it is a bijective relation). In the proposed UML model of ARD:

- a class corresponds to the ARD simple property,
- an abstract class with attributes corresponds to the ARD complex property,
- a «derive» dependency corresponds to the ARD dependency between attributes,
- a «refine» dependency corresponds to the ARD finalization transformation,
- an aggregation corresponds to the ARD split transformation.



Figure 20: A simple example of an ARD diagram a) original syntax, b) UML representation

Furthermore, the difference between a conceptual and a physical attribute is presented in the same form as in ARD. So, the name of a conceptual attribute begins with a capital, while the name of a physical attribute begins with a lower case.

Fig. 20 shows an example of a simple ARD diagram and its UML representation. The corresponding representation of a TPH diagram is shown in Fig. 21. The interpretation of ARD is as follows: A property *Temperature* can be derived from *Time*. In other words, the *Temperature* can be expressed as a function of *Time*. The TPH model shows the history of transformations: finalization – in *Thermostat* system there are distinguished two properties: *Time* and *Temperature*, split – a complex property is separated into two properties and identified a dependency between them – the *Temperature* depends on the *Time*.

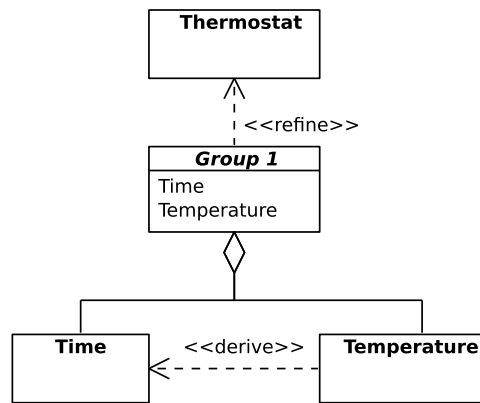


Figure 21: A TPH diagram corresponding to ARD from Fig. 20

### 4.2.3 Evaluation

The proposed UML model is very similar to the original ARD. It is very simple and intuitive. This approach does not introduce custom elements, only standard UML artifacts are used.

Regarding the relations between the artifacts, UML is a much richer language than ARD. There are some relationships which can be observed in ARD, but they are not distinguished.

The first not distinguished relation is the relation of realization. If a conceptual attribute is finalized to a physical attribute, this is a quite different case than if it is finalized to some other conceptual attribute or attributes. The first case is an example of realization. It is so, because the physical attribute realizes the concept of the conceptual attribute. The second one is an example of some refinement.

The second relation which is not distinguished is a «trace» dependency. A trace relationship among model elements can be used in a TPH diagram to show elements representing the same concept at different levels of the design process.

## 4.3 MOF Metamodel of the Proposed UML Model

The abstract syntax of UML is defined by the UML metamodel. The proposed model uses only a subset of UML artifacts and their relationships. So, its metamodel can be created as a subset of UML metamodel [59] with constraints imposed.

### 4.3.1 Metamodel for ARD Diagrams

Fig. 22 shows the metamodel of the UML representation for ARD diagrams. The representation uses only UML classes (with or without attributes) and the «derive» dependencies.

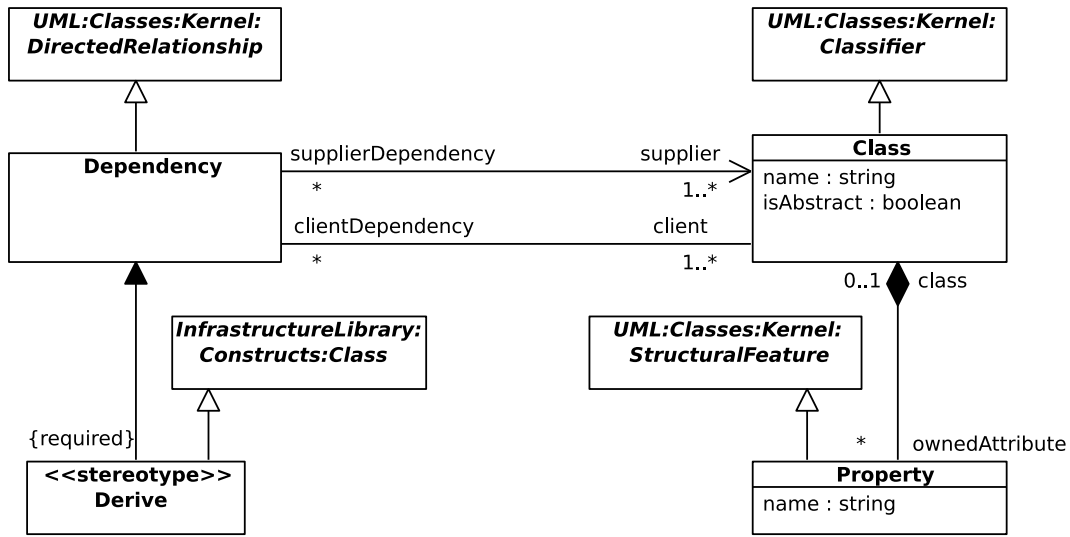


Figure 22: Metamodel for ARD diagrams

Moreover, there is a need to define some constraints in OCL for the metamodel. It is so, because the metamodel itself is not so precisely stated as required by the models. The following statements impose some constraints on the models developed using the provided metamodel:

1. *There are no ARD attributes which are functionally dependent on physical attribute.*  
A class which name begins from a lower case shall not be a supplier class in any dependency.
2. *An ARD attribute can not depend on itself.*  
A class which is a client of the dependency have to be distinct from a supplier class of this dependency.
3. *Every ARD complex property has two or more attributes.*  
Every abstract class has no less than two attributes and its name begins with string “Group”.
4. *Every ARD simple property has only one attribute.*  
Every class which is not abstract has no attributes.

OCL expressions for these statemets can be found in Appendix A.

### 4.3.2 Metamodel for TPH Diagrams

THP diagrams are more complex then ARD diagrams. So, there is a need to define the separate metamodel for TPH diagrams. It have to be based on the previous metamodel.

Fig. 23 shows the metamodel of the UML representation for TPH diagrams. The representation uses UML classes (with or without attributes), aggregations and the «refine» dependencies.

This metamodel also requires some OCL constraints to specify the exact syntax of the TPH representation. All OCL expressions from ARD metamodel are valid for TPH metamodel. However, there is more statements in case of TPH than it was in ARD metamodel.

The following statements impose some constraints on the models developed using the provided metamodel:

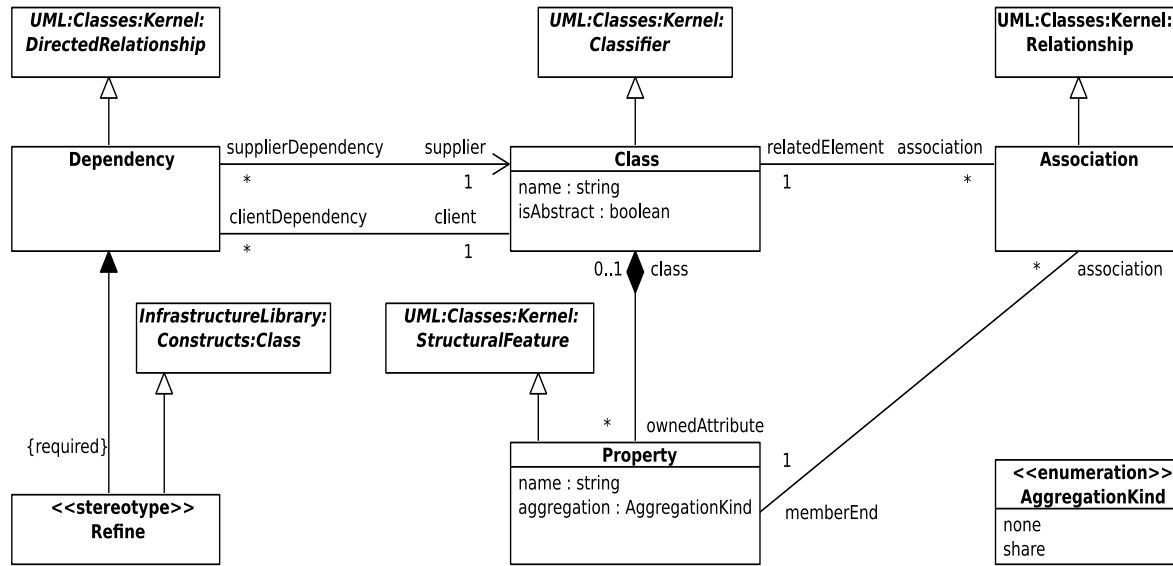


Figure 23: Metamodel for TPH diagrams

1. *There is only one simple ARD property as a root of TPH diagram.*

There is only one class which is not an abstract class and is not an end of an aggregation.

2. *ARD properties which are finalization products come from the finalized complex property.*

If a class has some aggregated classes, it has to be an abstract class and every attribute from this class shall become either a name of a simple class or an attribute of a new abstract class.

3. *Physical attribute can not be split or finalized.*

A class which name begins from a lower case shall not be a supplier class in any dependency and shall not have any aggregated classes.

OCL expressions conforms to above statements can be found in Appendix A.

## 5 UML Representation of XTT

Decision tables typically group elements of the same or similar kind [32]. They represent rules that have the same attributes. Such rules in a single table are processed sequentially. So, a reasonable idea for a representation of such a table is to use a UML diagram, that shows not so much the structure of the system, but its behavior. It could be a Use Case Diagrams, an Activity Diagram or a State Machine Diagram, as well as one of Diagrams of Interaction (a Sequence or Collaboration Diagram).

According to the discussion in [50] State Machine Diagrams and Activity Diagrams seem to be the best UML candidates for rule modeling. However, they are not good enough to serve the purpose of rule modeling with similar expressiveness as XTT. In case of larger systems, where the number of states grows fast, their use poses some practical problems. But it is possible to use them to express rules in case of smaller systems. So the attempts to use UML for XTT are to investigate Activity and State Machine Diagrams.

Activity and State Diagrams are related. However, there is a difference between them. The state diagram shows the possible states of the object and the transitions that cause a change in state. It focuses on an object undergoing a process (or on a process as an object). The activity diagram, in turn, focuses on the flow of activities involved in a single process and shows how they depend on one another.

## 5.1 Preliminary Approaches of Modeling XTT in UML

The fundamental issue for modeling XTT in UML is a representation of an XTT table. Although the UML Superstructure Specification ver. 2.2 [59] in *Appendix E* introduced Tabular Notation for Sequence Diagrams, this is not a representation of any custom table. So, it can not be just used for representing XTT table. Moreover, it is rather a serialization of Sequence Diagram to the tabular form.

Some attempts of XTT representation are shown on the example of XTT table using for determination of business hours. The exemplary XTT table is shown in Table 4.

today	hour	operation
workday	> 17	nbizhrs
weekend	ANY	nbizhrs
workday	< 9	nbizhrs
workday	in 9,17	bizhrs

Table 4: An exemplary XTT table [26]

The first attempt was carried out using State Machine Diagram for XTT modelling. Such a diagram captures the behavior of a software system. State Machine uses graph notation to represent this behavior [60].

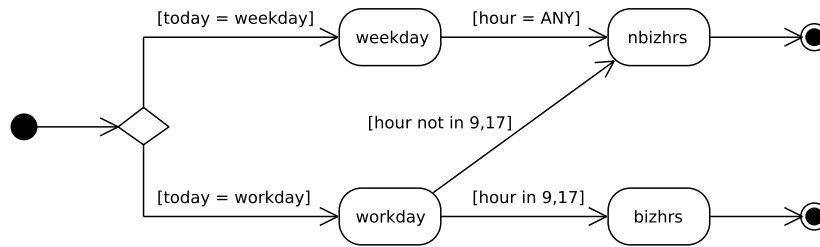


Figure 24: The first approach – State Machine Diagram corresponding to XTT table

The Fig. 24 shows the corresponding UML state diagram for the example XTT table. In this approach XTT table is transformed into an UML state diagram. The model is processed sequentially, so when some of attribute is set, it sets some state, and this process is proceed untill the control flow reach the final state.

However, in this approach it is not clear which attributes should be transformed to the states and which to guard conditions. This problem can be avoided by setting some rules. For example, every value of the XTT output attribute shall become a state and individual row compound of values of XTT input attributes (conjunction of values in their cells) become guard conditions.

Moreover, the diagram could be more readable if we use a Fork Pseudostate instead of a Choice Pseudostate. This is shown in Fig. 25. However, if there is an deficiency in an XTT table (e.g. as a result of a mistake) and different rows will not exclude each other, than the fork pseudostate duplicates the input value and may transfer the control to more than one edge of the subsequent states.

The limitation of all of these approaches is the lack of the output attribute naming, and for getting the names of input attributes it is needed to search for them in the guard conditions.

## 5.2 The Proposed XTT Model

This subsection introduces a proposal of XTT representation in UML. The representation uses Activity Diagrams to convey XTT. Next subsubsections present used artifacts, UML model of XTT and its evaluation.

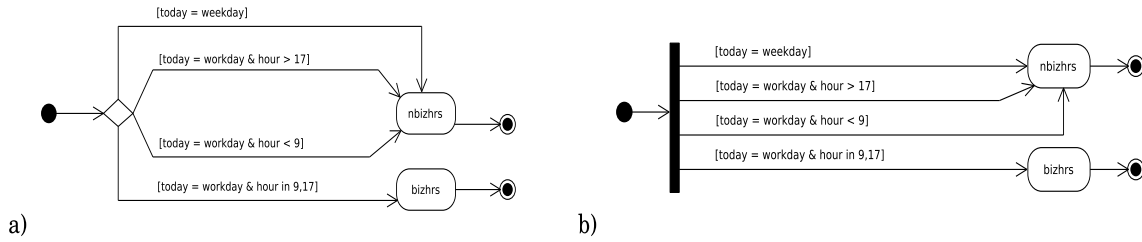


Figure 25: State Machines Diagrams corresponding to XTT table: a) with a Choice Pseudostate  
b) with a Fork Pseudostate

### 5.2.1 Used Artifacts

In general, activity diagrams are related to flow diagrams and can illustrate the activities taking place in the system. The Table 5 lists the types of nodes used in activity diagrams.

Table 5: UML artifacts [59] – types of nodes used in the proposed representation

	Parameter of Activity
	Action
	Decision Node
	Merge Node
	Fork Node
	Join Node

### 5.2.2 UML Model

There are two levels of abstraction of XTT model:

- the lower level – it is a level of XTT table, so only single table is represented,
- the higher level – it is a level of the whole XTT model, which consists of XTT tables.

**The Lower Level Model** At this level a single XTT table is represented. An example of such representation corresponding to XTT table from Section 5.1 is shown in Fig. 26.

In the proposed UML model of XTT table types of nodes are used as follows:

- Activity Parameter - the parameter representing a XTT attribute,
- Action - sets the value of the output parameter,
- Decision Node - enables checking distinct values of input parameter,





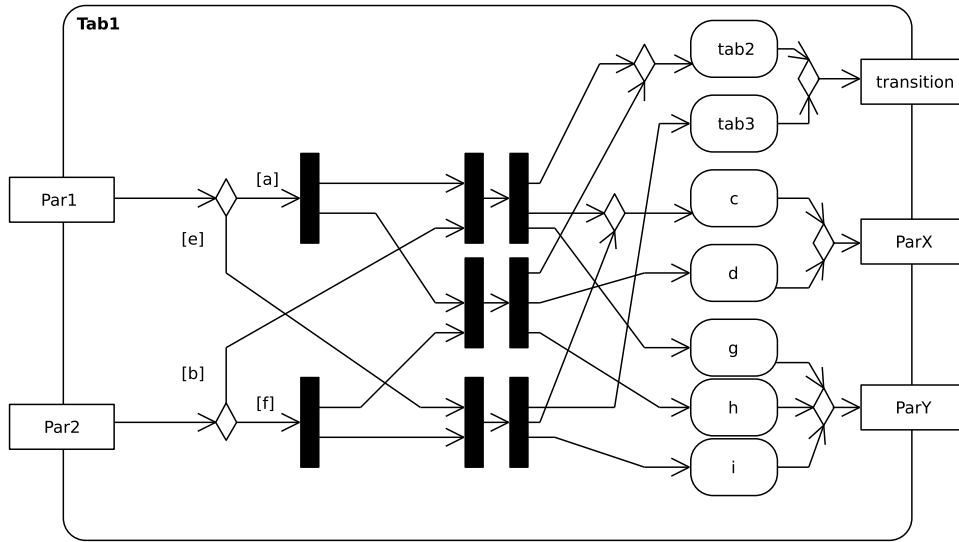


Figure 27: The lower level more complex model example

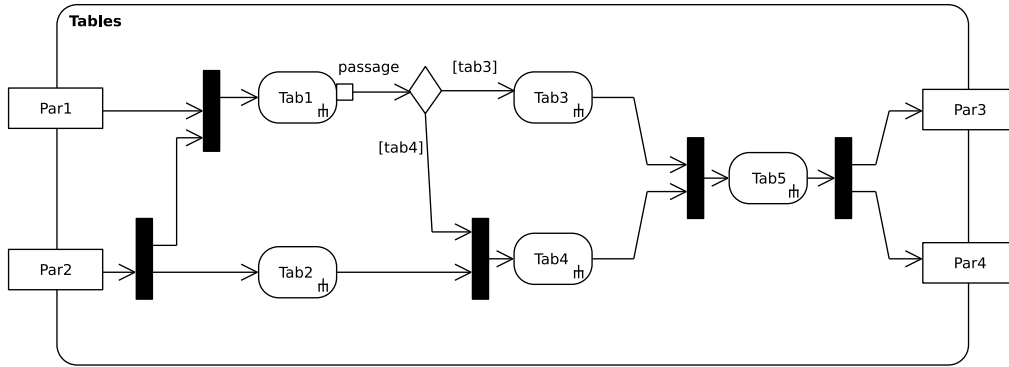


Figure 28: The higher level model example

Unfortunately, when the number of rules in XTT table grows, the diagram becomes poorly readable. If it comes to readability, tables seem to be irreplaceable. Hence, they are so popular in Knowledge Engineering. But, there are advantages of using the proposed UML representation. First, there is no redundancy of attribute values as in XTT table. Second, looking at the model one can easily see inputs and outputs of the system. Third, there is a bi-level hierarchy of diagrams in the model. So, there is possibility of creating more complex XTT trees, without the risk of readability.

### 5.3 Metamodel of the Proposed Model

The abstract syntax of UML is defined by the UML metamodel. The proposed model uses only a subset of UML artifacts and their relationships. So, the metamodel of it can be created as a subset of UML metamodel [59] with constraints imposed.

#### 5.3.1 Metamodel for XTT Diagrams

As in the case of ARD, the proposed XTT model uses a subset of UML artifacts and their relationships. The lower and the higher level models are based on Activity Diagrams. So, the metamodels of them have a lot in common. These metamodels are created as a subset of UML Activity Diagram metamodel [59] with constraints imposed.

Fig. 29 shows the metamodel of the UML representation for XTT diagrams.

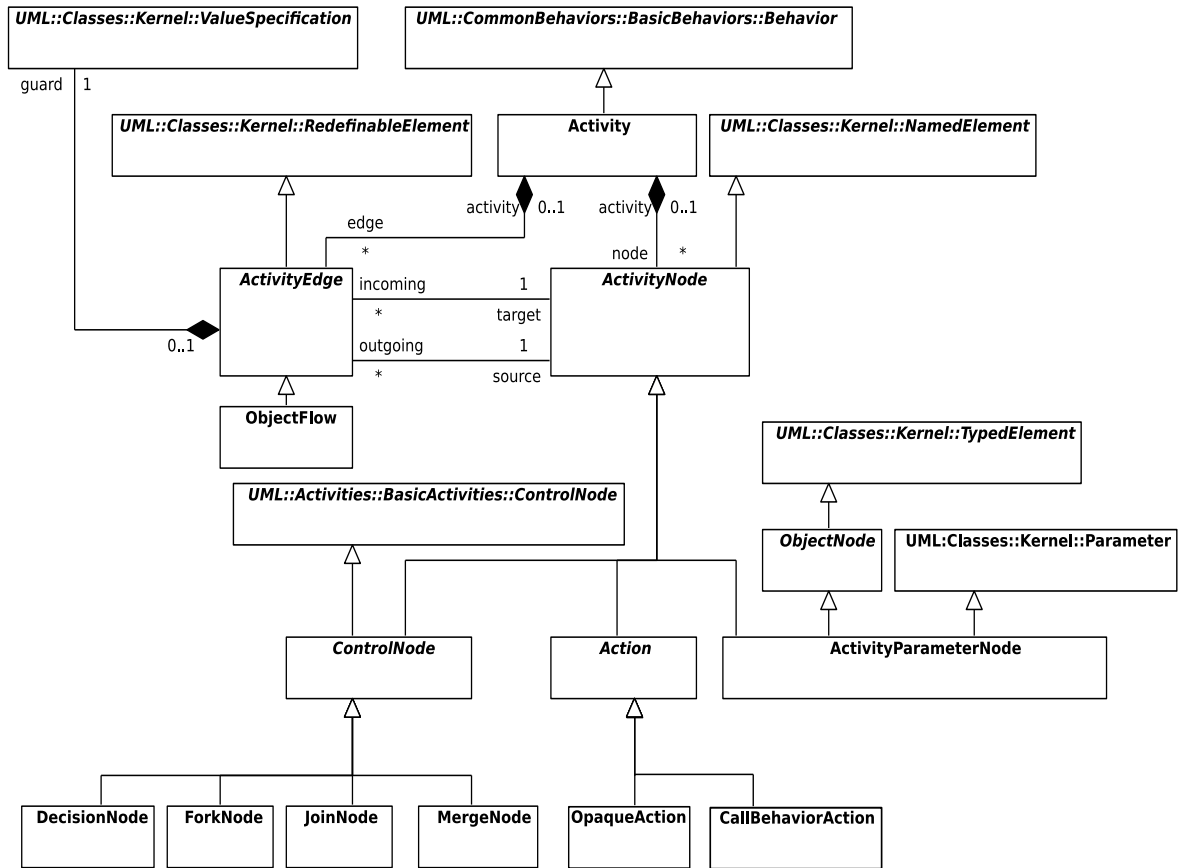


Figure 29: Metamodel for XTT diagrams

The metamodel generates also models, which do not match to the proposed kind of models. It is worth noting that the problem of the metamodel is that it does not require the order of nodes. The required order of nodes in XTT models is known.

**The Lower Level Metamodel** The required order of nodes for XTT model at the lower abstraction level is as follows (nodes which can occur optional are in brackets):

Activity Parameter Node → Decision Node → (Fork Node) → Join Node<sup>2</sup> → Fork Node<sup>3</sup> → (Merge Node) → Action → (Merge Node) → Activity Parameter Node.

The solution for this is to use constraint expressions in OCL which enforce the order of nodes. These OCL expressions can be found in Appendix B.

**The Higher Level Metamodel** The higher level model of XTT uses also an Output Pin artifact. So, the metamodel from Fig. 29 has to be extended by introducing an additional UML artifact. Additional part introducing Output Pin to the metamodel is presented in Fig. 30.

The required order of nodes for XTT model at the higher abstraction level is as follows (nodes which can occur optional are in brackets):

Activity Parameter Node → (Fork Node) → \* (Join Node) → Action →

- (Decision Node)<sup>4</sup> → back to \*,

<sup>2</sup> Element occurs when there is more than one input activity parameter node.

<sup>3</sup> Element occurs when there is more than one output activity parameter node or there is the passage parameter node (which enables the passage to different tables).

<sup>4</sup> Element occurs when previous Action has Output Pin.

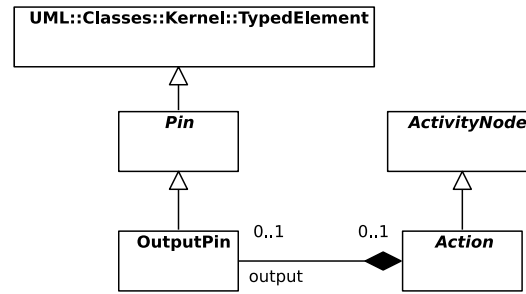


Figure 30: Introducing Output Pin to the metamodel of XTT

- (Fork Node) → Activity Parameter Node.

OCL expressions which enforce the order can be found in Appendix B.

## 6 Model Translations

### 6.1 XMI Representation

XML Metadata Interchange (XMI) [54] is an XML-based OMG standard for exchanging metadata information. Its strength is the compatibility with XML parsers. It can be used to represent any model or meta-model which is MOF compliant. However, XMI is mostly applied to encode and transport UML models and should increase the interchange of UML models. Despite this fact, there is still a variety of XMI formats. In fact, every UML tool storing models in XMI uses its own tool-specific XMI format. Possible causes of this can be difficult implementation, inconsistency and incompleteness of the XMI specification [25].

For the purpose of this report version 2.1 of XMI is adopted. It is the newest XMI version which is supported by some UML tools (e.g. Altova UModel, Visual Paradigm for UML). Although XMI is predominantly used to exchange model data between different UML tools [8], in this thesis it is considered as a format for serializing models which can be further used to validate and translate models to the HMR representation, and generate code of prototype application afterwards.

#### 6.1.1 Syntax of an XMI Document

XMI tag	Description
XMI	the root element which must have an <code>xmi.version</code> attribute
XMI.header	a placeholder for information on the model
XMI.documentation	holds end-user information

Table 7: Some of defined XMI tags of XMI header [2]

It is worth mentioning that in general XMI Specification does not define XML tags for UML artifacts (in general, because a few XMI tags are defined, see Table 7). Instead, it specifies how to create them for metamodel concepts [2]. Because of changes in UML and XMI specifications, versions of XMI differ from one to another significantly. But also XMI documents in the same version can differ between tools, either because of lack of precision or missinterpretation.

There are also some predefined attributes. Every UML artifact represented in an XMI document has its own and unique `xmi:id` value. However, this value is not required to be globally unique. It only helps identify such an artifact in the model (or different versions of the model) and refers (by an `xmi:idref` attribute) to that element from the context of other elements. A globally unique identifier for an XMI element can be provided by an attribute `xmi:uuid` [54].

Because XMI is based on XML, it can be validated with an appropriate Document Type Definition (DTD) or XML Schema. However, a validation with DTD or XML Schema does not guarantee that the model conforms to its MOF metamodel. It is so, because DTD or XML Schema cannot express rules that the MOF metamodel can.

### 6.1.2 XMI Representation of UML Model for ARD

Table 8 shows the examples of XMI code, generated by a UML tool Altova UModel, which represent some UML artifacts used in the proposed UML representation of ARD.

It is important to notice some differences between XMI documents generated by different UML tools. The main differences are:

- information about the exporter tool in the header of an XMI file, e.g.:
  - `<xmi:Documentation xmi:Exporter="Visual Paradigm for UML" xmi:ExporterVersion="6.3.0"/>`,
  - `<xmi:Documentation contact="www.altova.com" exporter="Altova UModel2009sp1" exporterVersion="4"/>`,
- nesting level, e.g.:
  - Visual Paradigm provides the information about the model:
 

```
<uml:Model name="Thermostat" xmi:id="ptxkdriD...">
```
  - Altova UModel provides the information about the package:
 

```
<uml:Package xmi:id="U00000001..." name="Root">
```
- instead of **packagedElement** elements in an XMI document generated by Visual Paradigm are **ownedMember** elements,
- stereotyped dependencies are defined differently, e.g. Visual Paradigm defines a stereotype separately from the dependency:
 

```
<ownedMember name="derive" xmi:id="Abstraction_derive_id"
xmi:type="uml:Stereotype"/>
<ownedMember client="Ufb5GYSD..." name="" supplier="k0b5GYSD..."
xmi:id="Tfn5GYSD..." xmi:type="uml:Abstraction">
  <xmi:Extension xmi:Extender="Visual Paradigm for UML">
    <appliedStereotype xmi:value="Abstraction_derive_id"/>
  </xmi:Extension>
</ownedMember>.
```

### 6.1.3 XMI Representation of UML Model for XTT

The examples of XMI code, generated by a UML tool Altova UModel, which represent UML artifacts used in the proposed UML representation of XTT, are shown in Tables 9 and 10.

Also in the XTT representation there are important differences between XMI documents generated by different UML tools. The main differences are:

- in Visual Paradigm *control nodes* are treated as *pseudostates* and the value of a *kind* attribute is set on *junction* when it refers to Merge or Decision Node, *join* when it concerns Join Node and *fork* when it concerns Fork Node, e.g.:
 

```
<node kind="junction" name="MergeNode2" xmi:id="rGECdriD..."
xmi:type="uml:Pseudostate">
```

```

    <xmi:Extension xmi:Extender="Visual Paradigm for UML">
        <mergeNode/>
    </xmi:Extension>
</node>,

```

- guard conditions in Altova UModel are saved in a `value` attribute whilst in Visual Paradigm they are saved in a `body` attribute.

## 6.2 HeKatE Markup Language

HeKatE Markup Language (HML) [38, 37, 27] provides a notation for XML serialization of the HeKatE rule base. This rule base is described in a human-readable HeKatE Meta Representation (HMR) [37, 27] format. DTD of HML and examples of HML documents one can find in [37, 27].

HML is divided in three subsets [37, 27]:

- *ATTML – Attribute Markup Language for describing rule attributes*,  
With this subset types and attributes can be specified.
- *ARDML – Attribute Relationship Markup Language for the ARD prototype*,  
With this subset properties, dependencies and transformations can be specified.
- *XTTML – XTT Rule Markup Language for the XTT structured rule representation*,  
With this subset XTT tables can be specified.

HML stores the knowledge in the HeKatE project in a machine-readable format which can be used to interchange knowledge between the tools from the HaDEs framework [37, 27].

## 6.3 Translation Algorithms

Next two subsections discuss creating translators from an XMI representation of the presented ARD and XTT models to the HML representation (refined version of algorithms presented in [29]) and back (refined version of the algorithm presented in [43]). They present algorithms to be used during implementation of these translators.

### 6.3.1 Translation Algorithm from XMI to ARD

In the case of XMI to ARD translation the algorithm is rather simple. Elements can be easily mapped as it is described in Section 4.2.2. So, there is only a need to specify this mapping. This can be done according to the following rules:

1. Every class without attributes becomes an ARD simple property.

For every empty-element `packagedElement` from XMI document, which has *uml:Class* value of `xmi:type` attribute:

- the name attribute becomes an attribute of a newly created `attr` HML element,
- a new `property` HML element (with an `id` having the same value as `xmi:id` of the `packagedElement`) and an `attref` element which refers to the `attr` are created.

2. Every abstract class becomes an ARD complex property.

For every `packagedElement` element from XMI document, which has *uml:Class* as the value of `xmi:type` attribute and its `isAbstract` attribute is set on *true*:

- a new `property` HML element is created (with an `id` having the same value as `xmi:id` of the `packagedElement`), and
- for every `ownedAttribute` XMI element (nested in the `packagedElement`), an `attr` HML element (with a `name` attribute having the same value as in `ownedAttribute`) and an `attref` element (inside the `property`) referring to this `attr` are created.

3. Every «derive» dependency becomes an ARD dependency.

For every `packagedElement` element from XMI document, which has *uml:Dependency* as the value of `xmi:type` attribute and has *derive* as the value of `name` attribute:

- a new `dep` HML element is created, and
- the value of `xmi:idref` attribute of `client` XMI element becomes the value of the `dependent` attribute of the `dep` HML element, and
- the value of `xmi:idref` attribute of `supplier` XMI element becomes the value of the `independent` attribute of the `dep` HML element.

In the case of the TPH, in addition, one has to use two following rules:

4. Every «refine» dependency becomes an ARD finalization transformation.

For every `packagedElement` element from XMI document, which has *uml:Dependency* as the value of `xmi:type` attribute and has *refine* as the value of `name` attribute:

- a new `hist` HML element is created, and
- the value of `xmi:idref` attribute of `client` XMI element becomes the value of the `dst` attribute of the `hist` HML element, and
- the value of `xmi:idref` attribute of `supplier` XMI element becomes the value of the `src` attribute of the `hist` HML element.

5. Every aggregation becomes a part of ARD split transformation.

For every `packagedElement` element from XMI document, which has *uml:Association* as the value of `xmi:type` attribute:

- a new `hist` HML element is created, and
- the value of `type` attribute of `ownedAttribute` XMI element (nested in the `packagedElement`) becomes the value of:
  - the `src` attribute of the `hist` HML element, if the `ownedAttribute` has an attribute aggregation having *shared* as a value,
  - the `dst` attribute of the `hist` HML element, otherwise.

### 6.3.2 Translation Algorithm from ARD to XMI

In the case of ARD to XMI translation the algorithm is very similar to the XMI to ARD algorithm. The mapping between elements is the same (described in Section 4.2.2 as well), and the translation can be done according to the following rules:

1. Every ARD property becomes a class with attributes (for complex property) or without (for simple property).

For every `property` HML element, the `packagedElement` element in XMI document, with *uml:Class* value of `xmi:type` attribute is created, and:

- if the `property` element has more than one `attref` element, the value of the `name` attribute of the `packagedElement` element starts with *Group\_* and the `isAbstract` attribute is set on *true*;, and for every `attref` element the `ownedAttribute` XMI element (nested in the `packagedElement`) is created, with a `name` attribute having the same value as in the `attr` element to which the `attref` refers.
- otherwise the value of the `name` attribute of the `packagedElement` element is the value of the `attr` element to which the `attref` refers.

2. Every ARD dependency becomes a «derive» XMI dependency.

For every `dep` HML element the `packagedElement` XMI element, which has *uml:Dependency* as the value of `xmi:type` attribute and has *derive* as the value of the `name` attribute, is created, and:

- the value of the `dependent` attribute of the `dep` HML element becomes the value of the `xmi:idref` attribute of the `client` XMI element, and
- the value of the `independent` attribute of the `dep` HML element becomes the value of the `xmi:idref` attribute of the `supplier` XMI element.

In the case of the TPH, in addition, one has to use two following rules:

3. Every ARD finalization transformation becomes the «refine» XMI dependency.

For every `hist` HML element the `packagedElement` XMI element, which has *uml:Dependency* as the value of `xmi:type` attribute and has *refine* as the value of the `name` attribute, is created, and:

- the value of the `dst` attribute of the `hist` HML element becomes the value of the `xmi:idref` attribute of the `client` XMI element, and
- the value of the `src` attribute of the `hist` HML element becomes the value of the `xmi:idref` attribute of the `supplier` XMI element.

4. Every ARD split transformation becomes an XMI aggregation.

For every `hist` HML element the `packagedElement` XMI element, which has *uml:Association* as the value of `xmi:type` attribute, is created, and:

- the value of the `src` attribute of the `hist` HML element becomes the value of the `type` attribute of the `ownedAttribute` XMI element (nested in the `packagedElement`) and the `aggregation` attribute with a value *shared* is created,
- the value of the `dst` attribute of the `hist` HML element becomes the value of the `type` attribute of the `ownedAttribute` XMI element (nested in the `packagedElement`).

### 6.3.3 Translation Algorithm from XMI to XTT

In the case of XTT representation the translation algorithm is more complicated than it was in the case of ARD. There is much more artifacts in UML representation of XTT and they have their own semantics. So, there is a need not only to specify the mapping of elements, but also some transformation has to be done. Fig. 31 shows the flowchart of the translation algorithm for XTT.

The algorithm for the lower level representation is as follows. For every table:

1. Name of the table is saved.

An `ownedBehavior` XMI element becomes `table` HML element, `xmi:id` attribute becomes `id` attribute and `name` attribute is saved.



2. Activity Parameter Nodes become attributes of the table.

- A node XMI element which has *uml:ActivityParameterNode* as the value of *xmi:type* and have a nested *outgoing* element, becomes an *attref* HML element (in subsection *precondition*) with reference to the proper *id* of attribute ARDML element which has the same name.
- A node XMI element which has *uml:ActivityParameterNode* as the value of *xmi:type* and have a nested *incoming* element, becomes an *attref* HML element (in subsection *conclusion*) with reference to the proper *id* of attribute ARDML element which has the same name.

3. In order to transform the rules, there is a need to simplify the table representation. This can be done in the following way (the process is illustrated in Fig. 32):

- (a) For every input Activity Parameter Node (a node XMI element which has as the value of *xmi:type* *uml:ActivityParameterNode* and has a nested *outgoing* element):
  - i. the proper *id* of the attribute ARDML element which has the same name is found,
  - ii. the found *id* is set as a value of the created *ref* attribute of the following Decision Node (a node XMI element which has *uml:DecisionNode* as the value of *xmi:type*),
  - iii. the Parameter Node is removed.
- (b) For every Object Flow (every edge XMI element) which has the source in a Fork Node (a node XMI element which has *uml:ForkNode* as the value of *xmi:type*):
  - i. the value of the *source* attribute of the incoming edge of the Fork Node becomes the source of the Object Flow, and
  - ii. if the incoming edge has a nested guard element, it becomes the nested element of the Object Flow,
  - iii. the Fork Node is removed.
- (c) For every Object Flow of which a target element is a Merge Node (a node XMI element which has *uml:MergeNode* as the value of *xmi:type*), the value of the *target* attribute of the outgoing edge of the Merge Node becomes the *target* of the Object Flow and the Merge Node is removed.
- (d) For every output Activity Parameter Node (a node XMI element which has as the value of *xmi:type* *uml:ActivityParameterNode* and has a nested *incoming* element):
  - i. the proper *id* of the attribute ARDML element which has the same name is found,
  - ii. the found *id* is set as a value of the created *ref* attribute of the precedent Action node (a node XMI element which has the value of *xmi:type* ending with the word "Action"),
  - iii. the Parameter Node is removed.

4. Now, it is much easier to transform the UML representation of rules to the HML.

- (a) If there is more than one input Activity Parameter Join Nodes become rule elements.  
Every Join Node (a node XMI element which has *uml:JoinNode* as the value) becomes a rule HML element and its *xmi:id* attribute becomes *id* attribute. The created rule has two nesting elements: *condition* and *decision*. For every Join Node:

- i. every value of the `value` attribute of the `guard` element in the incoming edges of the Join Node becomes the `expr` element in HML representation in the condition subsection for the proper attribute (the reference to the proper attribute shall be taken from the `ref` attribute of the Decision Node).
  - ii. every name of the `action` element in the outgoing edge of the Join Node becomes the `expr` element in HML representation in the decision subsection (the reference to the proper attribute shall be taken from the `ref` attribute of the Action).
- (b) If there is only one input Activity Parameter, Object Flows which have the source in a Decision Node become rule elements.
  - i. Every value of the `value` attribute of the `guard` element in the flow edges becomes the `expr` element in HML representation in the condition subsection for the proper attribute (the reference to the proper attribute shall be taken from the `ref` attribute of the Decision Node).
  - ii. every name of the `action` element in the outgoing edge of the Join Node becomes the `expr` element in HML representation in the decision subsection (the reference to the proper attribute shall be taken from the `ref` attribute of the Action).

At this stage of an algorithm the table representation is finished. However, there are no links between tables. The algorithm for the higher level representation is as follows:

5. In order to simplify transformation, there is a need to simplify the tree of tables representation. This can be done in the following way (the process is illustrated in Fig. 33):

- (a) For every Object Flow (every edge XMI element) of which a target element is a Join Node (a node XMI element which has `uml:JoinNode` as the value of `xmi:type`):
  - i. the value of the `target` attribute of the outgoing edge of the Join Node becomes the target of the Object Flow, and
  - ii. the Join Node is removed.
- (b) For every Object Flow which has the source in a Decision Node (a node XMI element which has `uml:DecisionNode` as the value of `xmi:type`):
  - i. the value of the `source` attribute of the incoming edge of the Decision Node becomes the source of the Object Flow, and
  - ii. a nested guard element is spared,
  - iii. the Decision Node is removed.

6. For every Action:

- without Output Pin, the `id` of the following table is found and for every rule in the table the `tabref` HML element is created with the `id` set as the value of the `ref` attribute.
- with Output Pin, the `id` of the following tables are found and according to the `guard` value of the Object Flow, the proper `transition` HML attribute is replaced by the proper `tabref` HML element with the `id` set as the value of the `ref` attribute.

#### 6.3.4 Translation Algorithm from XTT to XMI

The translation algorithm from XTT to XMI is more complicated than it was in the previous cases. This Section contains the refined version of the algorithm presented in [43] for XTT tables (with links table to table). Because of the semantics of UML artifacts, some transformations have to be done. The XMI representation of the UML activity diagram consists from nodes and edges (as it is described in Section 6.1.3).

The algorithm for the lower level representation is as follows (the algorithm is illustrated in Fig. 34). For every XTT table the Activity is created, and:

1. All condition attributes become input Activity Parameter Nodes and decision attributes become output Activity Parameter Nodes.
2. For each condition attribute (input Activity Parameter Node) the object flow goes to a decision node and for each unique value of the attribute:
  - (a) the Object Flow with this unique value as a Guard condition is introduced,
  - (b) if the value occurs frequently, the flow is finished with a Fork Node.
3. If there is more than one condition attribute, for each XTT rule a Join Node is created, and:
  - (a) for each condition attribute the proper Object Flow is connected to the Join Node (if the value of the considered condition attribute occurs frequently, the flow starts from the proper Fork Node),
  - (b) depending on the number of decision attributes:
    - i. if there is only one decision attribute: the proper Object Flow starts from the Join Node and is connected to the Action having a value corresponding to the decision attribute (if the value of the decision attribute occurs in the table more than once, a Merge Node is introduced and the connection is through this Merge Node),
    - ii. otherwise: a Fork Node is introduced and the Object Flow from the Join Node to this Fork Node is created.
4. If there is more than one decision attribute, from each Fork Node (corresponding to XTT rule) for each decision attribute the proper Object Flow from the Fork Node is introduced and connected to:
  - (a) the proper Merge Node, if the value of the decision attribute occurs in the table more than once (if the proper Merge Node does not exist yet, it should be introduced). Then, the Object Flow from the Merge Node is introduced and connected to:
  - (b) the Action (having a value corresponding to the value of the decision attribute in the rule).
5. For every decision attribute, outputs of Actions (having the values of particular attribute) are connected to a Merge Node and an Object Flow from the Merge Node leads to the corresponding output Activity Parameter Node.

The algorithm for the higher level representation is as follows. The activity diagram for the whole system is created, and:

1. for every XTT input and output system attributes, corresponding input and output Activity Parameter Nodes are created.
2. for every XTT table, the CallBehaviorAction is created, and:
  - if more than one table has the link to this table, a Join Node is created and the proper Object Flows from those tables to the Join Node are created,
  - if only one table has the link to this table, an Object Flow from that table to this table is created,
  - if the condition attribute of this table is the input system attribute, the Object Flow from the corresponding Activity Parameter Node to this table is created (if the attribute is the condition attribute in more than one table, a Fork Node is introduced and the connection is through this Fork Node).

3. To every output Activity Parameter Node an Object Flow from the proper CallBehaviorAction is created, according to the decision attributes of XTT tables, and:
  - if the attribute is the decision attribute in more than one table, a Join Node is introduced and the connection is through this Join Node,
  - if the attribute is not the only one decision attribute in the table, a Fork Node is introduced and the connection is through this Fork Node.

## 6.4 Implementation

Extensible Stylesheet Language Transformations (XSLT) [7] allows transforming of an XML document into other XML document or another format. It describes how to transform elements of one model into elements of another model and enables to convert the first model into the second one.

There are some disadvantages of that approach, such as:

- XSLT mostly has poor readability,
- The XSLT processor does not inform about e.g. application domain errors.

However, among the various methods, XSLT is particularly recommended, because [13]:

- both source and target formats are XML based,
- the concept of XSLT is matching elements of the source document with using its structure and associating them with the target document's structure, and
- there are many XSLT engines available, so there is no need to implement an evaluation engine, it is enough to describe the mappings precisely.

XSLT can be used to transform MOF-based models, which have a XMI representation. Thus, the implementation of proposed translation algorithms can be done with use of XSLT.

## 7 Closing remarks

### 7.1 Summary

In this report the problem of visual knowledge representation has been considered and the UML representation for the XTT rule design method has been presented and discussed.

Since the UML-based XTT and ARD representation is at the frontier of Knowledge and Software Engineering, in Section 2 the background of Software Engineering (Software Engineering process and basic concepts of software modeling in Software Engineering) has been presented. In Section 3 the integrated ARD design process and the concept of XTT design method have been introduced, as the proposed representation is a part of the Hybrid Knowledge Engineering Project project (HeKatE). The place of the HeKatE methodology among Knowledge and Software Engineering has been discussed as well. In Sections 4 and 5 the UML representation for ARD and XTT have been introduced, and the MOF-based metamodel for this representation has been presented as well. In Section 6 the XMI representation for the models has been discussed, the algorithms for the translation of XMI documents to a HML representation have been described, and an outline of the XSLT implementation has been provided.

Although XTT is a universal design method, for some systems this method is more suitable than for others, e.g. for the so-called expert systems. Such systems support human's decisions when making a decision depends on many factors. Thermostat problem (the problem of creating a temperature control system for an office) can be an example of a problem solved with the help of expert systems. The case study of the presented UML representation for ARD and XTT, based on the example of

*Thermostat – Intelligent Temperature Control* [52], is provided in Appendix C. Unfortunately, in larger examples the UML representation is not as efficient and compact as XTT tables.

Although tables and rules seem to be irreplaceable by UML models and the proposed representation is probably not optimal, there are some advantages of using this representation. The representation is good for problems which can be divided into many small problems (problems which generate many small size XTT tables). Thanks to the hierarchy of diagrams in the model, it is possible to create a more complex XTT tree, without the risk of illegibility. Opposed to the XTT table, the presented representation provides lack of attribute values redundancy. Moreover, models enable for quick identification of the system inputs and outputs.

The main difference between the HeKatE knowledge representation and UML is that UML does not provide a design process, but only provides the notation. HeKatE, in turn, is about the integrated design process. In order to ensure a possibility of using different tools, the most popular graphical notation for software modeling has been chosen.

Preliminary results of this report have been presented in [43, 30, 31] and described in thesis [29].

## 7.2 Future Work

The work based on the report is already in progress and the proposed algorithms are being implemented in XSLT and tested. Furthermore, several issues are considered.

There is a need to check if a certain created UML model, which conforms to the proposed UML representation, is valid. Validation of the models could be provided by some application which enables to validate a model with using its metamodel. Simplistic validation can check correctness of occurring obligatory and optional elements, nesting and relationships between them. This can be done with using DTD or XML Schema, which can be created based on MOF metamodel. Some algorithms of transforming a MOF metamodel to DTD one can find in the XMI Specification [54]. However, because of many different versions of XMI serialization, this can be a difficult problem. Moreover, according to [25] the XMI specification contains some errors in the EBNF syntax definition.

On the one hand, a possibility of checking a model after translation is also considered. That could be the first step to UML model verification. In this case, there would be a transformation from XMI to HML, and then to HMR. Such a constructed representation could be validated with using Prolog language. However, it can not be so simple and even if the HML code is valid, that will not mean that the UML model is valid.

On the other hand, if there is a UML tool which can verify in details whether a model conforms to its metamodel, that will provide a simple XTT validation method. It would be possible to translate the HML representation to XMI and validate the model.

Furthermore, a UML visual representation from HML is considered as well. However, the discussed XMI representation does not contain information about the visual position and arrangement of artifacts. Although it is possible to draw the model of ARD automatically (using e.g. UMLGraph, which allows drawing of UML class diagrams from the declarative specification), such generated diagrams are often unclear and sometimes even illegible.

The XTT models are much more complex. Nonetheless, due to the fact that XTT has rigid structure (enforced by the occurrence order of elements), there is a chance of creating an algorithm of automatic model drawing from UML representation of XTT, serialized to XMI – this has to be investigated. But in fact, it is extremely difficult to produce visual output just from XMI, which describes only features of artifacts and dependencies, but not their visual aspects. A more reasonable option is to model that manually in a certain UML tool.

In the more distant future the plan involves the incorporation of a complete rule-based logic core into an OO application, implementing I/O interfaces, including the presentation layer. In this case, the UML representation of XTT could be a part of the MDA process.

Artifact	An example of an XMI 2.1 representation generated by Altova UModel 2009
Class	<code>&lt;packagedElement xmi:type="uml:Class" xmi:id="U34aa0c61..." name="Thermostat" visibility="public"/&gt;</code>
Abstract Class with attributes	<code>&lt;packagedElement xmi:type="uml:Class" xmi:id="U7116d24b..." name="Group1" visibility="public" isAbstract="true"&gt;     &lt;ownedAttribute xmi:type="uml:Property" xmi:id="U5f28f818..." name="Time" visibility="public"/&gt;     &lt;ownedAttribute xmi:type="uml:Property" xmi:id="U994fbe2c..." name="Temperature" visibility="public"/&gt; &lt;/packagedElement&gt;</code>
Attribute	<code>&lt;ownedAttribute xmi:type="uml:Property" xmi:id="U5f28f818..." name="Time" visibility="public"/&gt;</code>
Aggregation	<code>&lt;packagedElement xmi:type="uml:Association" xmi:id="Ue94b2e6b..." visibility="public"&gt;     &lt;ownedEnd xmi:type="uml:Property" isUnique="false" xmi:id="U60a9aaf9..." visibility="public" aggregation="shared" type="U0754d304..."&gt;&lt;/ownedEnd&gt;     &lt;ownedEnd xmi:type="uml:Property" xmi:id="Ub2728ee4..." visibility="protected" type="U7116d24b..."&gt;&lt;/ownedEnd&gt; &lt;/packagedElement&gt;</code>
Dependency «derive»	<code>&lt;packagedElement xmi:type="uml:Dependency" xmi:id="U81fb052e..." name="derive" visibility="public"&gt;     &lt;supplier xmi:idref="U0754d304..."/&gt;     &lt;client xmi:idref="Uef9b82c4..."/&gt; &lt;/packagedElement&gt;</code>
Dependency «refine»	<code>&lt;packagedElement xmi:type="uml:Dependency" xmi:id="Uf31d167e..." name="refine" visibility="public"&gt;     &lt;supplier xmi:idref="U34aa0c61..."/&gt;     &lt;client xmi:idref="U7116d24b..."/&gt; &lt;/packagedElement&gt;</code>

Table 8: XMI representation of UML model for ARD

Artifact	An example of an XMI 2.1 representation generated by Altova UModel 2008
Activity	<pre>&lt;ownedBehavior xmi:type="uml:Activity" xmi:id="U601b5faf..." name="Tab1"&gt;   &lt;node...   &lt;edge... &lt;/ownedBehavior&gt;</pre>
Activity Parameter (an example of Input Activity Parameter)	<pre>&lt;node xmi:type="uml:ActivityParameterNode" xmi:id="Ue04ea3e5..." name="Par2"&gt;   &lt;outgoing xmi:idref="U286dc045..." /&gt; &lt;/node&gt;</pre>
Action ( <i>CallBehaviorAction</i> )	<pre>&lt;node xmi:type="uml:CallBehaviorAction" xmi:id="U7597f886..." name="Tab2" behavior="U9aa9ddda..."&gt;   &lt;incoming xmi:idref="U4765a71f..." /&gt;   &lt;outgoing xmi:idref="U7736bfab..." /&gt; &lt;/node&gt;</pre>
Action ( <i>CallBehaviorAction</i> ) with Output Pin	<pre>&lt;node xmi:type="uml:CallBehaviorAction" xmi:id="U7f2404c0" name="Tab1" behavior="Ua445c4e8"&gt;   &lt;result xmi:type="uml:OutputPin" xmi:id="Uaa48cfb3..." name="transition"&gt;     &lt;outgoing xmi:idref="U01242fec..." /&gt;   &lt;/result&gt;   &lt;incoming xmi:idref="Uf342ec53..." /&gt; &lt;/node&gt;</pre>
Action ( <i>OpaqueAction</i> )	<pre>&lt;node xmi:type="uml:OpaqueAction" xmi:id="U083c9b10..." name="tab2"&gt;   &lt;incoming xmi:idref="U74ec6b91..." /&gt;   &lt;outgoing xmi:idref="U8a0cc494..." /&gt; &lt;/node&gt;</pre>
Decision Node	<pre>&lt;node xmi:type="uml:DecisionNode" xmi:id="Ue8bd139f..." name="DecisionNode"&gt;   &lt;incoming xmi:idref="U01242fec..." /&gt;   &lt;outgoing xmi:idref="Ua3248a75..." /&gt;   ...   &lt;outgoing xmi:idref="U7b89aad7..." /&gt; &lt;/node&gt;</pre>
Merge Node	<pre>&lt;node xmi:type="uml:MergeNode" xmi:id="Ua3219f90..." name="MergeNode"&gt;   &lt;incoming xmi:idref="U52bab33b..." /&gt;   ...   &lt;incoming xmi:idref="U375245de..." /&gt;   &lt;outgoing xmi:idref="U74ec6b91..." /&gt; &lt;/node&gt;</pre>

Table 9: XMI representation of UML model for XTT – part 1/2

Artifact	An example of an XMI 2.1 representation generated by Altova UModel 2008
Fork Node	<pre> &lt;node xmi:type="uml:ForkNode" xmi:id="Uba703b0a..." name="ForkNode"&gt;   &lt;incoming xmi:idref="U1d41380e..." /&gt;   &lt;outgoing xmi:idref="U718d4b07..." /&gt;   ...   &lt;outgoing xmi:idref="Uc2a6e5d8..." /&gt; &lt;/node&gt; </pre>
Join Node	<pre> &lt;node xmi:type="uml:JoinNode" xmi:id="Ua50a0f33..." name="JoinNode"&gt;   &lt;incoming xmi:idref="U23cb726e..." /&gt;   ...   &lt;incoming xmi:idref="U01d83f38..." /&gt;   &lt;outgoing xmi:idref="U7be33ddb..." /&gt; &lt;/node&gt; </pre>
Object Flow	<pre> &lt;edge xmi:type="uml:ObjectFlow" xmi:id="U43c8507c..." source="Uc79f73d1..." target="U1fa6319a..." /&gt; </pre>
Object Flow (from Decision Node) with guards	<pre> &lt;edge xmi:type="uml:ObjectFlow" xmi:id="U7b89aad7..." source="Ue8bd139f..." target="Uc79f73d1..."&gt;   &lt;guard xmi:type="uml:LiteralString" xmi:id="U6edd0725..." visibility="public" value="tab4" /&gt; &lt;/edge&gt; </pre>

Table 10: XMI representation of UML model for XTT – part 2/2



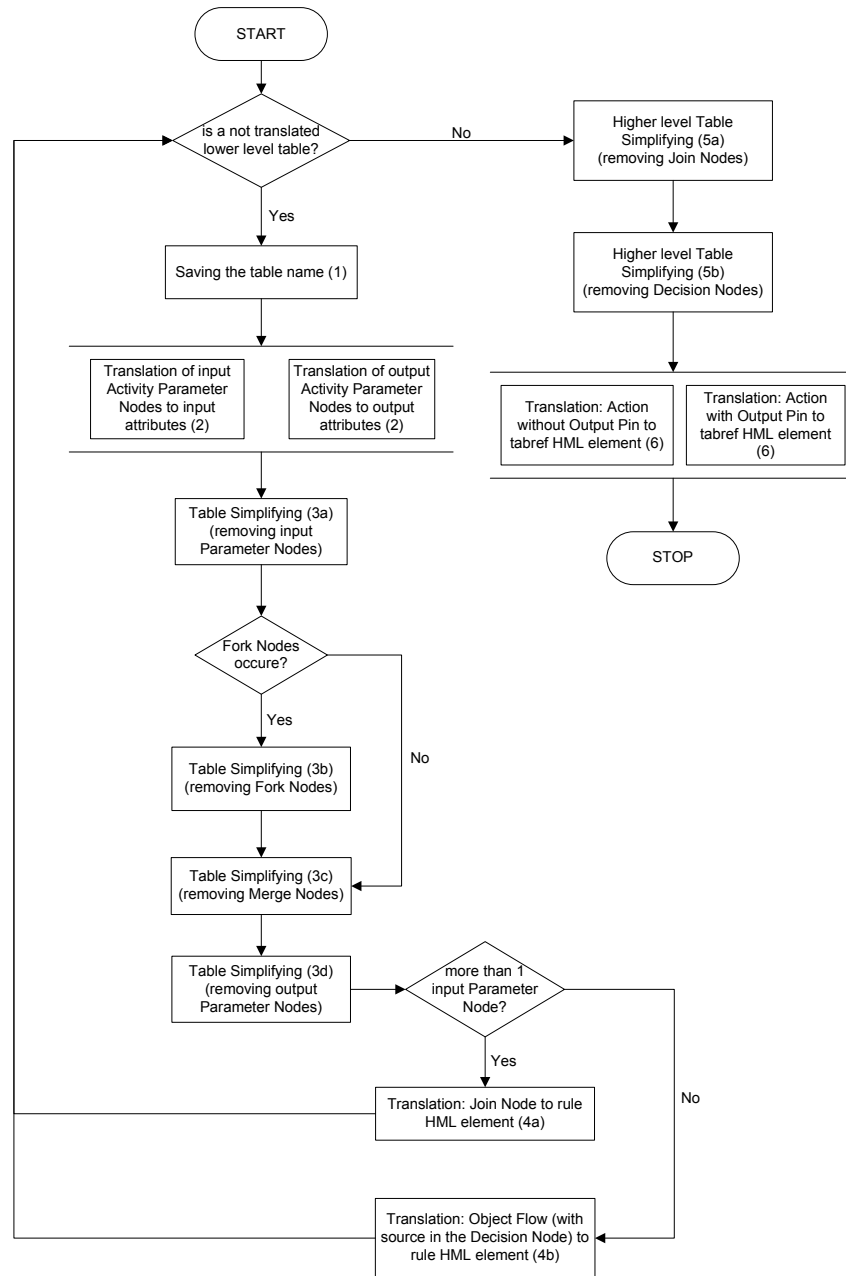


Figure 31: The flowchart shows the translation algorithm for XTT

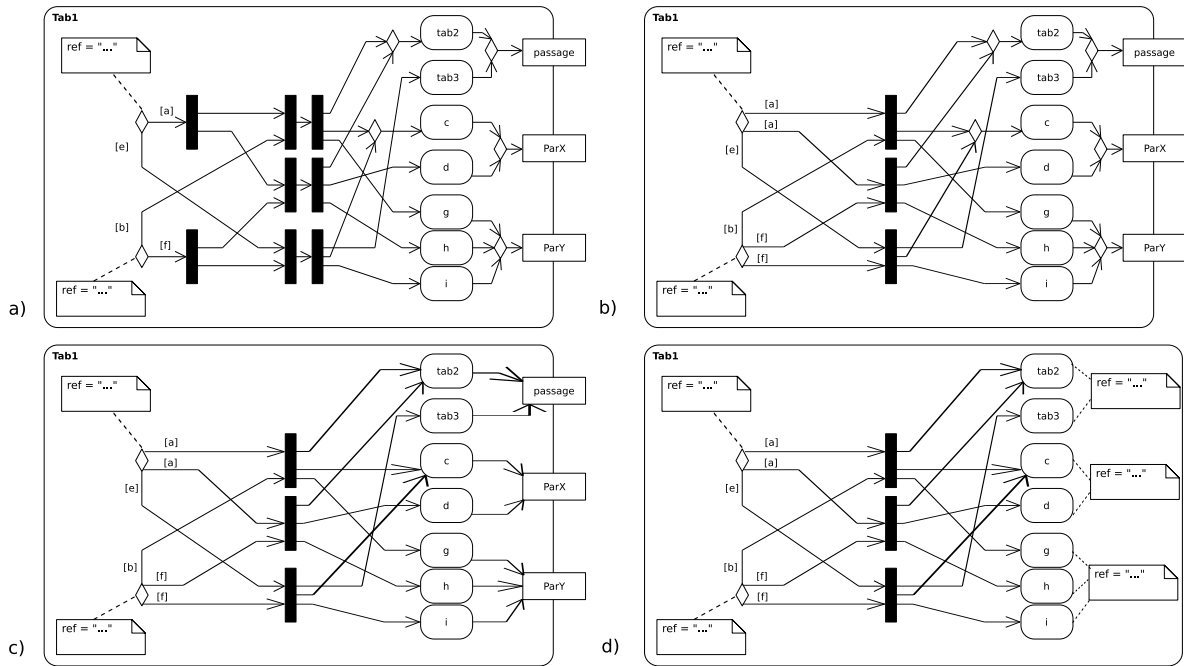


Figure 32: The proces of simplification of the table representation

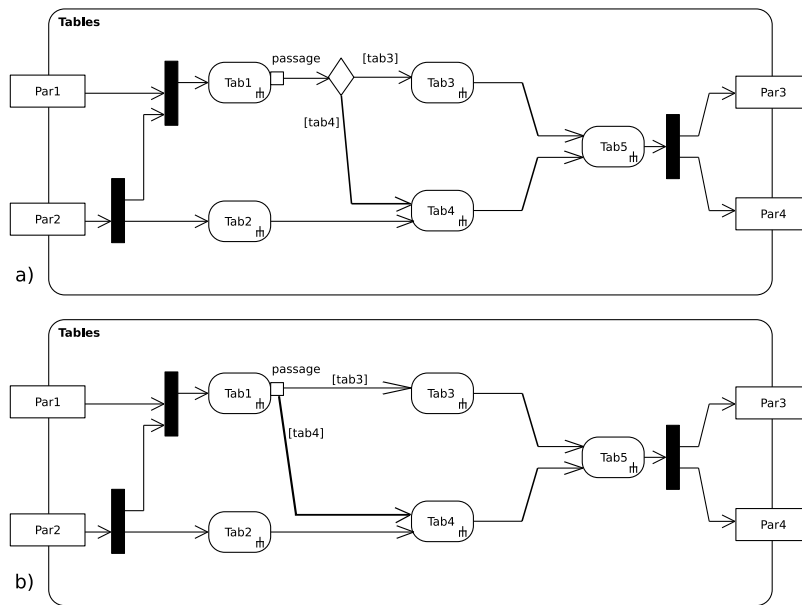


Figure 33: The proces of simplification of the tree of tables representation

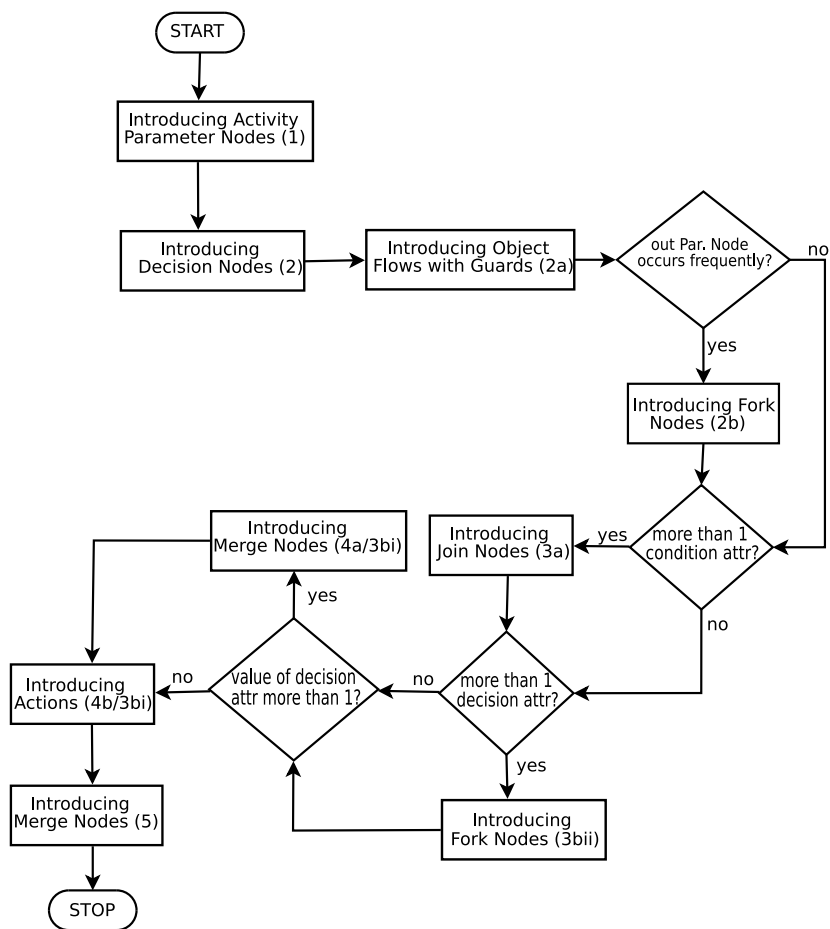


Figure 34: The translation algorithm from XTT to XMI for the lower level representation

## A OCL Constraints for ARD - Appendix

### A.1 OCL Constraints for the Metamodel of ARD Diagrams

The following statements impose the constraints on the models developed using the metamodel of ARD diagrams. These constraints are written in OCL [34].

1. *There are no ARD attributes which are functionally dependent on physical attribute.*

A class which name begins from a lower case shall not be a supplier class in any dependency.

```
context Class inv:
    self.name = self.name.toLower() implies
        self.supplierDependency->size() = 0
```

2. *An ARD attribute can not depend on itself.*

A class which is a client of the dependency have to be distinct from a supplier class of this dependency.

```
context Dependency inv:
    self.supplier <> self.client
```

3. *Every ARD complex property has two or more attributes.*

Every abstract class has no less than two attributes and its name begins with string “Group”.

```
context Class inv:
    self.isAbstract = true implies (
        self.ownedAttribute->size() > 1
        and self.name.substring(1,5) = 'Group'
    )
```

4. *Every ARD simple property has only one attribute.*

Every class which is not abstract has no attributes.

```
context Class inv:
    self.isAbstract = false implies
        self.ownedAttribute->size() = 0
```

### A.2 OCL Constraints for the Metamodel of TPH Diagrams

The following statements impose the constraints on the models developed using the metamodel of TPG diagrams. These constraints are written in OCL [34].

1. *There is only one simple ARD property as a root of TPH diagram.*

There is only one class which is not an abstract class and is not an end of an aggregation.

```
context Class inv:
    Class.allInstances()->one( c |
        c.isAbstract = false
        and c.association->size() = 0
    )
```

2. *ARD properties which are finalization products come from the finalized complex property.*

If a class has some aggregated classes, it has to be an abstract class and every attribute from this class shall become either a name of a simple class or an attribute of a new abstract class.

```
context Class inv:
  Class.allInstances()->forall( c |
    (c.association.memberEnd.aggregation = share) implies (
      c.isAbstract = true
      and ( c.name = c.association.memberEnd.name
        or c.ownedAttribute->one( a |
          a.name = c.association.memberEnd.name
        )
      )
    )
  )
)
```

3. *Physical attribute can not be split or finalized.*

A class which name begins from a lower case shall not be a supplier class in any dependency and shall not have any aggregated classes.

```
context Class inv:
  self.name = self.name.toLower() implies
    self.supplierDependency->size() = 0
    and self.association->size() = 0
```

## B OCL Constraints for XTT - Appendix

The statements presented in next sections impose the constraints on the models developed using the metamodel of XTT diagrams. These constraints are written in OCL [34].

### B.1 OCL Constraints for the Metamodel of XTT Diagrams at the Lower Level of Abstraction

OCL constraints presented in this section are based on the XTT metamodel and are introduced to enforce the order of nodes. The required order of nodes for XTT model at the lower abstraction level is as follows (nodes which can occur optional are in brackets):

Activity Parameter Node  $\rightarrow$  Decision Node  $\rightarrow$  (Fork Node)  $\rightarrow$  Join Node<sup>5</sup>  
 $\rightarrow$  Fork Node<sup>6</sup>  $\rightarrow$  (Merge Node)  $\rightarrow$  Action  $\rightarrow$  (Merge Node)  $\rightarrow$  Activity  
 Parameter Node.

OCL expressions for the metamodel of XTT based on the required order:

#### 1. Constraints for Activity Parameter Node:

- Every parameter node (`ActivityParameterNode`) has either exactly one incoming edge or exactly one outgoing edge.
- Incoming edge to every parameter node goes from either a merge node (`MergeNode`) or an action (`Action`).
- Outgoing edge from every parameter node goes to a decision node.

```
context ActivityParameterNode inv:
  self.outgoing->size()=1 xor self.incoming->size()=1
  self.incoming->forall( edge |
    edge.source.ocIsTypeOf(MergeNode)
    xor edge.source.ocIsKindOf(Action)
  )
  self.outgoing->forall( edge |
    edge.target.ocIsTypeOf(DecisionNode)
  )
```

#### 2. Constraints for Decision Node:

- Every decision node (`DecisionNode`) has exactly one incoming edge.
- Incoming edge to every decision node goes from a parameter node.
- Outgoing edges from every decision node go to either fork nodes (`ForkNode`) or join nodes (`JoinNode`).
- Outgoing edges from every decision have to contain a guard condition.

```
context DecisionNode inv:
  self.incoming->size()=1
  self.incoming->forall( edge |
    edge.source.ocIsTypeOf(ActivityParameterNode)
  )
```

<sup>5</sup> Element occurs when there is more than one input activity parameter node.

<sup>6</sup> Element occurs when there is more than one output activity parameter node or there is the *transition* parameter node (which enables the transition to different tables).

```

self.outgoing->forAll( edge |
    edge.target.ocIsTypeOf(ForkNode)
    xor edge.target.ocIsTypeOf(JoinNode)
)
self.outgoing->forAll( edge |
    edge.guard.stringValue().notEmpty()
)

```

### 3. Constraints for Fork Node:

- Incoming edge to every fork node goes from either a decision node or a join node.
- If incoming edge to a fork node goes from a decision node, outgoing edges from that fork node shall go to distinct join nodes.
- If incoming edge to a fork node goes from a join node, each outgoing edge from that fork node shall go either to a merge node or an action.

```

context ForkNode inv:
    self.incoming->forAll( edge |
        edge.source.ocIsTypeOf(DecisionNode)
        xor edge.source.ocIsTypeOf(ForkNode)
    )
    self.incoming->forAll( edge |
        edge.source.ocIsTypeOf(DecisionNode)) implies
        self.outgoing->forAll( edge |
            edge.target.ocIsTypeOf(JoinNode)
        )
    self.incoming->forAll( edge |
        edge.source.ocIsTypeOf(JoinNode)) implies
        self.outgoing->forAll( edge |
            edge.target.ocIsTypeOf(MergeNode)
            xor edge.target.ocIsKindOf(Action)
        )

```

### 4. Constraints for Join Node:

- Incoming edge to every join node goes from either a decision node or a fork node.
- Outgoing edge from every join node goes to a fork node.

```

context JoinNode inv:
    self.incoming->forAll( edge |
        edge.source.ocIsTypeOf(DecisionNode)
        xor edge.source.ocIsTypeOf(ForkNode)
    )
    self.outgoing->forAll( edge |
        edge.target.ocIsTypeOf(ForkNode)
    )

```

### 5. Constraints for Merge Node:

- Outgoing edge from every merge node goes to either an action or a parameter node.

- If outgoing edge from a merge node goes to an action, incoming edges to that merge node shall go from distinct fork nodes.
- If outgoing edge from a merge node goes to a parameter node, incoming edges to that merge node shall go from distinct actions.

```
context MergeNode inv:
  self.outgoing->forAll( edge |
    edge.target.ocIsKindOf(Action)
    xor edge.target.ocIsTypeOf(ActivityParameterNode)
  )
  self.outgoing->forAll( edge |
    edge.target.ocIsKindOf(Action)) implies
    self.incoming->forAll( edge |
      edge.source.ocIsTypeOf(ForkNode)
    )
  self.outgoing->forAll( edge |
    edge.target.ocIsTypeOf(ActivityParameterNode)) implies
    self.incoming->forAll( edge |
      edge.source.ocIsKindOf(Action)
    )
)
```

#### 6. Constraints for Action:

- Every action node has exactly one incoming and one outgoing edge (flow).
- Incoming edge to every action goes from either a merge node or a fork node.
- Outgoing edge from every action goes to either a merge node or a parameter node.

```
context Action inv:
  self.outgoing->size()=1 and self.incoming->size()=1
  self.incoming->forAll( edge |
    edge.source.ocIsTypeOf(MergeNode)
    xor edge.source.ocIsTypeOf(ForkNode)
  )
  self.outgoing->forAll( edge |
    edge.target.ocIsTypeOf(MergeNode)
    xor edge.target.ocIsTypeOf(ActivityParameterNode)
  )
)
```

## B.2 OCL Constraints for the Metamodel of XTT Diagrams at the Higher Level of Abstraction

OCL constraints presented in this section are based on the XTT metamodel and are introduced to enforce the order of nodes. The required order of nodes for XTT model at the higher abstraction level is as follows (nodes which can occur optional are in brackets):

Activity Parameter Node → (Fork Node) → \* (Join Node) → Action →

- (Decision Node)<sup>7</sup> → back to \*,
- (Fork Node) → Activity Parameter Node.

OCL expressions for the metamodel of XTT based on the required order:

---

<sup>7</sup> Element occurs when previous Action has Output Pin.



## 1. Constraints for Activity Parameter Node:

- Every parameter node has either exactly one incoming edge or exactly one outgoing edge.
- Incoming edge to every parameter node goes from either a fork node or an action.
- Outgoing edge from every parameter node goes to either a fork node, a join node or an action.

```
context ActivityParameterNode inv:
  self.outgoing->size()=1 xor self.incoming->size()=1
  self.incoming->forall( edge |
    edge.source.ocIsTypeOf(ForkNode)
    xor edge.source.ocIsTypeOf(CallBehaviorAction)
  )
  self.outgoing->forall( edge |
    edge.target.ocIsTypeOf(ForkNode)
    xor edge.target.ocIsTypeOf(JoinNode)
    xor edge.target.ocIsTypeOf(CallBehaviorAction)
  )
```

## 2. Constraints for Fork Node:

- Incoming edge to every fork node goes from either an activity parameter node or an action.
- If incoming edge to a fork node goes from an activity parameter node, outgoing edges from that fork node shall go to join nodes or actions.
- If incoming edge to a fork node goes from an action, each outgoing edge from that fork node shall go to an activity parameter node.

```
context ForkNode inv:
  self.incoming->forall( edge |
    edge.source.ocIsTypeOf(ActivityParameterNode)
    xor edge.source.ocIsTypeOf(CallBehaviorAction)
  )
  self.incoming->forall( edge |
    edge.source.ocIsTypeOf(ActivityParameterNode)) implies
    self.outgoing->forall( edge |
      edge.target.ocIsTypeOf(JoinNode)
      xor edge.target.ocIsTypeOf(CallBehaviorAction)
    )
  self.incoming->forall( edge |
    edge.source.ocIsTypeOf(CallBehaviorAction)) implies
    self.outgoing->forall( edge |
      edge.target.ocIsTypeOf(ActivityParameterNode)
    )
```

## 3. Constraints for Decision Node:

- Every decision node (DecisionNode) has exactly one incoming edge.
- Incoming edge to every decision node goes from a parameter node.
- Outgoing edges from every decision node go to either fork nodes (ForkNode) or join nodes (JoinNode).

- Outgoing edges from every decision have to contain a guard condition.

```
context DecisionNode inv:
  self.incoming->size()=1
  self.incoming->forall( edge |
    edge.source.ocIsTypeOf(ActivityParameterNode)
  )
  self.outgoing->forall( edge |
    edge.target.ocIsTypeOf(ForkNode)
    xor edge.target.ocIsTypeOf(JoinNode)
  )
  self.outgoing->forall( edge |
    edge.guard.stringValue().notEmpty()
  )
```

#### 4. Constraints for Join Node:

- Incoming edge to every join node goes from either an activity parameter node, a fork node, a decision node or an action.
- Outgoing edge from every join node goes to an action.

```
context JoinNode inv:
  self.incoming->forall( edge |
    edge.source.ocIsTypeOf(ActivityParameterNode)
    xor edge.source.ocIsTypeOf(ForkNode)
    xor edge.source.ocIsTypeOf(DecisionNode)
    xor edge.source.ocIsTypeOf(CallBehaviorAction)
  )
  self.outgoing->forall( edge |
    edge.target.ocIsTypeOf(CallBehaviorAction)
  )
```

#### 5. Constraints for Decision Node:

- Every decision node has exactly one incoming edge.
- Incoming edge to every decision node goes from an output pin of an action.
- Outgoing edges from every decision node go to either join nodes or actions.
- Outgoing edges from every decision have to contain a guard condition.
- The value of the guard condition has a name of the target action.

```
context DecisionNode inv:
  self.incoming->size()=1
  self.incoming->forall( edge |
    edge.source.ocIsTypeOf(OutputPin)
  )
  self.outgoing->forall( edge |
    edge.target.ocIsTypeOf(JoinNode)
    xor edge.target.ocIsTypeOf(CallBehaviorAction)
  )
  self.outgoing->forall( edge |
```

```

        edge.guard.stringValue().notEmpty()
    )
    self.outgoing->forall( edge |
        edge.target.ocIsTypeOf(CallBehaviorAction) implies
            edge.guard.stringValue() = edge.target.name
        edge.target.ocIsTypeOf(JoinNode) implies
            edge.guard.stringValue() = edge.target.edge.target.name
    )

```

## 6. Constraints for CallBehaviorAction:

- Every action node has exactly one incoming and one outgoing edge (flow).
- Incoming edge to every action goes from either an activity parameter node, a fork node or a join node.
- Outgoing edge from every action goes to either a decision node, a join node, an action, a fork node or an activity parameter node.
- An action can have one output pin (with the name 'transition').

```

context CallBehaviorAction inv:
    self.outgoing->size()=1 and self.incoming->size()=1
    self.incoming->forall( edge |
        edge.source.ocIsTypeOf(ActivityParameterNode)
        xor edge.source.ocIsTypeOf(ForkNode)
        xor edge.source.ocIsTypeOf(JoinNode)
    )
    self.outgoing->forall( edge |
        edge.target.ocIsTypeOf(DecisionNode)
        xor edge.target.ocIsTypeOf(JoinNode)
        xor edge.target.ocIsTypeOf(CallBehaviorAction)
        xor edge.target.ocIsTypeOf(ForkNode)
        xor edge.target.ocIsTypeOf(ActivityParameterNode)
    )
    self.output->notEmpty() implies
        self.output.name = 'transition'

```

## C Thermostat Case Study - Appendix

*Thermostat* problem has been described in [52]. The problem concerns a temperature control system for an office. The goal of the system is to set temperature at a certain set point. The temperature is set depending on the particular part of the week, current season, and working time. Current time (hour) and date (day and month) are on input of the system and the most appropriate temperature for that conditions, expressed in Celsius degrees, is on its output.

The detailed specification of the attributes, obtained in the ARD design process, is shown in Table 11 [26]. Such a specification can be also presented in a UML class diagram, an example of which is shown in Fig. 35. Enumerative attributes are presented as Enumerations (classes with the «enumeration» stereotype). Other kinds of attributes can be inherited from Primitive Types [59] (such as: *Integer*, *Boolean*, *String*, *Unlimited Natural*) or user defined Data Types, e.g. *Real*, *Time*.

Name	Type	Domain	Description
day	enumerative	{monday, tuesday, wednesday, thursday, friday, saturday, sunday}	The name of the current day. The attribute defines the today's value.
time	integer	<0; 24>	The value of the hour. The attribute defines the operation's value.
month	enumerative	{january, february, march, april, may, june, july, august, september, october, november, december}	The name of the current month. The attribute defines the season's value.
season	enumerative	{spring, summer, autumn, winter}	The name of the season.
today	enumerative	{workday, weekend}	The type of the day.
operation	enumerative	{bizhrs, nbizhrs}	This attribute describes if the current hour is the working time (bizhrs) or not (nbizhrs).
thermostat	integer	{14, 15, 16, 18, 20, 24, 27}	The value of the thermostat setting.

Table 11: XTT Thermostat attributes specification [26]

When all conceptual attributes are finalized to the physical attributes, the schema of XTT can be derived. Such a schema has to be fulfilled with rules. There are 18 rules describing the *Thermostat* system. Due to the Australian author of them, the rules 7–10 are correct from his point of view.

The original *Thermostat* system rules are as follows:

```
Rule: 1
if    'the day' is 'Monday'
or    'the day' is 'Tuesday' or 'the day' is 'Wednesday'
or    'the day' is 'Thursday' or 'the day' is 'Friday'
then  'today' is 'a workday'
```

```
Rule: 2
if    'the day' is 'Saturday'
or    'the day' is 'Sunday'
```

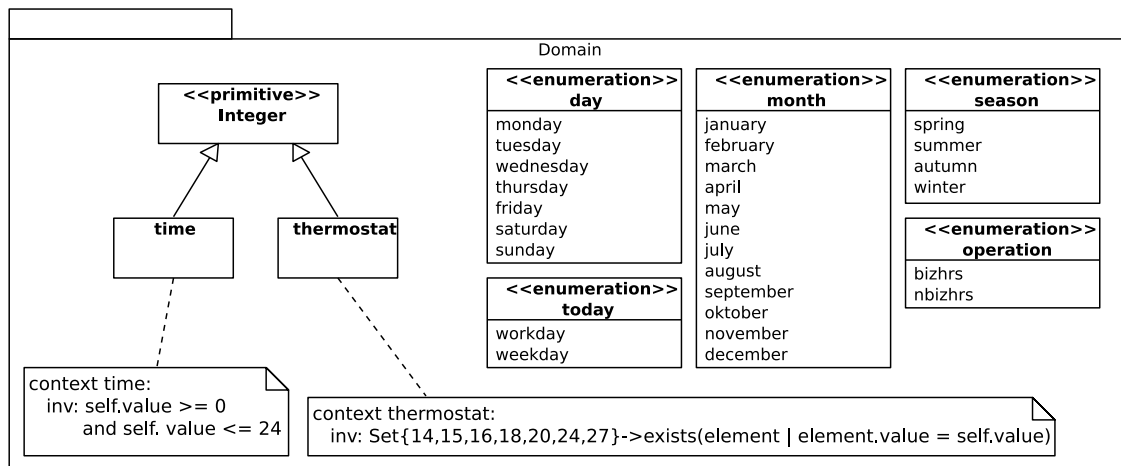


Figure 35: XTT Thermostat attributes specification UML representation

then 'today' is 'the weekend'

Rule: 3

```

if 'today' is 'workday'
and 'the time' is 'between 9 am and 5 pm'
then 'operation' is 'during business hours'

```

Rule: 4

```

if 'today' is 'workday'
and 'the time' is 'before 9 am'
then 'operation' is 'not during business hours'

```

Rule: 5

```

if 'today' is 'workday'
and 'the time' is 'after 5 pm'
then 'operation' is 'not during business hours'

```

Rule: 6

```

if 'today' is 'weekend'
then 'operation' is 'not during business hours'

```

Rule: 7

```

if 'the month' is 'January'
or 'the month' is 'February or the month is December'
then 'the season' is 'summer'

```

Rule: 8

```

if 'the month' is 'March'
or 'the month' is 'April or the month is May'
then 'the season' is 'autumn'

```

Rule: 9

```

if 'the month' is 'June'
or 'the month' is 'July' or 'the month' is August'
then 'the season' is 'winter'

```

```
Rule: 10
if    'the month' is 'September'
or    'the month' is 'October or the month is November'
then  'the season' is 'spring'

Rule: 11
if    'the season' is 'spring'
and   'operation' is 'during business hours'
then  'thermostat_setting' is '20 degrees'

Rule: 12
if    'the season' is 'spring'
and   'operation' is 'not during business hours'
then  'thermostat_setting' is '15 degrees'

Rule: 13
if    'the season' is 'summer'
and   'operation' is 'during business hours'
then  'thermostat_setting' is '24 degrees'

Rule: 14
if    'the season' is 'summer'
and   'operation' is 'not during business hours'
then  'thermostat_setting' is '27 degrees'

Rule: 15
if    'the season' is 'autumn'
and   'operation' is 'during business hours'
then  'thermostat_setting' is '20 degrees'

Rule: 16
if    'the season' is 'autumn'
and   'operation' is 'not during business hours'
then  'thermostat_setting' is '16 degrees'

Rule: 17
if    'the season' is 'winter'
and   'operation' is 'during business hours'
then  'thermostat_setting' is '18 degrees'

Rule: 18
if    'the season' is 'winter'
and   'operation' is 'not during business hours'
then  'thermostat_setting' is '14 degrees'
```

Fig. 36 shows the steps of the ARD design process rendered by the HQEd and the same process, but with using the proposed UML representation.

The TPH diagram, which shows the gradual refinement of a designed system, and its corresponding UML representation are shown consecutively in Fig. 37a and 37b. Such diagram enables to identify the origin of given properties. It captures the information about changes at consecutive diagram levels. A tree-like structure denotes how particular property has been split or finalized. The ARD

split transformation is presented in UML diagram as an aggregation while the ARD finalization transformation is presented as a «refine» dependency

The detailed description of *Thermostat – Intelligent Temperature Control* problem is presented in [52]. Fig. 38 shows already the fulfilled XTT schema. Such a schema can be automatically generated by HQEd [26], based on the created ARD model. The schema is fulfilled with certain rules.

Corresponding UML representation is shown in Fig. 39 - 43.

**Evaluation** As one can see, the proposed UML model of ARD and TPH diagrams is very similar to the original ARD and TPH diagrams. Although the representation is very simple, it presents every aspects of ARD and TPH. Moreover, the UML representation of TPH does not require introducing a new style of arrows (e.g. dashed) in the diagrams in order to show the difference between ARD dependencies and TPH transformations in TPH diagram. It is so, because in the proposed UML representation ARD dependencies are represented differently than TPH transformations.

In case of XTT diagrams, the proposed UML model is not so efficient. However, it allows for hierarchical division on many diagrams. If a table is small, the corresponding diagram is also small and very intuitive (as one can see in Fig. 40 - 42). Unfortunately, for tables containing more rules, the proposed representation is not so readable. When the number of rules grows, tables seem to be irreplaceable.

An advantage of using the proposed UML representation is that it lacks of attribute values redundancy, as the original XTT tables. Thanks the two level hierarchy, it is much easier to identify the inputs and outputs of the thermostat system.

Although the *Thermostat* case study is very simple, it shows that even so simple problem as temperature control can be divided into many smaller problems. The proposed UML representation can be useful, especially for such divisible problems.

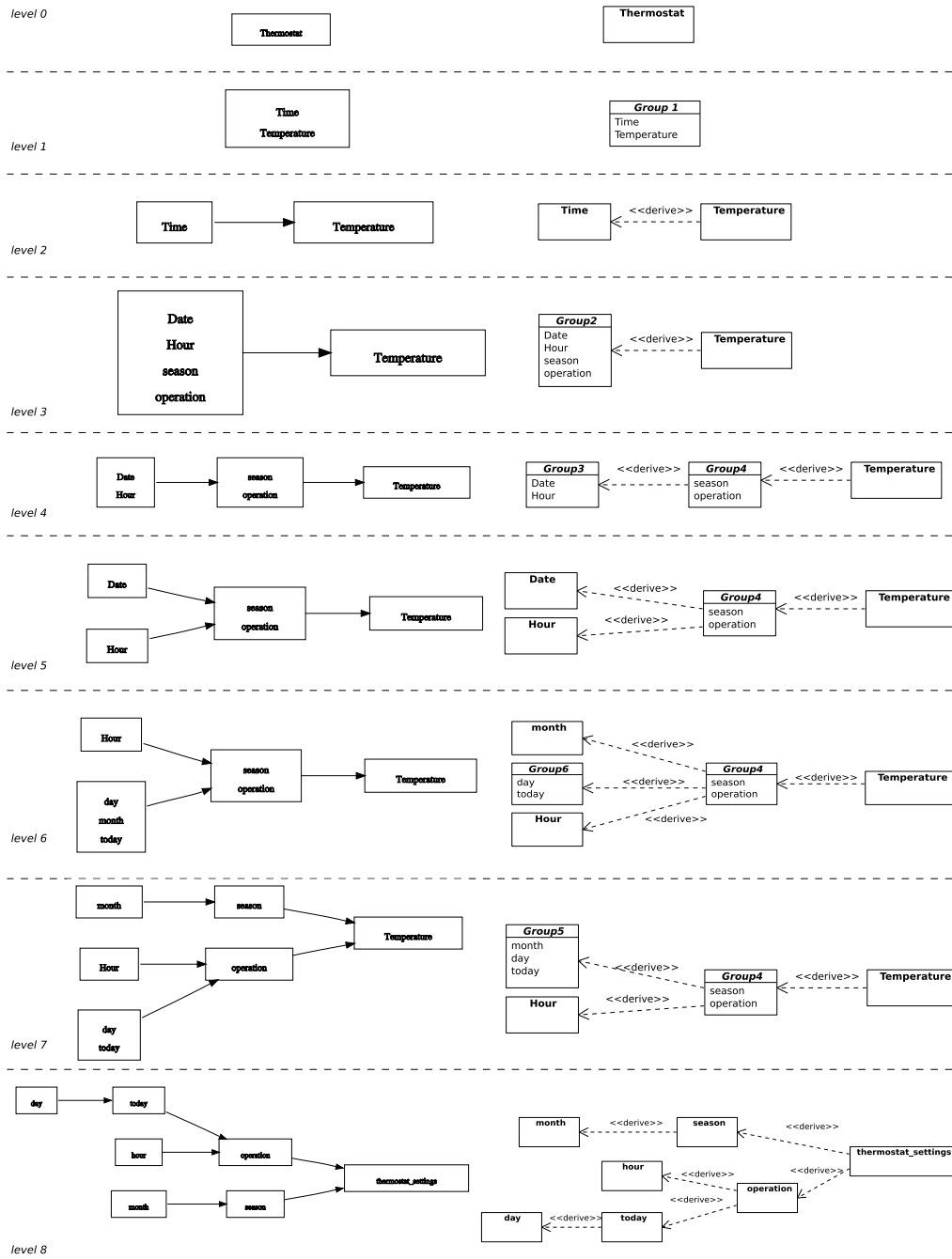


Figure 36: The steps of the ARD design process a) original b) UML representation



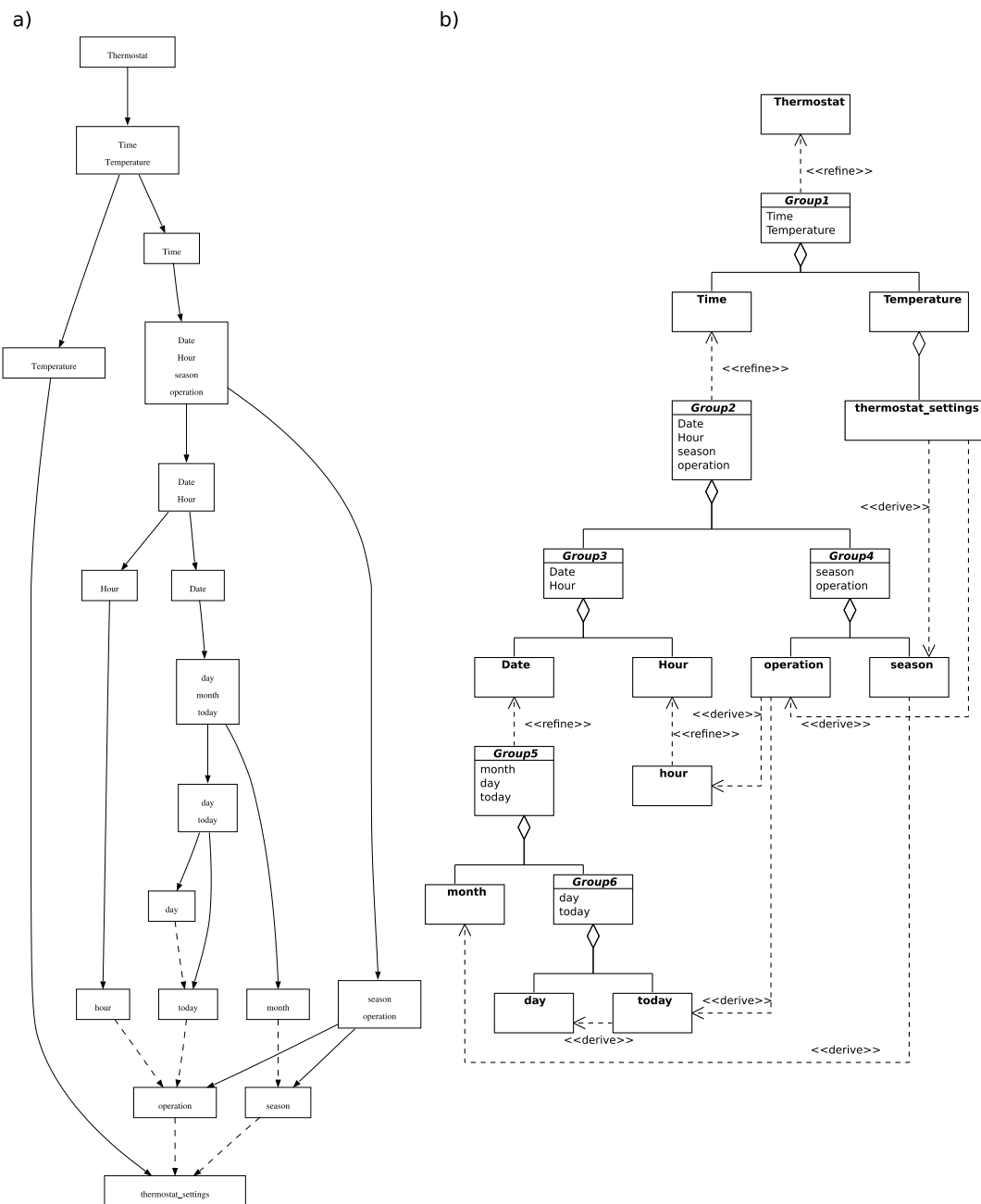


Figure 37: The TPH diagram and its corresponding UML representation a) original b) UML representation

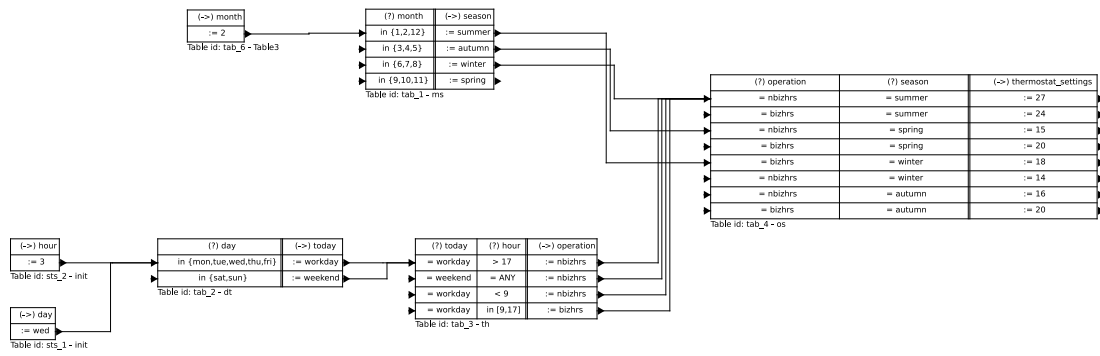


Figure 38: The XTT model of Thermostat [26]

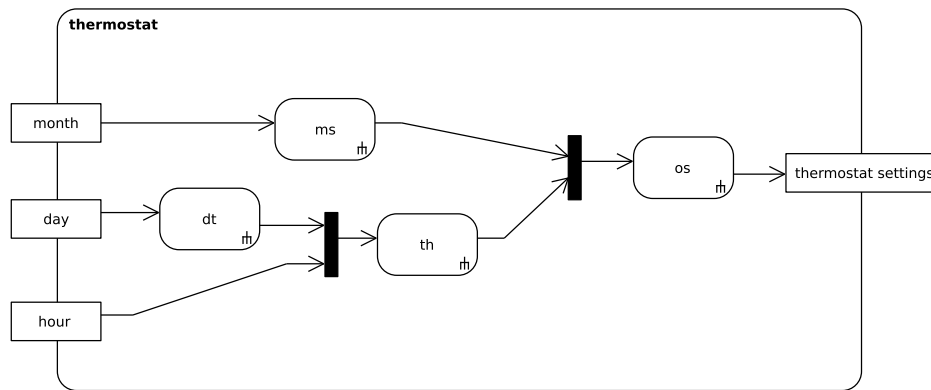
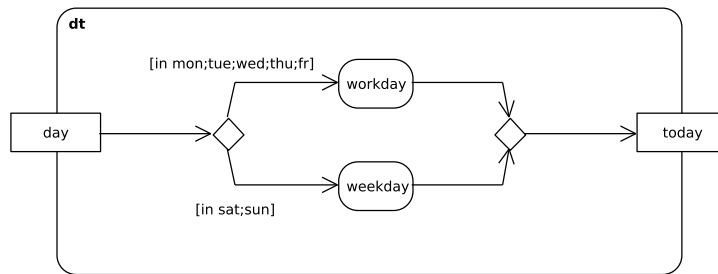
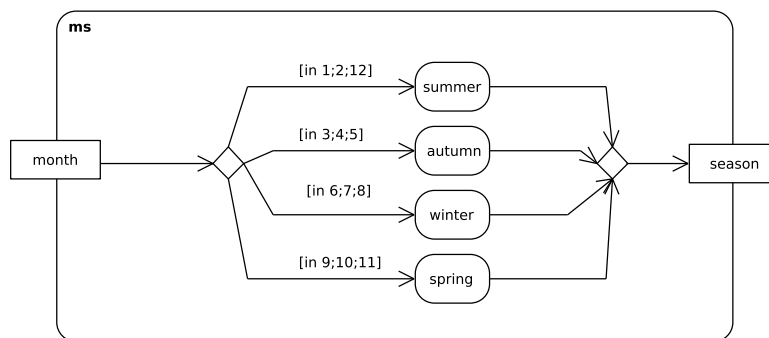


Figure 39: The UML representation of the whole XTT of Thermostat

Figure 40: The UML representation of the *dt* tableFigure 41: The UML representation of the *ms* table

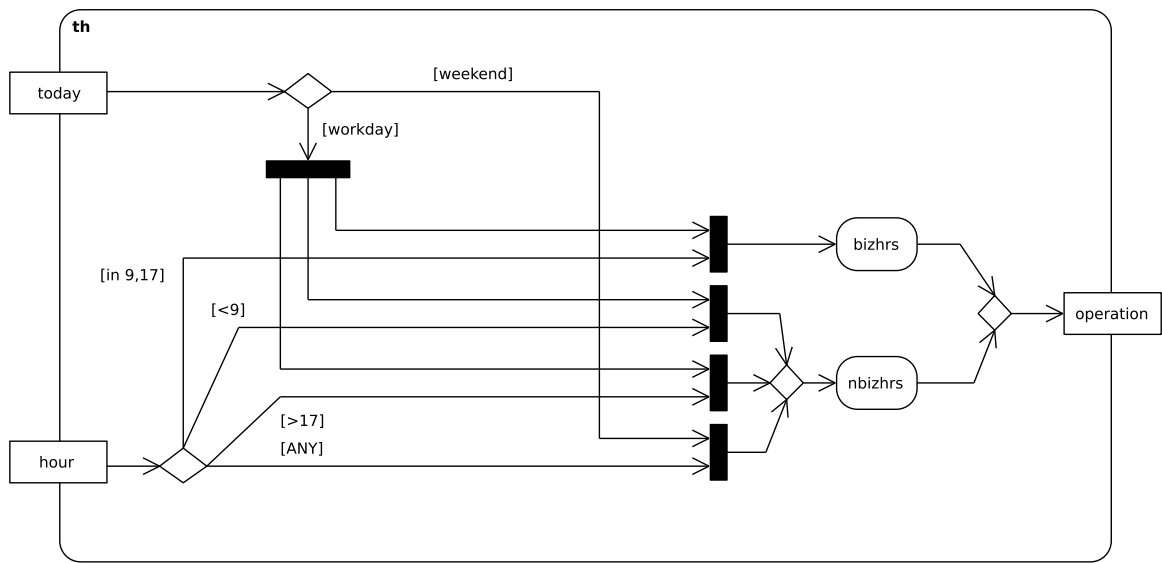


Figure 42: The UML representation of the *th* table

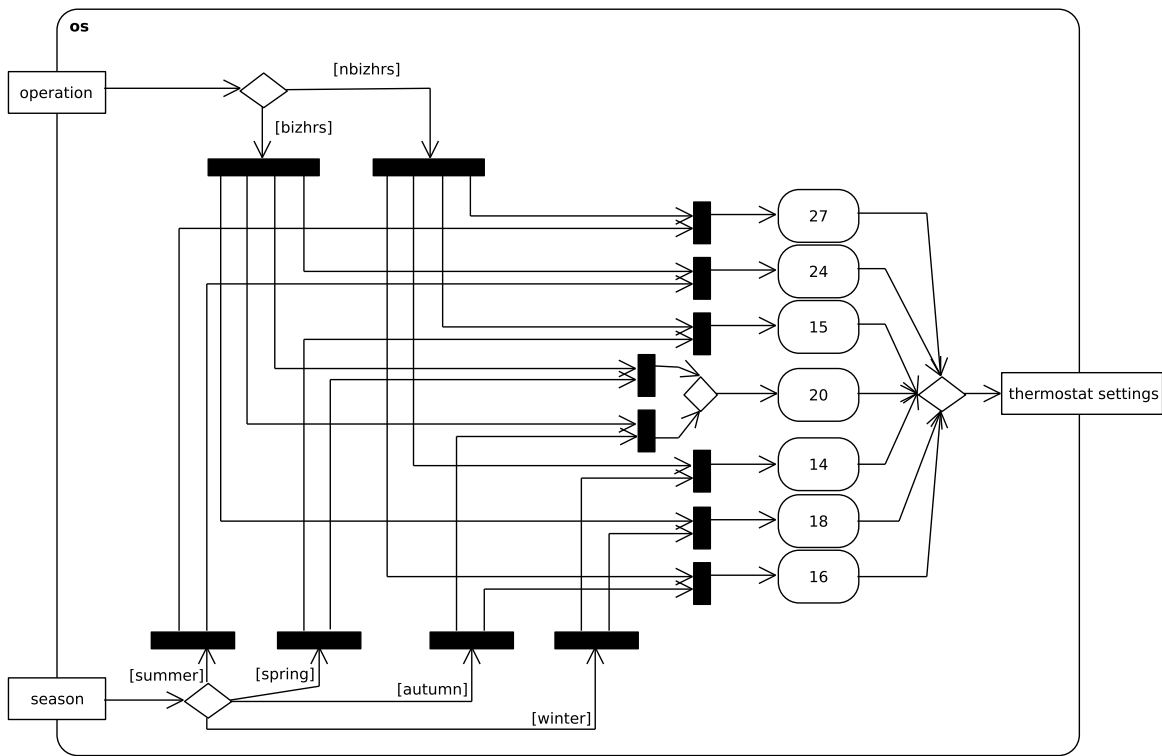


Figure 43: The UML representation of the *os* table

## References

- [1] S. W. Ambler. Business Rules. <http://www.agilemodeling.com/artifacts/businessRule.htm>, 2003.
- [2] M. Benoit. Working XML: UML, XML, and code generation, part 2. <http://www.ibm.com/developerworks/xml/library/x-wxxm24/>, 2004.
- [3] D. Bjørner. *Software engineering 3. Domains, Requirements, and Software Design*. Springer-Verlag Berlin Heidelberg, 2006.
- [4] B. Boehm. *A Spiral Model for Software Development and Enhancement*. IEEE Computer, May 1988.
- [5] S. Brockmans. Metamodel-based knowledge representation. Licentiaat Informatica. Technical report, University of Karlsruhe, Karlsruhe, 2007.
- [6] A. Burns, B. Dobbing, and T. Vardanega. Defining business rules ~ what are they really? Technical Report revision 1.3, The Business Rules Group, 2000.
- [7] J. Clark. XSL Transformations (XSLT) version 1.0 W3C recommendation 16 november 1999. Technical report, World Wide Web Consortium (W3C), 1999.
- [8] B. Daum and U. Merten. *System architecture with XML*. Morgan Kaufmann, 2002.
- [9] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 3rd edition, 2003.
- [10] D. S. Frankel. *Model Driven Architecture. Applying MDA to Enterprise Computing*. Wiley Publishing, Indianapolis, 2003.
- [11] E. Freeman, K. Sierra, and B. Bates. *Head First design patterns*. O'Reilly, 2004.
- [12] D. Gasevic, D. Djuric, and D. V. *Model Driven Architecture and Ontology Development*. Springer, 2006.
- [13] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In *Graph transformation: first international conference, ICGT 2002*, pages 252–265, Barcelona, Spain, October 2002.
- [14] J. C. Giarratano and G. D. Riley. *Expert Systems*. Thomson, 2005.
- [15] B. R. Group. The business rules manifesto. <http://www.businessrulesgroup.org/brmanifesto.htm>, 2002.
- [16] D. A. Gustafson. *Theory and Problems of Software Engineering*. McGraw-Hill, 2002.
- [17] K. Hamilton and R. Miles. *Learning UML 2.0*. O'Reilly, 2006.
- [18] B. Henderson-Sellers, C. Atkinson, T. Kühne, and C. Gonzalez-Perez. Understanding meta-modelling, October 2003.
- [19] M. G. Hinchey, J. L. Rash, and C. A. Rouff. Requirements to design to code: Towards a fully formal approach to automatic code generation. Technical Report TM-2005-212774, NASA Goddard Space Flight Center, Greenbelt, MD, 2004.
- [20] P. Horowitz and W. Hill. *The Art of Electronics*. Cambridge University Press, 1989.

- [21] J. Hunt. *Guide to the Unified Process featuring UML, Java and Design Patterns*. Springer, 2003.
- [22] R. W. G. I1. A UML-based rule modeling language. <http://oxygen.informatik.tu-cottbus.de/reverse-i1/?q=URML>, 2006.
- [23] J. P. Ignizio. *An Introduction To Model Driven Architecture. Applying MDA to Enterprise Computing*. David S. Frankel, Wiley Publishing, Inc., Indianapolis 2003 *Expert Systems. The Development and Implementation of Rule-Based Expert Systems*. McGraw-Hill, 1991.
- [24] International Organization for Standardization. *Information technology – Meta Object Facility (MOF)*, 2005.
- [25] S. L. Jim. From UML diagrams to behavioural source code. Master’s thesis, Universiteit van Amsterdam, July 2006. Supervisor: P. Klint. <http://homepages.cwi.nl/~paulk/thesesMasterSoftwareEngineering/2006/SabrinaJim.pdf>.
- [26] K. Kaczor and G. J. Nalepa. Design and implementation of hqed, the visual editor for the XTT+ rule design method. Technical Report CSLTR 02/2008, AGH University of Science and Technology, december 2008.
- [27] K. Kaczor and G. J. Nalepa. HaDEs – presentation of the HeKatE design environment. In J. Baumeister and G. J. Nalepa, editors, *5th Workshop on Knowledge Engineering and Software Engineering (KESE2009) at the 33rd German conference on Artificial Intelligence: September 15, 2009, Paderborn, Germany*, pages 57–62, Paderborn, Germany, 2009.
- [28] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.
- [29] K. Kluza. UML-based knowledge representation for XTT+ and ARD+. Master’s thesis, AGH University of Science and Technology, July 2009. Supervisor: G. J. Nalepa.
- [30] K. Kluza and G. J. Nalepa. Metody i narzędzia wizualnego projektowania reguł decyzyjnych. In *Inżynieria Wiedzy i Systemy Ekspertowe*, 2009.
- [31] K. Kluza and G. J. Nalepa. MOF-based metamodeling for the XTT knowledge representation. In *CMS’09 : Computer Methods and Systems : 7th conference : 26–27 November 2009, Kraków, Poland*. AGH University of Science and Technology, Cracow, Oprogramowanie Naukowo-Techniczne, 2009.
- [32] A. Ligeża. *Logical Foundations for Rule-Based Systems*. Springer-Verlag, Berlin, Heidelberg, 2006.
- [33] S. Lukichev and G. Wagner. Visual rules modeling. In *Sixth International Andrei Ershov Memorial Conference Perspectives of System Informatics, Novosibirsk, Russia, June 2006*, LNCS. Springer, 2005.
- [34] Object Management Group. Object constraint language version 2.0. Technical report, OMG, May 2006.
- [35] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley Professional, 1st edition, 2002.
- [36] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. OMG, 2003.
- [37] G. Nalepa. HeKatE design environment. <https://ai.ia.agh.edu.pl/wiki/hekate:hades>, 2009. HeKatE Project Homepage.

- [38] G. Nalepa. HeKatE markup language. [https://ai.ia.agh.edu.pl/wiki/hekate:hekate\\_markup\\_language](https://ai.ia.agh.edu.pl/wiki/hekate:hekate_markup_language), 2009. HeKatE Project Homepage.
- [39] G. Nalepa. HeKatE meta representation. <https://ai.ia.agh.edu.pl/wiki/hekate:hmr>, 2009. HeKatE Project Homepage.
- [40] G. J. Nalepa. Business rules design and refinement using the XTT approach. In D. C. Wilson, G. C. J. Sutcliffe, and FLAIRS, editors, *FLAIRS-20 : Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference : Key West, Florida, May 7-9, 2007*, pages 536–541, Menlo Park, California, may 2007. Florida Artificial Intelligence Research Society, AAAI Press.
- [41] G. J. Nalepa. XTT rules design and implementation with object-oriented methods. In H. C. Lane and H. W. Guesgen, editors, *FLAIRS-22: Proceedings of the twenty-second international Florida Artificial Intelligence Research Society conference: 19–21 May 2009, Sanibel Island, Florida, USA*, 2009.
- [42] G. J. Nalepa, S. Bobek, M. Gawędzki, and A. Ligeza. HeaRT Hybrid XTT2 rule engine design and implementation. Technical Report CSLTR 4/2009, AGH University of Science and Technology, 2009.
- [43] G. J. Nalepa and K. Kluza. UML representation proposal for XTT rule design method. In G. J. Nalepa and J. Baumeister, editors, *4th Workshop on Knowledge Engineering and Software Engineering (KESE2008) at the 32nd German conference on Artificial Intelligence: September 23, 2008, Kaiserslautern, Germany*, pages 31–42, Kaiserslautern, Germany, 2008.
- [44] G. J. Nalepa and A. Ligeza. A graphical tabular model for rule-based logic programming and verification. *Systems Science*, 31(2):89–95, 2005.
- [45] G. J. Nalepa and A. Ligeza. *Software engineering : evolution and emerging technologies*, volume 130 of *Frontiers in Artificial Intelligence and Applications*, chapter Conceptual modelling and automated implementation of rule-based systems, pages 330–340. IOS Press, Amsterdam, 2005.
- [46] G. J. Nalepa and A. Ligeza. A visual edition tool for design and verification of knowledge in rule-based systems. *Systems Science*, 31(3):103–109, 2005.
- [47] G. J. Nalepa, A. Ligeza, K. Kaczor, and W. T. Furmańska. HeKatE rule runtime and design framework. In G. W. Adrian Giurca, Grzegorz J. Nalepa, editor, *Proceedings of the 3rd East European Workshop on Rule-Based Applications (RuleApps 2009) Cottbus, Germany, September 21, 2009*, pages 21–30, Cottbus, Germany, 2009.
- [48] G. J. Nalepa and I. Wojnicki. Filling the semantic gaps in systems engineering. In P. Scharff, editor, *52. IWK : Internationales Wissenschaftliches Kolloquium = International Scientific Colloquium : computer science meets automation : 10–13 September 2007 : proceedings*, volume 1, pages 107–112, Ilmenau : TU Ilmenau. Universitätsbibliothek, 2007. Technische Universität Ilmenau. Faculty of Science and Automation.
- [49] G. J. Nalepa and I. Wojnicki. A proposal of hybrid knowledge engineering and refinement approach. In D. C. Wilson, G. C. J. Sutcliffe, and FLAIRS, editors, *FLAIRS-20 : Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference : Key West, Florida, May 7-9, 2007*, pages 542–547, Menlo Park, California, may 2007. Florida Artificial Intelligence Research Society, AAAI Press.
- [50] G. J. Nalepa and I. Wojnicki. Using UML for knowledge engineering – a critical overview. In J. Baumeister and D. Seipel, editors, *3rd Workshop on Knowledge Engineering and Software*

*Engineering (KESE 2007) at the 30th annual German conference on Artificial intelligence : [September 10, 2007, Osnabrück, Germany]*, pages 37–46, september 2007.

- [51] G. J. Nalepa and I. Wojnicki. Towards formalization of ARD+ conceptual design and refinement method. In D. C. Wilson and H. C. Lane, editors, *FLAIRS-21: Proceedings of the twenty-first international Florida Artificial Intelligence Research Society conference: 15–17 may 2008, Coconut Grove, Florida, USA*, pages 353–358, Menlo Park, California, 2008. AAAI Press.
- [52] M. Negnevitsky. *Artificial Intelligence. A Guide to Intelligent Systems*. Addison-Wesley, Harlow, England; London; New York, 2002. ISBN 0-201-71159-1.
- [53] L. Nemuraite, L. Ceponiene, and G. Vedricka. Representation of business rules in UML and OCL models for developing information systems. *The Practice of Enterprise Modeling*, 15:182–196, 2008.
- [54] OMG. MOF 2.0/XMI mapping version 2.1. specification. Technical Report formal/2005-09-01, Object Management Group, September 2005. <http://www.omg.org/cgi-bin/doc?formal/2005-09-01.pdf>.
- [55] OMG. Meta object facility (MOF) version 2.0, core specification. Technical Report formal/2006-01-01, Object Management Group, January 2006. <http://www.omg.org/cgi-bin/doc?formal/2006-01-01.pdf>.
- [56] OMG. UML 2.0 diagram interchange version 1.0, specification. Technical Report formal/06-04-04, Object Management Group, April 2006. <http://www.omg.org/cgi-bin/doc?formal/2006-04-04.pdf>.
- [57] OMG. Production rule representation (OMG PRR) version beta 1. specification. Technical Report dtc/07-11-04.pdf, Object Management Group, November 2007. <http://www.omg.org/cgi-bin/doc?dtc/07-11-04.pdf>.
- [58] OMG. Unified modeling language version 2.1.2. infrastructure. specification. Technical Report formal/2007-11-04, Object Management Group, November 2007. <http://www.omg.org/cgi-bin/doc?formal/2007-11-04.pdf>.
- [59] OMG. Unified modeling language (OMG UML) version 2.2. superstructure. Technical Report formal/2009-02-02, Object Management Group, February 2009. <http://www.omg.org/cgi-bin/doc?formal/2009-02-02.pdf>.
- [60] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell*. O'Reilly, 2005.
- [61] J. L. Rash, M. G. Hinchey, C. A. Rouff, D. Gracanin, and J. Erickson. A tool for requirements-based programming. In *Integrated Design and Process Technology, IDPT-2005*. Society for Design and Process Science, 2005.
- [62] J. Robin. The object constraint language (OCL). <http://www.cin.ufpe.br/if710/2007/slides/OCL.ppt>, 2007.
- [63] R. G. Ross. *Principles of the Business Rule Approach*. Addison-Wesley Professional, 1 edition, 2003.
- [64] I. Sommerville. *Software Engineering*. International Computer Science. Pearson Education Limited, 7th edition, 2004.
- [65] W. M. Turski. *Informatics: A Propaedeutic View*. Elsevier Science Ltd, 2000.
- [66] F. van Harmelen, V. Lifschitz, and B. Porter, editors. *Handbook of Knowledge Representation*. Elsevier Science, 2007.