ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/308597928

# xtUML model complexity metrics and the influence of their distribution on model understandability

**Thesis** · July 2016

DOI: 10.13140/RG.2.2.35599.18088

CITATIONS
0

READS
49

**1 author:**



Nenad Ukic
Ericsson Nikola Tesla d.d. Croatia
**8** PUBLICATIONS **10** CITATIONS

SEE PROFILE

All content following this page was uploaded by Nenad Ukic on 25 September 2016.

The user has requested enhancement of the downloaded file.

U N I V E R S I T Y O F S P L I T
FACULTY OF ELECTRICAL ENGINEERING, MECHANICAL ENGINEERING
AND NAVAL ARCHITECTURE

S V E U Č I L I Š T E U S P L I T U
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

**Nenad Ukić**

# *xtUML model complexity metrics and the influence of their distribution on model understandability*

# *Mjere složenosti xtUML modela i utjecaj njihove distribucije na razumljivost*

DOCTORAL THESIS
DOKTORSKA DISERTACIJA

Split, 2016.

The research reported in this thesis was carried out at Department for Modelling and Intelligent Systems, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split.

Supervisor: Doc. dr. sc. Ljiljana Šerić, FESB, University of Split, Croatia
Disertation number: 126

Doktorska disertacija je izrađena na katedri za modeliranje i inteligentne računalne sustave, Fakulteta elektrotehnike, strojarstva i brodogradnje Sveučilišta u Splitu

Mentor: Doc. dr. sc. Ljiljana Šerić, FESB, Sveučilište u Splitu, Hrvatska
Rad broj: 126

There are 74 figures, 27 tables, 42 equations and 62 references in this doctoral thesis.

Committee for assessment of doctoral dissertation:

1. Prof. dr. sc. Maja Štula, FESB/Split
2. Doc. dr. sc. Ljiljana Šerić, FESB/Split
3. Prof. dr. sc. Vjeran Strahonja, FOI/Varaždin
4. Doc. dr. sc. Branko Žitko, PMF/Split
5. Doc. dr. sc. Toni Jakovčević, FESB/Split

Committee for defence of doctoral dissertation:

1. Prof. dr. sc. Maja Štula, FESB/Split
2. Doc. dr. sc. Ljiljana Šerić, FESB/Split
3. Prof. dr. sc. Vjeran Strahonja, FOI/Varaždin
4. Doc. dr. sc. Branko Žitko, PMF/Split
5. Doc. dr. sc. Toni Jakovčević, FESB/Split

Dissertation defended on: July, 22$^{nd}$ 2016.

Povjerenstvo za ocjenu doktorske disertacije:

1. Prof. dr. sc. Maja Štula, FESB/Split
2. Doc. dr. sc. Ljiljana Šerić, FESB/Split
3. Prof. dr. sc. Vjeran Strahonja, FOI/Varaždin
4. Doc. dr. sc. Branko Žitko, PMF/Split
5. Doc. dr. sc. Toni Jakovčević, FESB/Split

Povjerenstvo za obranu doktorske disertacije:

1. Prof. dr. sc. Maja Štula, FESB/Split
2. Doc. dr. sc. Ljiljana Šerić, FESB/Split
3. Prof. dr. sc. Vjeran Strahonja, FOI/Varaždin
4. Doc. dr. sc. Branko Žitko, PMF/Split
5. Doc. dr. sc. Toni Jakovčević, FESB/Split

Disertacija obranjena dana: 22.srpnja.2016.

## Abstract

In this thesis, we investigate the influence of distribution of model complexity on the understandability of *Executable Translatable UML*(xtUML) models. We adapted several metrics traditionally used for measuring software complexity to different xtUML sub-models and presented two different dimensions of measuring complexity distribution: *horizontally*, among elements of the same abstraction level, and *vertically*, among different abstraction levels. In order to test our hypothesis that complexity distribution influences the understandability of xtUML models, we have performed an experiment with student participants in which we have evaluated the understandability of three semantically equivalent xtUML models with different complexity distributions. Results indicate that a more uniform distribution of complexity has a positive influence on model understandability.

**Keywords:** executable software model, xtUML, complexity, metric, distribution, understandability

## Sažetak

U ovoj diseratciji istražili smo utjecaj raspodjele složenosti *xtUML izvršnih softverskih* modela na njihovu razumljivost. Prvo smo definirali mjere složenost xtUML modela na način da smo prilagodili nekoliko postojećih softverskih mjera složenosti različitim xtUML podmodelima. Istaknuli smo dva različita načina mjerenja raspodjele složenosti: *horizontalno*, među elementima iste apstrakcijske razine, i *vertikalno*, među različitim slojevima apstrakcije. Kako bi se testirati hipotezu da raspodjela složenosti utječe na razumljivosti xtUML modela, izvršili smo eksperiment sa studentima kao sudionicima u kojima smo ocijenili razumljivosti triju funkcionalno istovjetnih xtUML modela s različitim raspodjelama složenosti. Rezultati pokazuju da ravnomjernija raspodjela složenosti ima pozitivan utjecaj na razumljivost modela.

**Ključne riječi:** izvršni softverski model, xtUML, složenost, mjera, raspodjela, razumljivost

Mojoj majci, za trud, posvećenost i energiju koju si uložila za moj odgoj. Tek kad vidim Anteja naslućujem razmjere tog truda.

Mom ocu, na mirnoću, humor i vodstvo. Zbog tebe sam ovo što jesam.

Mojoj supruzi, za podršku koji si mi pružila i sve one večeri koje sam propustio.

Teti Mariji, za onaj vatrogasni kamiončić što sam ga razbio. Tada se upalio plamen znatiželje koji još uvijek gori u meni.

# Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Ljiljana Šerić for all support during my doctoral research. Our weekly meetings helped me keep the momentum and they are, I believe, the main reason this thesis has been completed. Her guidance helped me in all the time of research and writing of this thesis.

I especially want to thank Dr. Josip Maras, whose technical guidance and support has had a profound impact on this dissertation. His support during the research study, especially during the experiment, was invaluable for my work.

Besides them, I would like to thank the rest of my thesis committee: Prof. Maja Štula, Prof. Vjeran Strahonja, Dr. Branko Žitko and Dr. Toni Jakovčević for their insightful comments and encouragement.

I also wish to thank all my colleagues during 5 exciting years of Model Driven Workflow project. I am grateful and honoured to have had the opportunity to work with you. Special thanks goes to Mr. John R. Wolfe who *infected* me with xtUML and Mr. Cortland Starrett for the inspiration about the hypothesis.

Last but not the least, I would like to thank my family: my wife, my parents and to my brother and sister for supporting me throughout the work on this thesis and my life in general.

# Contents


Abstract . . . . . iv
Sažetak . . . . . v
List of Tables . . . . . xiv
List of Figures . . . . . xviii
List of Acronyms . . . . . xix

**1 Introduction** **1**

**2 Background and related work** **3**
2.1 xtUML . . . . . 3
2.1.1 Components . . . . . 3
2.1.2 Classes . . . . . 5
2.1.3 State machines . . . . . 7
2.1.4 Processing code . . . . . 9
2.2 Other executable software methodologies . . . . . 13
2.2.1 Real-time Object-Oriented Modeling (ROOM) methodology . . . . 14
2.2.2 Foundational Subset for Executable UML Models (fUML) . . . . . 14
2.2.3 Action Language for fUML (ALF) . . . . . 14
2.3 Software understandability . . . . . 15
2.3.1 Software understandability factors . . . . . 15
2.3.2 Software understanding process and theories . . . . . 16
2.3.3 A relation between software metrics and understandability . . . . . 17
2.4 Cyclomatic complexity . . . . . 18
2.4.1 Cyclomatic complexity and modularization . . . . . 20
2.4.2 Cyclomatic complexity for modules with multiple entry and/or exit nodes . . . . . 22
2.4.3 A critique of cyclomatic complexity as a software metric . . . . . 23
2.5 Entropy-based complexity metrics . . . . . 25
2.6 Data and information flow complexity metrics . . . . . 26

**3 Measuring complexity of xtUML models** **29**
3.1 Cyclomatic complexity of xtUML models . . . . . 29
3.1.1 Cyclomatic complexity of components . . . . . 29

3.1.2 Cyclomatic complexity of classes . . . 32
3.1.3 Cyclomatic complexity of state machines . . . 35
3.1.4 Cyclomatic complexity of processing code . . . 40
3.1.5 Calculating the overall cyclomatic complexity . . . 45
3.1.6 Calculating the distribution of cyclomatic complexity . . . 47
3.2 Entropy as a measure of xtUML component model complexity . . . 49
3.2.1 Model elements . . . 50
3.2.2 Vertical distribution of entropy . . . 51
3.2.3 Horizontal distribution of entropy across classes . . . 52
3.2.4 Horizontal distribution of entropy across bodies . . . 53
3.2.5 Entropy as a complexity metric: conclusion . . . 53
3.3 Data complexity of xtUML models . . . 54
3.3.1 Introduction to data types in xtUML . . . 55
3.3.2 Data type complexity . . . 55
3.3.3 Data flow complexity . . . 56

**4 Calculating procedure of xtUML complexity metrics 61**
4.1 BridgePoint translation process . . . 61
4.2 Implementation of xtUML cyclomatic complexity . . . 63
4.2.1 Vertical distribution of cyclomatic complexity . . . 64
4.2.2 Horizontal distribution of cyclomatic complexity . . . 73
4.3 Implementation of entropy complexity metric . . . 76
4.3.1 Vertical distribution of entropy complexity metric . . . 77
4.3.2 Horizontal distribution of entropy complexity . . . 78
4.4 Implementation of data complexity metric . . . 80
4.4.1 Distribution of data type complexity metric . . . 81
4.4.2 Vertical distribution of data flow complexity metric . . . 85
4.4.3 Horizontal distribution of data flow complexity metric . . . 89

**5 Hypothesis and experiment setup 91**
5.1 Study objects . . . 91
5.1.1 Comparing the naming conventions used by models . . . 92
5.1.2 Comparing the models in terms of LOC . . . 93
5.1.3 Comparing the models in terms of cyclomatic complexity . . . 95
5.1.4 Comparing model entropies . . . 97
5.1.5 Comparing the data type complexity of models . . . 102
5.1.6 Comparing the data flow complexity of models . . . 105
5.1.7 Conclusion on model comparison . . . 108
5.2 Study subjects . . . 109
5.2.1 Preparation . . . 109

5.2.2 Data collection . . . . . 111

**6 Experiment results** **113**

6.1 The relation between experiment results and complexity distribution . . . . 114

6.2 Threats to validity . . . . . 115

6.2.1 Internal validity . . . . . 115

6.2.2 External validity . . . . . 116

6.2.3 Construct validity . . . . . 117

**7 Conclusion** **119**

**Bibliography** **121**

# List of Tables


2.1 *Different flag combinations and associated meaning.* . . . . . 5

3.1 *Summary of associative relation effect to cyclomatic complexity* . . . . . 36

5.1 *Number of common words in the languages of three models and the language of the specification* . . . . . 92

5.2 *Horizontal distribution of lines-of-code (LOC) across bodies* . . . . . 94

5.3 *Horizontal distribution of lines-of-code (LOC) across classes* . . . . . 94

5.4 *Horizontal distribution of cyclomatic complexity across bodies* . . . . . 95

5.5 *Horizontal distribution of cyclomatic complexities across classes* . . . . . 96

5.6 *Vertical distribution of cyclomatic complexity.* . . . . . 98

5.7 *Vertical distribution of entropy and model element probabilities for the first model* . . . . . 98

5.8 *Vertical distribution of entropy and model element probabilities for the second model* . . . . . 99

5.9 *Vertical distribution of entropy and model element probabilities for the third model* . . . . . 99

5.10 *Horizontal distribution of entropy and model element probabilities per classes (first model)* . . . . . 99

5.11 *Horizontal distribution of entropy and model element probabilities per classes (second model)* . . . . . 99

5.12 *Horizontal distribution of entropy and model element probabilities per classes (third model)* . . . . . 100

5.13 *Horizontal distribution of entropy and model element probabilities per bodies (first model)* . . . . . 101

5.14 *Horizontal distribution of entropy and model element probabilities per bodies (second model)* . . . . . 101

5.15 *Horizontal distribution of entropy and model element probabilities per bodies (third model)* . . . . . 101

*5.16 Data type complexity for second model* . . . . . 102

*5.17 Data type complexity for the third model* . . . . . 103

*5.18 Vertical distribution of data flow complexity for all three models* . . . . . 106

*5.19 Horizontal distribution of data flow complexity across classes for the second model.* . . . . . 106

*5.20 Horizontal distribution of data flow complexity across classes for the third model.* . . . . . 106

*5.21 Horizontal distribution of data flow complexity across bodies for all three models* . . . . . 107

*5.22 Distribution of student ability across groups with results of ANOVA analysis* 111

*6.1 Summary of experiment results with ANOVA single factor analysis* . . . . . 113

*6.2 Cohen's* d *factor indicating the effect size of observed differences (S = small, M = medium and L = large effect size)* . . . . . 114

*6.3 Distribution of absence across the groups* . . . . . 116

# List of Figures


2.1 *An example of component model.* . . . . . 4

2.2 *xtUML component observed as class container.* . . . . . 4

2.3 *An example of associative relation.* . . . . . 6

2.4 *Two examples of a generalization relation.* . . . . . 7

2.5 *Graphical view of the state machine* . . . . . 8

2.6 *State-event matrix view of the state machine* . . . . . 9

2.7 *An example of OAL code.* . . . . . 10

2.8 *Another example of OAL code.* . . . . . 12

2.9 *Example of a control flow graph[1]* . . . . . 19

2.10 *Control flow graph as set of separate connected components[2]* . . . . . 20

2.11 *Control flow graph as single integrated connected graph obtained by splitting technique of Henderson-Sellers[2]* . . . . . 21

2.12 *Handling single entry, multiple exit (SEME) modules when calculating cyclomatic complexity.* . . . . . 23

3.1 *MESE CFG considered by Henderson-Sellers [2] and the one created from xtUML component.* . . . . . 30

3.2 *An example of two functionally equivalent interfaces at different level of abstraction.* . . . . . 32

3.3 *An example of a simple xtUML relation.* . . . . . 34

3.4 *An example of a generalization relation.* . . . . . 34

3.5 *An example of associative relation.* . . . . . 35

3.6 *A replacement class model for an associative relation.* . . . . . 36

3.7 *State machine of the* Shop *class.* . . . . . 37

3.8 *A pseudo-code description of the execution semantics of Shop instance state machines.* . . . . . 37

3.9 *A CFG created from the execution semantics of the Shop class instance state machine.* . . . 39

3.10 *McCabe's approach to cyclomatic complexity.* . . . 41

3.11 *Henderson-Sellers approach to cyclomatic complexity. Compare this graph with the one on figure 3.10* . . . 42

3.12 *Different approaches for handling multiple calls when calculating cyclomatic complexity.* . . . 44

3.13 *Handling of asynchronous communication and its effect to cyclomatic complexity.* . . . 45

3.14 *Horizontal and vertical complexity distribution.* . . . 48

3.15 *A part of Bridgepoint xtUML meta-model describing interfaces.* . . . 50

3.16 *Distribution of elements in xtUML models.* . . . 54

3.17 *An example of a state with multiple incoming transitions triggered by different events.* . . . 59

3.18 *Statements in OAL language that affect the number of definitions.* . . . 60

4.1 *The translation process used by the BridgePoint tool.* . . . 61

4.2 *An example of RSL template file for C++ language.* . . . 62

4.3 *An example of RSL queries.* . . . 63

4.4 *RSL code for calculating cyclomatic complexity of component ports.* . . . 64

4.5 *Counting the number of operations in classes.* . . . 64

4.6 *Part of xtUML meta-model describing components.* . . . 65

4.7 *Part of xtUML meta-model describing classes.* . . . 66

4.8 *Two ways to calculate cyclomatic complexity of state machine layer within a component.* . . . 67

4.9 *Cyclomatic complexity calculus for a single state machine.* . . . 68

4.10 *Part of xtUML meta-model describing state-machines.* . . . 69

4.11 *Part of xtUML meta-model describing bodies.* . . . 70

4.12 *Part of xtUML meta-model describing values (expressions).* . . . 71

4.13 *Calculating cyclomatic complexity of all bodies within a component.* . . . 72

4.14 *Calculating cyclomatic complexity of a single body.* . . . 73

*4.15 Generating CSV files needed for horizontal distribution across classes and bodies.* . . . . . 73

*4.16 Functions producing literal text during generation of CSV files.* . . . . . 74

*4.17 Calculating class contribution to overall cyclomatic complexity.* . . . . . 76

*4.18 Counting decisions and calls in all bodies within a class.* . . . . . 76

*4.19 The main template used by the first RSL script (used to generate the second RSL script).* . . . . . 77

*4.20 The second, generated, RSL script used on actual application models.* . . . 78

*4.21 A template with commented RSL lines used for partial generation of script for horizontal distribution of entropy.* . . . . . 79

*4.22 Partially generated script that relates model elements to the class it belongs.* 80

*4.23 Calculating distribution of data type complexity across classes.* . . . . . 81

*4.24 Recursive function for counting the number of primitive fields within a structured type.* . . . . . 82

*4.25 Part of xtUML meta-model describing data types.* . . . . . 83

*4.26 Part of xtUML meta-model describing relations.* . . . . . 84

*4.27 Calculating vertical distribution of data flow complexity.* . . . . . 85

*4.28 Calculating data flow complexity visualized by class and component model.* 86

*4.29 Calculating data flow complexity visualized by state machine model.* . . . . 87

*4.30 Calculating data flow complexity within bodies (total data flow complexity).* 89

*4.31 Calculating horizontal distribution of data flow complexity across classes.* . . 90

*5.1 Horizontal distribution of LOC across bodies.* . . . . . 93

*5.2 Horizontal distribution of LOC across classes.* . . . . . 94

*5.3 Horizontal distribution of cyclomatic complexity (decisions and calls) across bodies.* . . . . . 95

*5.4 Horizontal distribution of cyclomatic complexity (decisions and calls) across classes.* . . . . . 96

*5.5 Vertical distribution of cyclomatic complexity.* . . . . . 97

*5.6 Vertical distribution of model elements (used to calculate their probabilities).* 100

*5.7 Horizontal distribution of model elements across bodies (including empty ones).* . . . . . 102

*5.8 Horizontal distribution of data type complexity across classes.* . . . . . 103

*5.9 A class model used in the second model.* . . . . . 104

*5.10 A single class, data structure and an enumeration used for data modelling in the first model.* . . . . . 104

*5.11 Vertical distribution of data flow complexity for all three models* . . . . . 105

*5.12 Summary of horizontal distribution of data flow complexity across classes.* . . 107

*5.13 Horizontal distribution of data flow complexity across bodies.* . . . . . 108

# List of Acronyms

**UML** Unified Modelling Language
**xtUML** Executable and Tranlatable Unified Modelling Language
**OO** Object-Oriented
**RTC** Request-To-Completion
**OAL** Object Action Language
**ALF** Action Language For fUML
**fUML** Foundational Subset For Executable UML Models
**LSI** Latent Semantic Indexing
**SESE** Single-entry, single-exit
**SEME** Single-entry, multiple-exit
**MEME** Multiple-entry, multiple-exit
**MESE** Multiple-entry, single-exit
**LOC** Lines of code
**CC** Cyclomatic complexity
**CFG** Control flow graph
**AST** Abstract Syntax Tree
**MM** Meta-model
**def-use** Definition-usage
**TCA** Theory Correct Answers
**TP** Theory Points
**DCA** Domain Correct Answers
**DP** Domain Points
**CA** Correct Answers
**IA** Incorrect Answers
**P** Point
**TT** Total Time
**CAPM** Correct Answers Per Minute
**PPM** Points Per Minute

# 1 Introduction

Traditionally, software models are used in the initial stages of the software development process: for requirements analysis, documentation, or early design purposes. The first formal use of UML-based models in software development typically meant using class models to generate the initial source code skeleton, which is then used as a base for traditional software development. The problem with such, *elaborative* approach is that the software skeleton generated from the initial model typically changes over time which makes the model deprecated.

Executable software models are changing this elaborative paradigm by making the model a central development artifact. A key feature of executable models is their ability to be executed, which implies the possibility of testing. In order for a model to become executable, it must become semantically complete, that is, in addition to the structure, the model must specify the details of application behavior. The traditional iterative software development process of testing and correction can then be applied to models themselves. After a model is verified and meets all functional requirements, it is used to generate the application source code. In this *translational* process, the source code is considered as a temporary asset on the way from a model to binary form used in production. The intellectual property of such software development is not the source code, but a model of the application, as well as the translation mechanism used to generate the source code.

One of the oldest and most mature executable UML methodologies is *Executable Translatable UML* (xtUML), a successor to the Shlaer-Mellor object-oriented methodology [3], adapted to UML graphical notation. An open source tool – BridgePoint [4] supports the xtUML model development and is the main enabler of the methodology.

There are several important implications of the translational approach used by executable software models. First, it is possible to separate functional and non-functional requirements. The model is used to specify functional requirements in the simplest possible way, while decisions about platform, language, robustness, and speed are handled in the code generator. In this way, the software model becomes independent of the platform on which the application will be executed and of the target language of the generated source code. Besides obvious reuse benefits, the separation of functional and non-functional requirements means that models can become more abstract and therefore simpler and more approachable to domain experts.

In general, cognitive simplicity and understandability of software is the result of clear abstractions, i.e. clear links between formal concepts used to formalize the software and actual domain concepts they represent [5]. Since functional and non-functional requirements can be separated, executable software models no longer have to be a compromise between simplicity and application implementation constraints. The focus of executable software models can be entirely moved to the clarity of application's functional specifications. This means that the understandability of a model becomes essential. Because of this, objectively measuring model understandability is of paramount importance.

Approaches to measuring software understandability can be categorized into two groups: *i) linguistic*, which focus on the similarity between software element identifiers and domain concepts [5], and *ii) metric-based* which are concerned with the relationship between different software metrics and software understandability.

In this dissertation we use a *metric-based* approach to investigate the hypothesis that distribution of complexity influences the understandability of xtUML models. For this, as a complexity metric, we have used cyclomatic complexity, a metric based on the number of linearly independent execution paths through an application. Traditionally, the majority of software metrics are applied to source code, but in this case, since we are dealing with models, we adapted the existing software complexity measures to xtUML models. For each of those measures, we present a way to calculate the integrated complexity of a complete xtUML model and different ways to distribute it across the model. To validate our hypothesis, we conducted an experiment in which we evaluated the understandability of three semantically equivalent models with different complexity distributions. Understandability of models was evaluated using questionnaires on three groups of 20 students. Experiment results confirmed our hypothesis and have indicated that complexity distribution influences model understandability.

This thesis is structured as follows: In section 2 we describe the xtUML model and the four sub-models comprising the model. This is followed by an explanation of factors influencing software understandability, along with a description of each of the complexity metrics that we analysed. In section 3 we present adaptation of cyclomatic complexity metric to each of the xtUML sub-models, and a way to calculate cyclomatic complexity of complete xtUML model. We finish the chapter by describing different ways to calculate distribution of cyclomatic complexity. Similar pattern is followed in following 2 chapters: section 4 presents entropy as measure of xtUML model complexity while section 5 describes data type and data flow complexity of xtUML models. In section 6, we describe the hypothesis and setup of our experiment. We compare the models in terms of metrics and describe how experiment subject were prepared and experiment data collected. Experiment results are presented and interpreted in section 7. Finally, the thesis is finished with a conclusion.

# 2 Background and related work

In this section, we describe the xtUML model and the four sub-models comprising the model. This is followed by an explanation of factors affecting software understandability, along with a description of cyclomatic complexity.

## 2.1 xtUML

The xtUML (eXecutable and Translatable Unified Modelling Language) is the successor to the Shlaer-Mellor method [3] [6], an object-oriented software development methodology introduced by Sally Shlaer and Stephen Mellor in 1988. It is a software development language that uses the graphical notation of the standard UML, but also defines precise execution semantics and timing rules. xtUML models are computationally (Turing) complete, and all information about software structure, behaviour, and processing is integrated in such a way that xtUML models can be executed. Although intended as a general-purpose language for executable software modelling, xtUML is probably best suited for embedded applications[7]. In addition, as a graphical software modelling methodology, it may not be a best choice for highly algorithmic applications.

A system designed with xtUML is composed out of four interconnected types of models: *i) component* models, which define the overall system architecture; *ii) class* models, which define concepts and relations within a component; *iii) state machine* models, which define class instance life-cycle; and the *iv) processing* model which specifies execution details. The component, class, and state machine models are graphical models, while the processing model is textual.

### 2.1.1 Components

The foundational building block of xtUML models are components. Each component is considered as a black-box that uses only interfaces to communicate with other components. An xtUML interface is a definition of a message set that can be used for inter-component communication. Interfaces are bidirectional and messages on a single interface can go in both directions.

Interfaces specify the types of component ports across which the actual communication is performed. Two components may only be connected across ports that are typed by the same

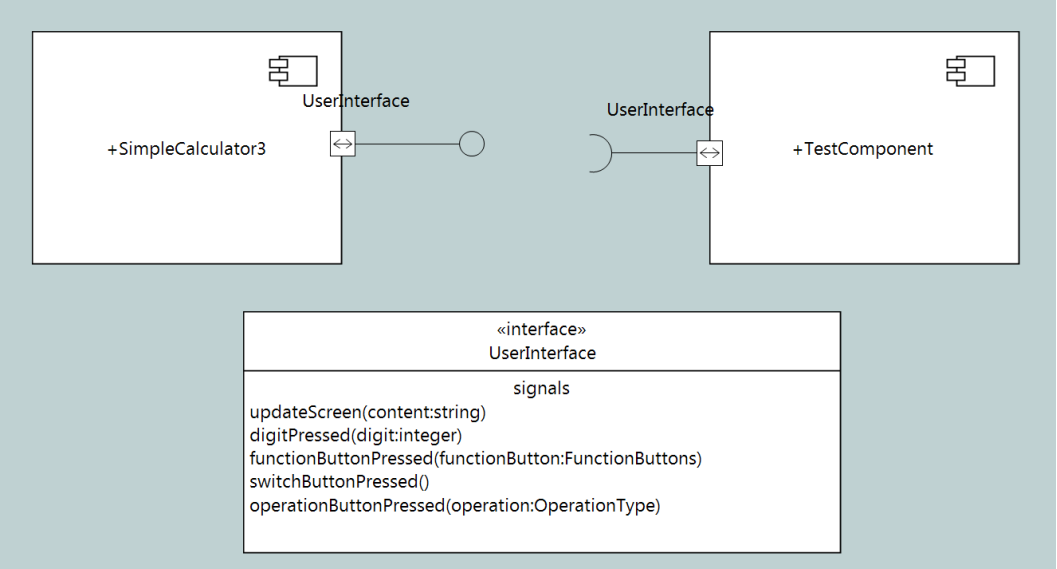



*Figure 2.1. An example of component model.*

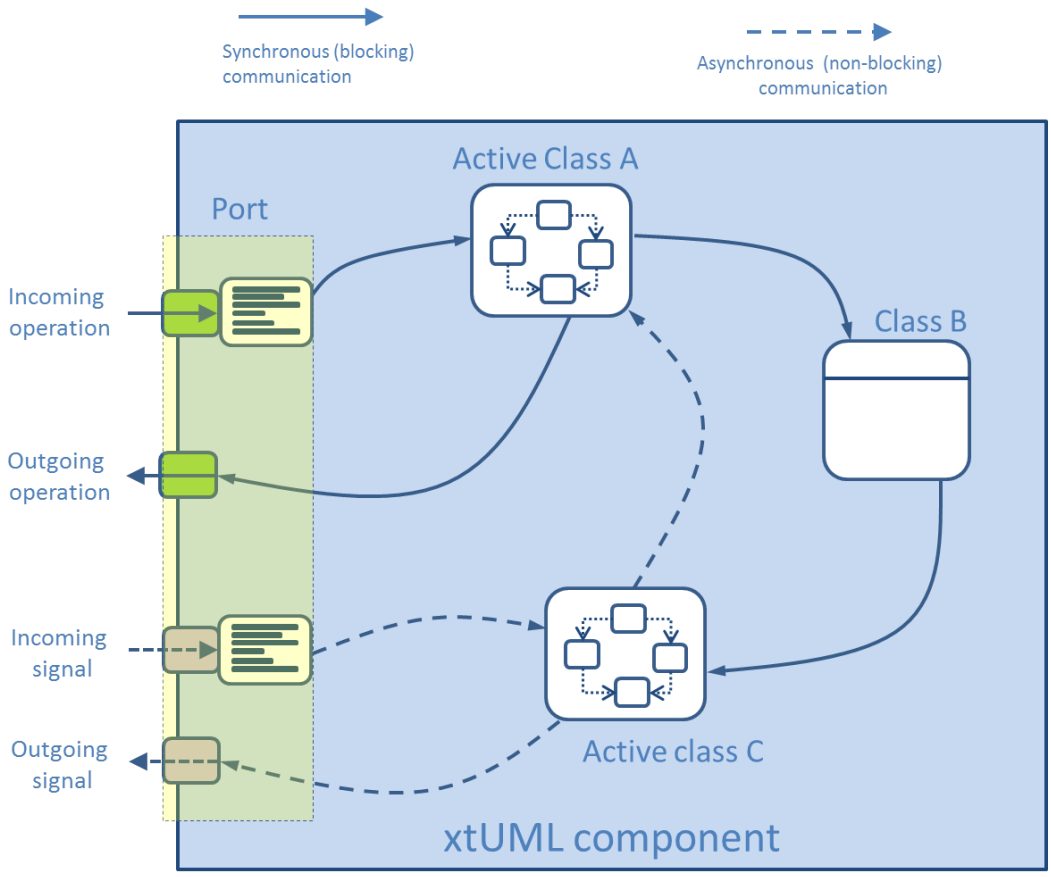



*Figure 2.2. xtUML component observed as class container.*

interface, and where one component should provide the interface, while the other should require it. While interfaces are merely a specification of possible messages described with names, parameters, and directions relative to the "provider side", ports can actually contain action code that specifies actions to be taken when a certain message is received. It is important to emphasize that only incoming messages can be associated with action code.

The communication between two components can be either *synchronous* or *asynchronous*. Synchronous inter-component communication is enabled with interface operations, while asynchronous communication is achieved through interface signals. Figure 2.1 shows an example of an xtUML component model. Component *SimpleCalculator3* provides its service through *UserInterface* (lollipop symbol), while *TestComponent* requires the same interface (cup symbol). The interface consist of five signals from which at least one has a non-default direction, *from provider* (in our case, from *SimpleCalculator3* component towards *TestComponent*). This can be inferred from bi-directional arrows on component ports which indicate that port messages go in both directions.

The main purpose of components is to serve as containers for classes and their instances. At design time, the visibility of classes is limited to the component in which the class is defined. Similarly, at runtime, the visibility of class instances is limited to the enclosing component. Since components can be considered as singleton (static) elements, the action code on incoming port messages usually creates a new or searches for an existing class instance to which to proxy the original port message (see figure 2.2).

Notice however that, on a component diagram, no classes, state machines or action code is shown. When showing component diagrams, we are only interested in components of our system, interfaces between them and data (types) exchanged across them.

### 2.1.2 Classes

An xtUML class model describes domain concepts and the relationships between instances of those concepts. Similarly as in other object-oriented languages, a simple relation in xtUML is defined by two relation ends, each of them defined with multiplicity and conditionality flags, as well as with a phrase describing its semantics. Different combinations of multiplicity and conditionality flags specify different runtime limitations that apply to the number of instances on a relation end (see table 2.1)

*Table 2.1. Different flag combinations and associated meaning.*

| **Conditionality flag** | **Multiplicity flag** | **Symbol** | **Instance number limitation** |
|---|---|---|---|
| TRUE | FALSE | 0..1 | at most one |
| FALSE | FALSE | 1 | exactly one |
| TRUE | TRUE | * | zero or more |
| FALSE | TRUE | 1..* | at least one |

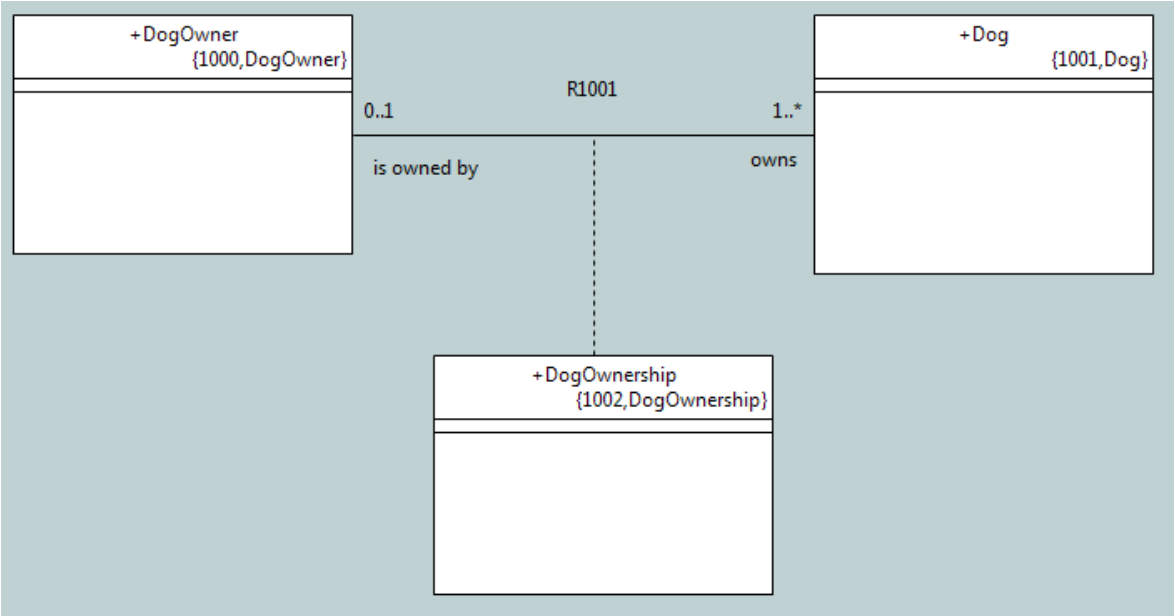



*Figure 2.3. An example of associative relation.*

Notice that conditionality and multiplicity flags are properties of a relation end, not the relation itself. This implies that the same relation may expose different runtime limitations depending on the *direction* that we traverse it.

Associative relations are used in cases where there is need to model details of a simple relation. Typically this includes cases when a relation has some attributes or a life-cycle of its own. In that case, a relation is represented as an associative class which can have all typical class features: a state machine, attributes, operations, or even relations towards other classes. Since an associative class is both a relation and a class, the mere existence of an associative class instance implies the existence of a pair of instances of *related classes* (non-associative classes involved in associative relation) and a link between them. On figure 2.3, if an instance of *DogOwnership* class exists, it is related to exactly one instance of *Dog* and exactly one instance of *DogOwner*. In most other aspects, the associative relations are very similar to simple relations.

Unlike some other object-oriented (OO) languages such as C++ or Java, a generalization relation in xtUML assumes the existence of two separate instances which should be created independently and related explicitly by the user, the same way as it is done with simple relations. In addition, an xtUML subclass does not inherit superclass operations and attributes. In the majority of OO languages, generalization is mostly used as an extension mechanism, but this is not the case with xtUML. The main purpose of such relation is to split an instance population into *complete* and *disjoint* sets of subclass instances. *Complete* sets means that there may be no other subclasses other than those stated initially, i.e. we cannot easily add new subclasses to an existing generalization relation; we can only add a new generalization relation and specify its complete and disjoint subclasses. *Disjoint* subclass instance sets imply that a single superclass instance is only related to a single subclass instance per

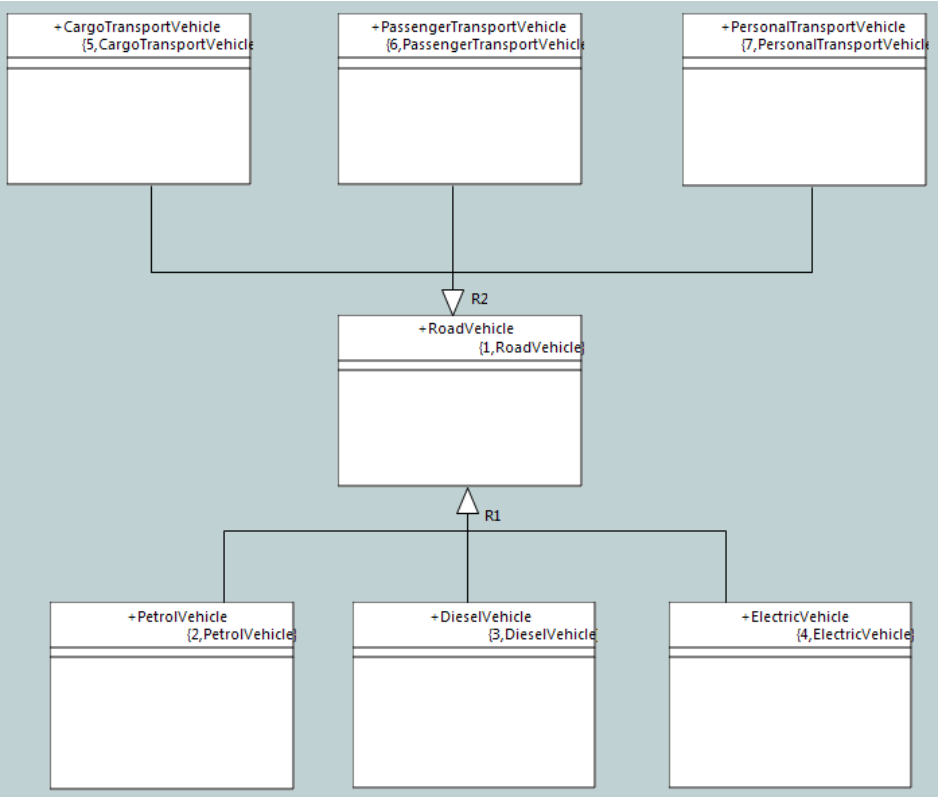



*Figure 2.4. Two examples of a generalization relation.*

generalization relation; a superclass is related to as many subclass instances as the number of generalization relations it defines.

On figure 2.4, each generalization relation defines a single generalization set that specializes the *RoadVehicle* class by some distinct property. Notice the orthogonality between those generalization sets: relation R1 differentiates the road vehicles by the fuel they use, while the R2 differentiates the road vehicles by their usage. At the same time, an instance of *RoadVehicle* is related to one instance across relation R1 and one across relation R2. This semantics is very different from traditional OO semantics, but it does not assume anything about the target language. Generalization relations in xtUML do not rely on inheritance mechanism of traditional OO generalization and may be applied easily to non-OO languages as well. Considering the translational nature of the xtUML language, this is very important feature, because it does not put a constraint on a target language.

### 2.1.3 State machines

State machine models in xtUML can only be defined within a context of a class. Typically, a state machine is used to visualize the life cycle of a class instance, and is usually composed out of one or more numbered states. Figure 2.5 shows an example state machine model.

At a certain point in time, an instance can only be in one state, and it can move from one state to another only if there exists an explicit transition between those states. A transition

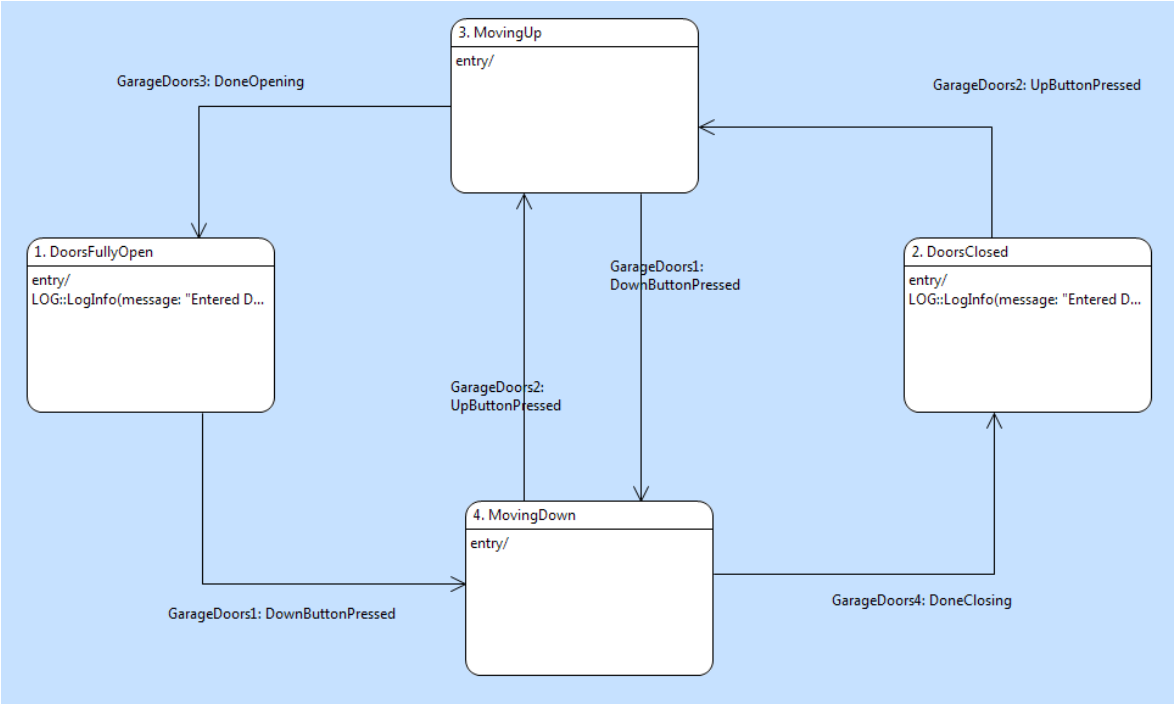



*Figure 2.5. Graphical view of the state machine*

is triggered by a certain event and cannot be additionally guarded with extra conditions. In case where a transition does not have an associated event, the transition cannot occur.

With each transition we can associate some action code which will be executed when the instance moves from one state to another, following the transition. In addition, action code can also be associated with state entries, which means that we can execute additional action code, just after the execution of the transition code. On the other hand, action code cannot be associated with state exits nor with the fact that an instance resides in some state.

When a class instance, that has an associated state machine, is created, it immediately starts in the *initial* state, the lowest numbered state. Even though action code can be associated with state transition and state entries, in this case, even if the initial state has associated entry code, no code is executed since the instance is immediately placed in a state, without transitioning to it.

After being created, an instance waits for events that will move it to different states. If no events occur, the instance will remain in the initial state until deleted. If an event occurs, the corresponding transition action code (also called transition effect behaviour) is executed. This is followed by target state entry behaviour. Note that those two behaviours are always executed in a sequence, one after another, as a single *Request-To-Completion* (RTC) step, without any interruptions. After an event is processed, the instance resides in a new state and awaits for further events. This is repeated until the instance reaches the end of its life-cycle, which happens in the following situations:

- The instance reaches its final state. After executing the final-state entry behavior, the

instance is deleted.

- The instance receives an event that it does not know how to handle; a runtime error happens. The current execution is stopped and all available information is logged for troubleshooting.
- The instance is explicitly deleted, if some other code request instance deletion.

Figure 2.5 shows a graphical view of a garage door state machine. Upon creation, an instance is placed in the *DoorFullyOpen* state. In this state, a garage door only expects a *DownButtonPressed* event which triggers transition to the *MovingDown* state. From that state, after an instance of garage doors receives a *DoneClosing* event, the transition to the *DoorsClosed* state will happen. Alternatively, if an *UpButtonPressed* event happens before, the door will start moving in reverse direction and the state machine will end up in the *MovingUp* state.

An instance can process a certain event if its current state has exactly one outgoing transition that is triggered by that event. For other cases, the developer should specify a *State-Event matrix* (figure 2.6) which defines what action should be performed for any state, event combination. Unlike the graphical view (figure 2.5) of the state machine, the state-event matrix represents a complete state machine with all possible combinations of states and events.

| | GD1: DownButtonPressed | GD2: UpButtonPressed | GD3: DoneOpening | GD4: DoneClosing |
|---|---|---|---|---|
| DoorsFullyOpen | MovingDown | Event Ignored | Event Ignored | Can't Happen |
| DoorsClosed | Event Ignored | MovingUp | Can't Happen | Event Ignored |
| MovingUp | MovingDown | Event Ignored | DoorsFullyOpen | Can't Happen |
| MovingDown | Event Ignored | MovingUp | Can't Happen | DoorsClosed |

*Figure 2.6. State-event matrix view of the state machine*

Notice that for handling cases when a state is not ready to receive an event, the modeller has two options: to simply ignore the event (*Event Ignored*, figure 2.6), or to trigger an error handling procedure (*Can't Happen*, figure 2.6). When an event is ignored, nothing happens, an instance remains in its current state, ready to receive new events. On the other hand, when an error case occurs, the execution is stopped and the error details are logged. The act of deciding which action should be taken in case an unexpected event occurs is not automated, and should be a part of the modelling decisions done by the modeler.

### 2.1.4 Processing code

Processing code, written in Object Action Language (OAL), describes the behavioural (runtime) details of an xtUML model. Code execution starts at the first statement of the action and proceeds sequentially through the succeeding lines as directed by program's control

logic. The execution of an action terminates when the last statement has been executed. In xtUML, processing code can be placed within a *body*[1] of many different model elements:

```
//Creation and deletion of instance
create object instance some of SOME;
create object instance secondSome of SOME;
create object instance car of Car;
create object instance king of King;
delete object instance secondSome;

//(Un)relating instances
relate some to car across R3;
relate king to car across R4;
unrelate king from car across R4;

//Unary and binary operations
a=3*6;
b=a+58;
c=b/4;
d=b%10;
e=true;
f=(a==3);
e=not f;
g=e and f;
h=e or f;
```

*Figure 2.7. An example of OAL code.*

- **Incoming messages on ports:** When a component receives an incoming message (synchronous operation or asynchronous signal) on one of its ports, associated action code usually creates new or selects existing instance to which a message will be relayed. Therefore, processing code on incoming port message is typically very simple.
- **Class or instance operations:** Instance operations have the same purpose as in most other OO languages and are usually used to perform some processing on the instance. Class operations are very similar to *static* operations and are invoked by directly referencing a class, instead of a particular instance.
- **State machine transitions:** Triggered by an incoming event, a transition between states occurs, leading to the next state. Transition itself may define a processing (called *transition effect behaviour*) and is executed before the destination state is entered. This processing may be parametrized by the data items carried by the event that triggered the transition. Transitions processing are frequently used and typically invoke asynchronous communication towards other instances or component ports.

[1]The term *body* comes from an xtUML meta-model element used as container of OAL statements.

- **State machine states:** State (entry) processing is invoked each time a state is entered, just after the transition effect behaviour is executed. Similarly as the transition effect behaviour, state processing can be parametrized by the data wrapped in the event that triggered the transition. As transition effect behaviours, state entry behaviours are typically used to invoke asynchronous communication with other instances or ports. Note however, if there are more than one incoming transition (triggered by more than one different event), only a common subset of event data items present in all incoming events will be available in processing code. Because of this, state entry processing is not used as frequent as the transition effect processing.

- **Functions:** Functions are standalone behaviours that have no context. Unlike many other popular languages, in xtUML, there is no dedicated *main* function. Any function without parameters can be used as an entry to the application behaviour[2]. Such functions typically initialize the model by creating and initializing instances. After this (synchronous execution), functions usually fire the initial event or signal that triggers the chain of asynchronous communication within the model.

- **Derived attributes:** Derived attributes are the attributes that have their value calculated upon access, using the processing specified as the body of the attribute. They act as instance operations but are accessed using traditional attribute access syntax. Derived attribute value is usually calculated from other instance attributes and rarely invoke asynchronous communication.

- **Bridge bodies:** Logically grouped in external entities, bridges act like functions that can be invoked from the model, but which have their behaviour defined *outside* the model. Typically bridges provide platform independent wrapper for platform or language specific behaviour. Although their behaviour is not intended to be specified in the model, it can be, the same way it is done for functions. This is easy way to create a *mock-up* model for early testing purposes, and bridge OAL is mostly used for that purpose. Otherwise, bridge behaviour is specified in the language in which code will be generated. Note also that there are some built-in bridges that provide platform specific behaviour for common functionalities such as printing to console output or time related functions.

Although OAL language may seem as a (textual) language on its own, it is an integral part of xtUML model and has no sense outside the context of a model. Unlike most other programming languages today, the main purpose of OAL is to serve as an action language to specify processing details. It cannot be used to specify structural parts or a complete xtUML

[2]This is true only for proactive models. Reactive models, that only react to external stimuli, do not have to define starting function, because their behaviour is triggered *from outside the model* by incoming messages on the ports

model[3]. The OAL was made as a simple action language very similar to (english) natural language. OAL allows following type of statements:

```
//Selection (from instances)
select any otherSome from instances of SOME;
select many someSet from instances of SOME where selected.age>25;

//Selection (across relations)
select one car related by some->Car[R3];
select many kingSet related by some->Car[R3]->King[R4];

//for loop
for each king in kingSet
    //Do something
end for;

//If conditional branch
if(not_empty car)
    //Do something else
end if;

//While loop
while(a>0)
    //Iterate!
    a=a-1;
end while;

//Synchornous invocations
some.operation(); //instance operation
t = SOME::static_operation(aInt:1, aReal:3.14); //Class-based (static operation)
LOG::LogInfo(message:"This is bridge invocation!"); //bridge invocation
z = CALC::add(aInt:1, aReal:3.14); //port operation invocation
::funny(aInt:1, aReal:3.14); //Function invocation

//Event generation
generate SOME1:JustDoIt(what:"Jump", when:3.13, howManyTimes:100) to some;

//Signal sending
send CALC::addSig(aInt:1, aReal:3.14);
```

*Figure 2.8. Another example of OAL code.*

- **Object creation and deletion:** Similarly as in many other languages, objects in OAL can be created and deleted.
- **Object linking and unlinking:** Links between instances are created using *relate* and deleted using *unrelate* statements. As expected, it is only possible to relate instances whose classes are related through class diagrams.
- ***For* and *while* loops:** A *while* loop is used for general purpose looping, while a *for* loop is used only to iterate through a set of instances.

[3]This feature differs the OAL from standard UML modelling action language, ALF (Action Language for fUML), which is, at the *minimum syntactic conformance*, used as OAL, only for processing purposes. However, ALF also specifies the *extended conformance* which allows it to be used as standalone language and to specify complete fUML models, including the structural parts. For more information about ALF please refer to [8]

- **Conditional branching:** An *if* branch, similarly as in other programming languages, is used to branch from the execution control flow depending on the value of some boolean expression.
- **Standard unary and binary operations:** Standard arithmetic and logical operations such as adding, subtracting, multiplying, dividing, modulo operation, equality check, non-equality check, negation, logical *and*, logical *or* are supported.
- **Instance selection:** There are two different kinds of selections: selections from a set of all existing instances using some criteria and a selection across a relation starting from a single (or a set of) instances. In a general case, a result of a selection may be an empty set, a single instance or a set of instances. Because of this reason, selections are often followed by conditional branching and/or *for* loops.
- **Different synchronous invocations:** In OAL it is possible to synchronously invoke operations on an instance or a class, a bridge, a port operation or a function. If a synchronous invocation does not return any value it is standalone statement, otherwise it is an expression in some other (typically assignment) statement.
- **Event generation:** An event generation statement is used for asynchronous communication between class instances. This means that, after an event is generated, the sender side control flow continues execution without waiting for the event to be processed. Any OAL code following the event generation statement should not assume the event has been processed by the target instance.
- **Sending signals to ports:** Sending signals to ports is another way of asynchronous communication used when an instance needs to communicate with the outside of the component. Similarly to event generation, OAL code following the signal send statement should not depend on the fact that the signal has been processed.

Processing code is the only textual model in xtUML, because it is more efficient to specify processing instructions in a textual than in a visual way. Behind the scenes, processing models are represented with corresponding instances in the xtUML meta-model, similarly to component, class, or state machine models. From the perspective of execution and code generation tools, the processing code is handled similarly as other three models.

## 2.2 Other executable software methodologies

This section will briefly cover the other executable software model methodologies and standards.

### 2.2.1 Real-time Object-Oriented Modeling (ROOM) methodology

UML-RT (Unified Modeling Language - Real Time) is a UML profile defined by IBM for the design of distributed event-based applications. The profile is based on the Real-Time Object-Oriented Modeling (ROOM) methodology [9] and was implemented in the RSA-RTE [10] product, an extension of IBM's Rational Software Architect (RSA) product.

The basis of UML-RT is the notion of *capsules* that at the same time can have both, the internal structure (via capsule parts) and behaviour (via a state machine). Capsules can be nested, and can communicate synchronously and asynchronously via messages that are sent and received through ports. The types of messages that a port can transmit are defined by protocols. Unlike xtUML components, capsules can be created both statically at design time and dynamically at run-time.

RSA-RTE tool allows several languages to be used to specify actions (e.g. C, C++ and Java). Model execution is only possible by translation to source code, but no model-level interpreter is possible, at least not in the sense provided by the BridgePoint [4] tool.

### 2.2.2 Foundational Subset for Executable UML Models (fUML)

Realizing the importance of simplicity and clear semantics that are missing from UML standards, the OMG defined the semantics of a *foundational subset for executable UML models*(fUML) [11]. The main goal of this standard is to act as an intermediary language between *surface subsets* of UML used for modeling and computational platform languages used as the target for model execution. fUML is designed to be *compact* in order to facilitate the definition of clear semantics and the implementation of execution tools. In addition, it is supposed to be *easily translated* from common surface subsets of UML to fUML and from fUML to common computational platform languages. However, if the feature to be translated is excluded from fUML, the *surface-to-fUML* translator has to generate a *coordinated set of fUML elements* that has the same effect as that feature. Then the *fUML-to-platform* translator would need to recognize the pattern generated by the surface-to-fUML generator, in order to map this back into the desired feature of the target language. Compactness can therefore conflict with ease of translation [11].

One of the future directions of UML is executable modelling which assumes simplicity and clear execution semantics. The FUML standard is a huge step forward in this direction. However, the problem with fUML is weak tool support and it remains to be seen if it will be accepted by the community.

### 2.2.3 Action Language for fUML (ALF)

The Action Language for Foundational UML (or *ALF*) [8] is a textual surface representation for UML modeling elements. The execution semantics for ALF are given by mapping

the ALF concrete syntax to the abstract syntax of the fUML. A primary goal of an action language is to act as the surface notation for specifying executable behaviors within a wider model. ALF also provides an extended notation that may be used to represent structural modelling elements. Therefore, it is possible to represent an UML model entirely using ALF. However, ALF syntax only directly covers the limited subset of UML structural modelling available in the fUML subset.

ALF is important because it is an attempt to standardize the action language for executable software models based on UML. In addition to that, a possibility to completely specify an fUML model using textual notation also helps alleviate the problem of poor tool support for resolving version conflicts in graphical models [12]. Since ALF is actually a textual surface notation for fUML models, higher level languages can be translated directly to ALF (instead to fUML). In such form, higher level models will be able to use ALF virtual machines and translate to target source code using ALF translators.

## 2.3 Software understandability

Software understandability or program understanding includes activities needed to obtain a general knowledge of what a software product does and how the parts work together [13]. In this section, we describe the factors influencing the understandability and related work.

### 2.3.1 Software understandability factors

There are several factors influencing software understandability:

- *Functional size of software* is the main factor influencing understandability. It refers to the number and complexity of use cases that the software system satisfies, and not the implementation size [14] [15] [16]. Unless we are interested in the relation between functional size and the understandability of software, comparing the understandability of software applications of various sizes is problematic.

- *Programming language* used for implementing the software system. Different implementation languages have different readability which influences source code understandability.

- *Consistency and quality of naming conventions*. As indicated by Laitinen [5] and Rajlich [17], the similarity between concepts used for software elements identifiers and domain concepts used in application specification is one of the key factors influencing the understandability of the application. The strength of mapping between those two sets of concepts reflects the consistency and intuitiveness of naming and coding conventions in the source code.

- *Modularisation and choice of design alternative*. Different approaches to source code modularization [18] and the quality of the modularization [19] have an impact on the comprehension of the software source code.

Most empirical evaluations of software understandability are based on subjective evaluation made by experts in which understandability of a number of software applications of different sizes, domains, and authors is ranked [20]. The main problem with such evaluations is precision, because the factors affecting software comprehension are confounded. In order to precisely relate source code or model understandability with some of the mentioned factors, it is necessary to eliminate all other factors, and variate the remaining factor.

### 2.3.2 Software understanding process and theories

Traditionally, there are two classic approaches to program comprehension: *top-down* and *bottom-up*. In the top-down approach [21], a programmer makes a hypothesis and then rejects it or confirms it based on evidence found in the source code. The bottom-up approach [22], [23] is based on chunking, which represent parts of code that programmer recognizes. Code chunks have a name and a meaning and are often nested. Understanding source code refers to building larger and larger chunks from already existing, smaller ones.

The process of software comprehension usually starts by mapping domain concept names with source code identifiers relying on programmers intuition and experience. If this technique fails, a more formal, string pattern matching process is typically used. When these name-mapping techniques fail to locate the concepts, programmers typically instrument the code with various logs and execute different application features. This process of execution and trace-mapping is known as *dynamic search* or *software reconnaissance* [24]. In addition to dynamic analysis, static analysis can also be performed [25]. Starting from the program or test case main function, top down control and/or data flow tracing can be used to find the relevant part of the source code.

Rajlich et. al. [17] introduced the process of *concept location* which implies a mapping between domain concepts and their code implementation. The first and the most significant phase in the concept location process is based on the similarity between names in software specification documents and identifiers used in software implementation. In order to be able to measure this similarity, Laitinen [5] considered a *language* to be a set of symbols with associated meanings. In this sense, every person, article, or source code has a language of its own. Languages are said to be related if they share the same symbol and its meaning. Smaller and more closely related languages are easier to understand than larger and those distantly related languages. With such definition of a language, only a relative measure of understandability makes sense.

To demonstrate the process of concept location and its effect on the maintenance process, Patrenko *et. al.* [26] introduced an ontology that describes the software application domain.

The ontology is built from pre-existing knowledge about the domain and is extended as programmers acquire new knowledge about the application. During concept location process, the domain facilitates the querying process by presenting words, phrases, concepts and attributes from the ontology domain. This helps programmers to narrow down the search and to make the queries more precise. The benefits of this, ontology-based, concept location approach is also verified in a large-scale software project with more that 50000 methods.

Since these techniques are based on naming similarities, they are often poor at handling the problem of homonyms, synonyms and polysemy. An additional problem in maintenance is that the vocabulary used to describe the software evolves and it may significantly differ from the initial vocabulary used for implementation. In addition to that, some domain concepts are not represented explicitly in the source code and cannot be found with in this way.

There are also some more advanced approaches that are not based on the naming similarities. *Latent Semantic Indexing*(LSI) [27] [28] is based on the fact that words with similar meaning appear close in the documents. The meaning of words in LSI is derived from their usage rather than from a dictionary or thesaurus. LSI has been shown to address problems of polysemy and synonymy [27] quite well which makes it a good fit for source feature/concept search problem because developers usually construct queries without precise knowledge about target vocabulary [29].

### 2.3.3 A relation between software metrics and understandability

The understandability of a software artefact is a key factor in software maintainability. Problems with managing software projects and predicting maintenance efforts indicate a need for objective prediction of software maintainability. The common usage of software metrics is the creation of maintainability prediction models.

Welker [30] noticed a spiral of code degradation through maintenance and has proposed an integrated maintainability measure. The benefit of such integrated measure is the objective quantification of code degradation which could be used in software management decision support. Zhou [31] empirically investigated the relationship between 15 design metrics and the maintainability of 148 Java open source software. The results indicate that size and complexity are strong predictors of maintainability while cohesion and coupling metrics do not seem to have a significant impact on maintainability.

Nazir [32], proposed a regression model for estimating the understandability of OO software using the number of attributes, the number of associations, and the maximum depth of inheritance as metrics. He evaluated his model by correlating it with expert rating on 28 programs, and reported the correlation of 0,948. Van Koten [33] proposed a Bayesian network maintainability prediction model for an object-oriented software system and compared it with prediction of traditional regression models. The model used Li and Henry's

set of object-oriented metrics [34] collected from two different object-oriented systems. His results suggest that the Bayesian network model can predict maintainability more accurately than the regression-based models. Shibata *et. al.* [35] proposed a stochastic, queueing model for predicting software reliability and maintainability. The model used real software fault detection/correction data obtained from practice. Aggarwal *et. al.* [36] proposed an integrated fuzzy-logic model of understandability which takes into account the number of comments in the source code, the quality of documentation calculated as Gunnings Fog index [37] and the similarity between the language of the specification and the one used in the source code [5].

## 2.4 Cyclomatic complexity

McCabe [1] presented a graph-theoretic view on software complexity by abstracting software as a control flow graph. In that graph, nodes are groups (blocks) of commands of a program that can be executed only in (one possible) sequence. A directed edge connects two nodes if the first command in the target node *might* be executed immediately after the last command in the source node. Typically, edges in a control flow graph are result of conditional execution of a block of commands, such as *if* branches or conditional loops.

Presenting software as a control flow graph is interesting because it visualizes possible execution paths in the source code. Since the total number of possible paths may be hard, impractical or even impossible to calculate, McCabe introduced *basic paths*, a set of paths that can be used to construct all other paths.

For any single connected graph, the cyclomatic number or circuit rank represents the number of linearly independent cycles and is given with:

$$V(g) = e - n + 1 \tag{2.1}$$

where $e$ is the number of edges and $n$ is the number of nodes. For a more general case, when a graph contains several *connected components* (i.e. islands of connected nodes) the formula will have the following form:

$$V(g) = e - n + p \tag{2.2}$$

where $p$ is the number of connected components.

Assuming a control flow graph has a single entry and a single exit node, a control flow graph becomes strongly connected (each node is reachable from every other node) if we **add a single artificial directed edge from the last to the first node**. McCabe defines cyclomatic complexity of a program (module) as a cyclomatic number of a corresponding strongly connected graph calculated with:

$$V(g) = e - n + 2 \tag{2.3}$$

or in general case with $p$ connected components:

$$V(g) = e - n + 2p \tag{2.4}$$

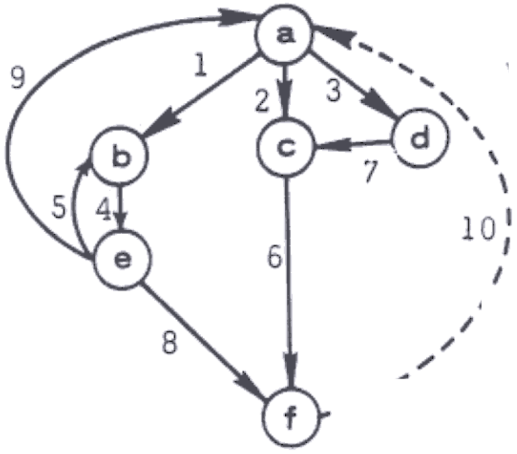



*Figure 2.9. Example of a control flow graph[1]*

Connected components in a control flow graph represent individual functions or subroutines of a program. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program. It is frequently used as basis for testing methodology because cyclomatic complexity defines the minimal number of test cases needed to achieve complete branch coverage (each possible branch of execution is exercised).

Cyclomatic complexity has several properties:

- Always larger or equal to 1: $V(G) \leq 1$
- It is a maximum number of linearly independent paths in graph
- Inserting or deleting a functional statement does not affect V(G)
- A graph has only one path if and only if V(G)=1
- Inserting a new edge increases V(G) by one
- V(G) depends only on the decision structure of G

The number of edges and nodes is not trivial to calculate for larger source codes, a simplification formula may be applied: $V(G) = D + 1$ where $D$ is the number of control flow decisions[4] in the source code. Because of its simplicity, this way of calculating cyclomatic complexity is probably the most widely used. For visually presented strongly connected control flow graphs, there is an additional simplification for calculating cyclomatic complexity by counting the number of regions of that graph.

[4]Notice that each decision may include more than one condition

### 2.4.1 Cyclomatic complexity and modularization

Modularization of code assumes splitting a bigger block of code into smaller units, typically through the use of subroutines. There are two main reasons for modularization: *abstraction* and *code reuse*. When modularization is done for the sake of abstraction, a logical, self-contained, part of processing code is identified within a bigger block of code and factored out, hopefully by using a name which clearly describes its functionality and follows all agreed naming conventions. Modularization for the sake of code reuse is commonly used to remove code duplication. Such modularization assumes one definition and several usages of the created subroutine. Those two types of modularization are fundamentally different from testing difficulty perspective. Modules that are reused several times, cannot be considered as a separate component each time they are called and need to be tested only once. This means that reused modules should only add to the cyclomatic complexity once, not as many times as they are used (called). However, this may not be the case if we consider cyclomatic complexity in the context of a general software complexity measure (or cognitive software complexity measure), since multiple calls to the same subroutine may actually increase general software complexity.

Even if we observe cyclomatic complexity only as a testing difficulty measure (not as a general or cognitive complexity measure) it may be calculated in several different ways. McCabe's paper [1] considers each component (subroutine) of the program separately. As a consequence, each of those components needs to be strongly connected (for each subroutine an artificial edge from end to start node has to be added). Contrary to McCabe's cyclomatic complexity, which is focused on module (or unit) testing, Henderson-Sellers metric [2], [38] is focused on the integration test paths. This means it is interested in counting the basic test paths of one big control flow graph and not many smaller ones. The merging of modules into one big control flow graph is achieved by the Henderson-Sellers splitting technique.

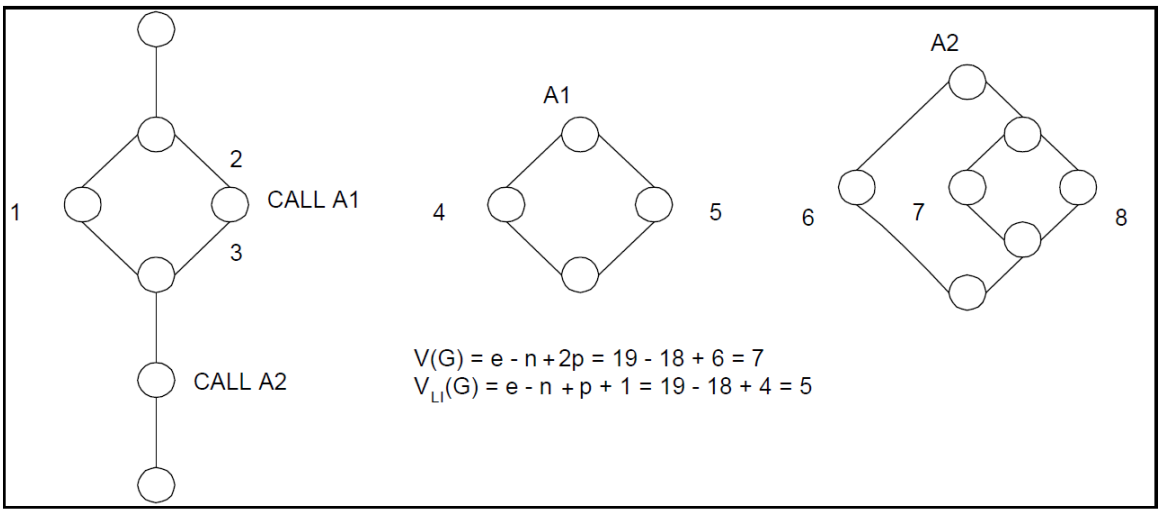



*Figure 2.10. Control flow graph as set of separate connected components[2]*

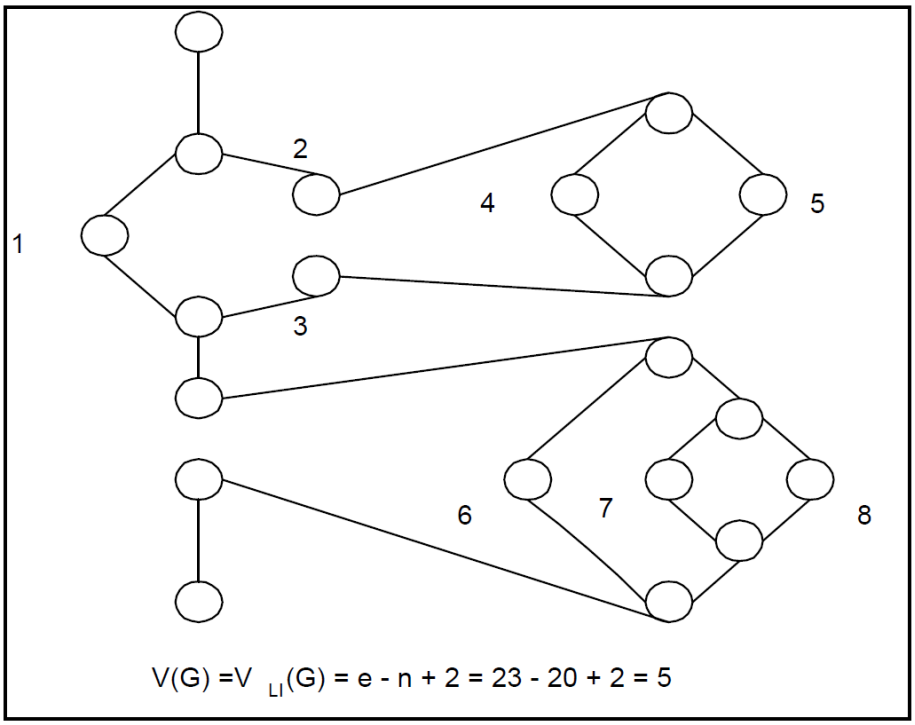



*Figure 2.11. Control flow graph as single integrated connected graph obtained by splitting technique of Henderson-Sellers[2]*

The graph in figure 2.10 is equal to the single graph from figure 2.11, when this technique is applied: each node calling a subroutine is replaced by 2 nodes (one for the entry in a subroutine and one for the exit) and 2 new edges are added (one for the entry and one for the exit from a subroutine). Now we can apply the well-known McCabe's formula for calculating cyclomatic complexity of a single-component control flow graph : $V(g) = e - n + 2$. The merging increases cyclomatic complexity by one (1=2-1) **for each connected component except for the one which we are merging into**.

$$V_{LI}(G) = e' - n' + 2 = [e + 2(p-1)] - [n + (p-1)] + 2 = e - n + p + 1 \tag{2.5}$$

where $e'$ and $n'$ respectively represent the number of edges and nodes in a single, merged, control flow graph. This altered cyclomatic complexity metric has different properties with regard to modularization and is much more suitable for integration testing. The value of $V_{LI}(G)$ for the full program is equal to the total number of decisions, D, plus one:

$$\sum_{i=1}^{p} d_i + 1 = D + 1 \tag{2.6}$$

The value of $V_{LI}(G)$ is unchanged when subroutines are merged back into the program either by nesting or sequence (see figures 2.10 and 2.11). This confirms the argument that the integration testing procedures are unchanged by modularization.

Relationship between the whole and the sum of the parts is also different. McCabe [1]

shows that $V(G) = \sum V(G_i)$ while Henderson-Sellers deduce that:

$$V_{LI}(G) = \sum V_{LI}(G_i) - (p-1) = \sum_{i=1}^{p} (e_i - n_i + 2) + 1 - p = e - n + p + 1 \qquad (2.7)$$

where $e$ and $n$ are the total number of edges and nodes, respectively. To summarize, McCabe (V(G)) treats modules in programs essentially independently, while Henderson-Sellers ($V_{LI}(G)$) retain an interpretation with respect to testing paths both at the unit level and at the complete program level.

Note that the last equation disregards modularization due code reuse by counting only the number of components and not the number of times they are called.

Multiple calls of a single subroutine do not add to the value of $V_{LI}(G)$ since it does not introduce additional control flow paths. This is aligned with the premise that testing difficulty should not be affected by multiple calls to same subroutine. It should be noted that this cannot be extrapolated to cognitive complexity or understandability: multiple calls of the same subroutine have an influence on software understandability and contribute to cognitive complexity.

### 2.4.2 Cyclomatic complexity for modules with multiple entry and/or exit nodes

One of the main assumptions in calculation of cyclomatic complexity is that each component has a single entry and a single exit node. Henderson-Sellers also considered cases where this assumption is not fulfilled and observed its effect on cyclomatic complexity.

Multiple entries in a module represent module reuse mechanism. Thus, equation $V_{LI}(G) = \sum V_{LI}(G_i) + 1 - p$ applies to *multiple entry, single exit* (MESE) modules as well.

Generally, *single entry multiple exit* (SEME) nodes typically occur in branches of *if* structures where an immediate return ("early exit") to the calling module occurs. One way to handle such cases is to add an additional (virtual) node for each early end and connect it with virtual edges to early and normal exit nodes. This means we need to add one for each exit point (+1 = 2 new edges - 1 new node). So the formula is given with:

$$V_{LIseme}(G) = e - n + p + 1 + \sum_{j=1}^{p-1} (r_j - 1) \qquad (2.8)$$

where $r_j$ is the number of exit points in the graph representation of *j-th* module and there are *p-1* such modules (subroutines). This means that modules with multiple exit points increase complexity.

In addition, *multiple entry, multiple exit* (MEME) modules can be treated as modules with multiple exit points, and the previous equation applies.

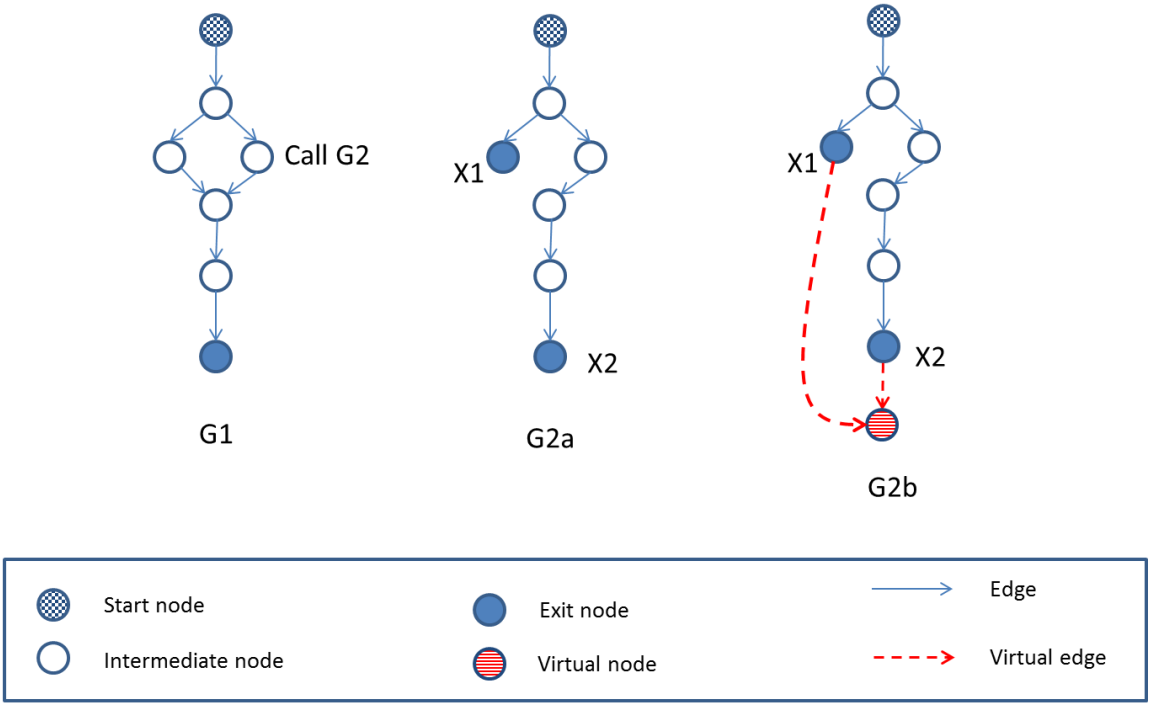



*Figure 2.12. Handling single entry, multiple exit (SEME) modules when calculating cyclomatic complexity.*

### 2.4.3 A critique of cyclomatic complexity as a software metric

McCabe's cyclomatic complexity is based on solid theoretical foundations and is useful as a measure of software testing difficulty, but it can also be considered as a general purpose software complexity measure or metric of cognitive software complexity. In that case, cyclomatic complexity may be criticised on several grounds [39].

First, there is an issue of compound predicates. We can choose either to count predicates or individual conditions, but both alternatives may be too simplistic. Myers [40] proposes a complexity interval which will have the *number of predicates + 1* as the lower bound and the *number of individual conditions + 1* as the upper bound.

The second issue of cyclomatic complexity is the failure to distinguish between *if* and *if/else* constructs: they have the same cyclomatic complexity because they have the same number of execution paths, the difference is only that the alternative ("else") path is explicitly stated.

The third issue is handling switch-case construct. Although it has the same cyclomatic complexity as an equivalent *if* construct, it is significantly simpler and should contribute to complexity with logarithmic scale $log_2(n)$.

There is also an additional issue that cyclomatic complexity remains one for any linear sequence of statements.

A more fundamental problem of cyclomatic complexity measure is the fact that **generally accepted techniques for modularization (splitting in subroutines) effectively increase**

**complexity because the number of connected components (p in the equation) increases**. The problem is even more complicated because graph complexity may be reduced by eliminating code duplication through modularization (when the same subroutine is called more than once). Cyclomatic complexity is then given with:

$$v(P') = v(P) + i - \sum_{j=1}^{i} ((v_j - 1) * (u_j - 1)) \tag{2.9}$$

where P is equivalent to P' but with a single connected component, $i$ is the number of modules, $v_j$ is the cyclomatic complexity of *j-th* module calculated using original McCabe's equation 2.3, and $u_j$ is the number of times *j-th* module is called.

This means that the complexity of a program increases with modularization, but decreases with factoring out duplicate code. This leads to the conclusion that programs should only be modularized for the parts that are reusable, but this is not an acceptable guideline to reduce complexity.

Suleman [41] indicated an additional problem with the way how cyclomatic complexity handles nested control structures. Although they have the same cyclomatic complexity as non nested (sequential) control structures, nested counterparts have greater *computational* complexity. As far as testing is concerned, this works fine, but when it comes to benchmarking the code, this is not acceptable. There are existing solutions for nested *if* problems but those cannot be applied to nested loops. Suleman proposes a solution to this problem by adding the total number of iterations of nested loop to cyclomatic complexity:

$$V(G)^* = V(G) + \prod_{i=1}^{n} P_i \tag{2.10}$$

where $n$ is nesting depth (the number of nested loops) and $P$ represents the number of iterations in each loop. The problem with this approach is that the number of iterations is usually unknown at design time. This makes the calculation of computational complexity hard. Also, note that this formula works only if we have a single top-level loop.

In addition to theoretical objections, there are also some empirical objections related to cyclomatic complexity as a general software complexity metric. The number of empirical studies have been carried out with different interpretations: The problem is that there is no explicit hypothesis being evaluated. Possible *a posteriori* hypothesis which can be used to examine empirical work are:

1. Total program cyclomatic complexity can be used to predict various software characteristics (such as development time, incidence of error and program comprehension)

2. Programs with cyclomatic complexity lower than 10 are easier to test and maintain than those where this is not the case (original McCabe's hypothesis)

Empirical data does not a give great deal of support for either of the hypothesis. The

clearest empirical result indicates strong relationship between the number of lines of code (LOC) and cyclomatic complexity (CC). This is not very good because one of the motivations for cyclomatic complexity is the inadequacy of LOC as complexity metric. In addition, there are many studies that show that LOC actually outperforms CC.

A lot of empirical validations are done by measuring the correlation with Pearson's product moment as a coefficient. That correlation gives a value between -1 and 1 where 0 indicates no correlation while -1 and 1 indicate strong, negative or positive correlation. However, that coefficient requires roughly normal distribution, which is particularly problematic when correlating cyclomatic complexity and module error rates because it is impossible to get negative error count and the corresponding distribution is skewed.

Empirical validation is very often done in two scenarios: large uncontrolled and small controlled ones. Both have issues. Large scale empirical validations have problem of not taking into account the individual ability of a single programmer. Small scale empirical validations are usually done on small (trivial) program samples (up to 300 LOC) and do not take into account programmer familiarity with the problem.

## 2.5 Entropy-based complexity metrics

Several authors [42], [43] [44] considered software as an information source and computed software complexity as some variation of classical Shannon's ***entropy*** [45] of the source code:

$$H_n(p) = -\sum_{k=1}^{n} p_k log_2 p_k = \sum_{k=1}^{n} p_k log_2(1/p_k) \tag{2.11}$$

where $p_k \geq 0 (k = 0, 1, 2, 3...n)$ and $\sum_{k=1}^{n} p_k = 1$, $n$ is the number of symbols (events), and $p_k$ is the probability of *k-th* symbol. The main issue with this entropy complexity metrics is the selection of symbols. Once defined, their probabilities are calculated by counting their appearance in the source code. Abd-El-Hafiz [42] counted only function/method calls, while Harrison [43] additionally counted reserved words and special symbols.

Kim et. al. [44] took a somewhat different approach. They constructed intra- and inter-class dependency graphs built from object oriented software and used the entropy-based approach for calculating complexity. For each class, they defined a *Data and Function Relationship* (DFR) graph that represents dependencies between data and function members of a class and between function members as well. They represent each data and function member as a node, using weights on arcs to denote the number of times a data member is read or written, or, in the case of a function, how many times it has been called or how many times it calls other functions. The symbols used in entropy calculations are nodes in the graph. The probability of each symbol (node) required for entropy function is calculated as the division of the sum of node weights (on all incoming and outgoing arcs) and duplicated sum of all

weights in the graph (because each weight is counted twice, for originating and terminating node).

In a similar way, the inter-object graph, called *Object Relationship* (OR), is constructed, with nodes being classes, arcs being messages between them, and weights being the number of times a given message is called from the originating class. Weights in that graph are then used to calculate intra-object entropy representing intra-class complexity of software. Total complexity of software is then calculated as the sum of intra- and inter-object entropy of all classes in the software.

## 2.6 Data and information flow complexity metrics

Henry and Kafura [46] defined a set of software metrics based on the information flow between system components. This set includes metrics for procedure complexity, module complexity and module coupling. The following information flow types are identified:

1. *Global information flow* between module A and B happens when there is structure D in which module A writes and module B reads
2. *Direct local flow* from module A to B happens if A calls B
3. *Indirect local flow* from A to B happens if B calls A and A returns a value which B later uses or in case when the third module C calls A and B, passing output from A as a parameter into B

Metrics based on information flow are measuring *simplicity of relations between modules*. For single procedure, Henry and Kafura defined **fan-in** and **fan-out** as:

- **Fan-in** of a procedure is the number of local flows into the procedure plus the number of data structures from which A retrieves information
- **Fan-out** of a procedure is the number of local flows from the procedure plus number of data structures which procedure updates

Procedure complexity depends on two aspects: complexity of its internal code and, complexity of procedure's connections with its environment. With the number of lines of code as measure of complexity of its internal code, procedure complexity can be formalized as [46]:

$$length * (fan_{in} * fan_{out})^2 \tag{2.12}$$

Fan-in and fan-out are lifted to the power of 2 because of the **belief** that complexity is more than linearly dependant when it comes to connections to its environment. If the effect of the number of lines of code ($length$) is estimated or ignored, the metric can be calculated

even before the implementation phase. This is important in order to achieve the *design-measure-redesign* cycle instead of the usual *design-implement-test-redesign* cycle.

In the previous formula, the product of $fan_{in}$ and $fan_{out}$ represents the total number of input-output flow combinations, which is a result of a simplistic assumption that each input of procedure affects each output. Detailed data-flow analysis, similarly to the one typically done in compiler optimizations, may be performed to improve this [47], [48], [49].

Analysis of the formula for procedure and module complexity may lead us to some useful conclusions. Procedures with high fan-in and fan-out numbers have many connections which might indicate that they perform more than one task. In addition, such procedures are considered as stress points where changes have many effects to the environment. Procedure complexities in a module are summed to obtain module complexity. High module complexities typically indicate improper modularizations. High global flows and low or average module complexity indicate poor internal construction of modules. In that case procedures directly access data structures, but there is little communication between procedures. In the case of low global flows and complicated module complexity, it is probable that functional decomposition within module is poor or there is a complicated interface between modules.

An important aspect of any metric is that its calculations is completely automated so it can easily be applied to large scale systems. Henry and Kafura validated their metric set on the source code of the Unix operating system (version 6.0) and found strong correlation (0.94) with occurrence of changes (errors). Additionally, they noticed that most of the modules complexity (85%) comes from three most complex procedures [46].

Unlike Henry and Kafura [46], Oviedo [49] did not assume that every input of procedure affects every output variable. Inspired by compiler optimization techniques, he defined precise *data flow complexity* metric which counted the number of all prior (re)definitions of all locally used variables. The following text explains the metric in a more details using the following concepts:

- *Variable definition* appears as a left side expression in an assignment statements or as an input parameter of a subroutine.
- *Variable reference* appears in expressions, typically the right side of an assignment expression, as part of predicates or is used in subroutine output statement (i.e. "return" statement).
- *Locally available variable (re)definition* is a (re)definition of the variable within a block.
- *Locally exposed* variable reference is a variable that is used (referenced) within a block, but is not (re)defined within it.
- Variable definition defined in block $k$ is said to *reach* block $i$ if it has not been (re)defined along the path from block $k$ to block $i$.

- Variable (re)definition in the block *kills* all previous definitions of that variable that might *reach* the block.

Let $V_i$ be set of locally exposed variables (set of variables used) in block $i$ and $R_i$ be set of variable definitions reaching block $i$. Note that, for each variable used (referenced) in block $i$ there might be several definitions, depending on the number of possible control paths leading to the block $i$.

*Data flow complexity* of block $i$ is defined as the number of (re)definitions reaching block $i$ of all the variables used (i.e. referenced, *locally exposed*) in that block, or formally:

$$DF_i = \sum_{j=1}^{|V_i|} DF(v_j) \tag{2.13}$$

where $|V_i|$ is the number of variables used in block $i$ and $DF(v_j)$ is the number of definitions of variable $v_j$ reaching block $i$. *Data flow complexity* of program body is then defined as a sum of data flow complexities of all blocks in the program. Note that only inter-block data flows contribute to data flow complexity. Such definition is closely related to the "all-uses" test data selection criteria which requires that all *definition-reference* pairs are exercised [50].

Oviedo's data flow complexity metric is one of the first complexity metrics that has focused on the complexity of data manipulation within a block of processing code. That makes this metric context dependant, which is a property not typically present among other complexity metrics. In its nature, data flow complexity metric is independent of, at least theoretically, control-flow complexity metric described by McCabe [1].

# 3 Measuring complexity of xtUML models

This chapter describes how a selected set of traditional software complexity metrics can be applied to xtUML (sub)models. This work represents one of the main scientific contributions of this dissertation.

## 3.1 Cyclomatic complexity of xtUML models

xtUML builds applications from four different types of models: component, class, state-machine and processing model. Component and class models describe the structure, while the state machines and action code describe the runtime behaviour of the application.

When calculating cyclomatic complexity of an application, we are interested in application runtime behaviour, so the focus should be on the behavioural aspects of the model: state machines and processing code. However, structural parts of xtUML models are considered to partially visualize model's cyclomatic complexity. Because of this, when discussing cyclomatic complexity of structural models, we discuss the complexity of processing code visualized by those models. In this section, we present our approach for calculating cyclomatic complexity visualized by structural parts of the application (component and class models), as well as the total cyclomatic complexity from the behavioural parts of the model (state machines and processing code). These metrics are then utilized in order to determine the distribution of cyclomatic complexity across different model layers.

### 3.1.1 Cyclomatic complexity of components

Components are foundational building blocks of xtUML models. They are considered as black boxes that communicate though their interfaces. For this reason, the cyclomatic complexity of xtUML components mostly depends on the interfaces that the component uses. Since the component model falls into the category of structural models, we cannot calculate its cyclomatic complexity, but we can calculate the complexity of visualized entry and exit points to the behaviours wrapped within the component.

The basic approach for calculating cyclomatic complexity is constructing a strongly connected single-entry, single-exit (SESE) control flow graph (CFG). Unfortunately, an xtUML component typically has many entry and exit points, and it is not trivial to construct an SESE CFG from it.

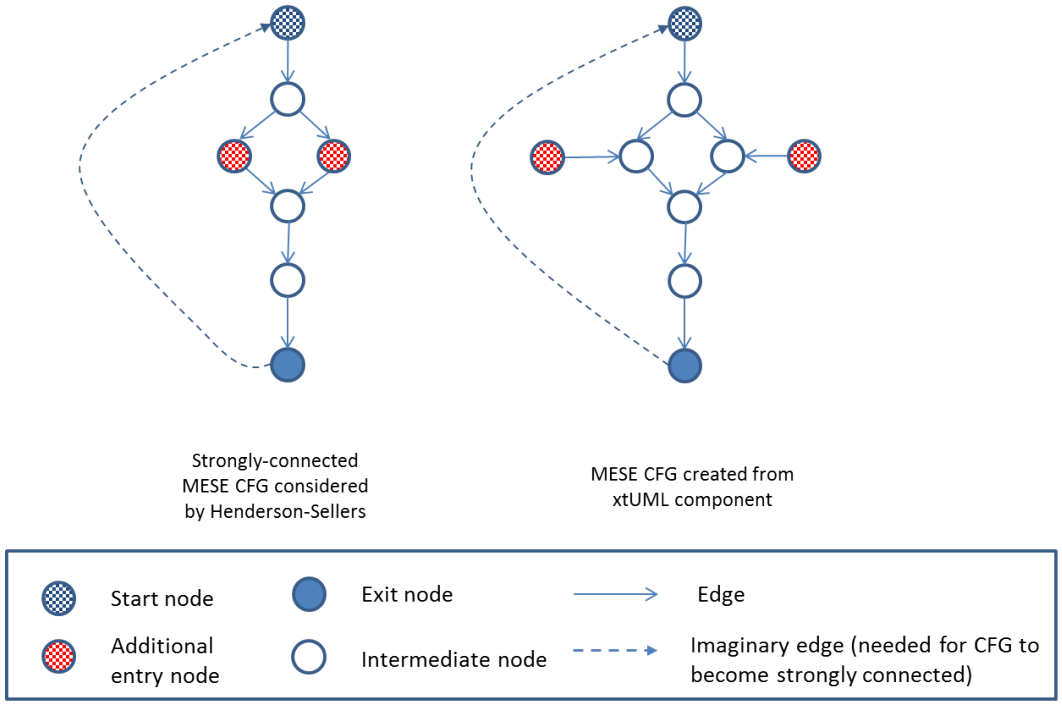



*Figure 3.1. MESE CFG considered by Henderson-Sellers [2] and the one created from xtUML component.*

The original Henderson-Seller's approach states that multiple entries to the module can be ignored because multiple entries are used as a reuse mechanism for parts of the CFG already taken into account. If we apply this idea to xtUML components, an estimation of cyclomatic complexity of an xtUML component should be calculated by taking into account only exits on all component ports, while entries to the component can be ignored. However, the issue when applying the Henderson-Seller's approach to xtUML components is dealing with multiple entries. Henderson-Sellers considered only strongly connected CFGs with multiple entries, because this is the limitation introduced in the original McCabe's calculus [1]. This means that additional entry nodes in a MESE CFG are reachable even when they are not used as entry nodes (left graph in figure 3.1). In other words, there is at least one incoming edge leading to each entry node, implying that there exists at least one control flow path in which those nodes are not entry nodes. This justifies ignoring multiple entry nodes when calculating cyclomatic complexity using the Henderson-Sellers approach.

However, an entry point into an xtUML component is an entry point to an implementation of a port incoming message which cannot be invoked from behaviours within a component. This means that entry nodes are not part of any existing path in a CFG and that such graphs are not strongly connected (even when a *single* virtual edge is added; the right graph in figure 3.1). Consequently, neither the original, nor the adapted Henderson-Sellers calculation can be applied to such graphs. To handle this, we will adapt an approach similar to what Henderson-Sellers used for single-entry, multiple-exit modules. For $n$ entry nodes, we will add one new (virtual) node and $n$ new edges to connect the new virtual node with all entry

nodes. The new virtual node will then act as a single entry for the CFG which will make the graph an SESE. This effectively increases cyclomatic complexity by $n-1$. When we incorporate this with the existing Henderson-Sellers [2] formula applied to MEME CFGs we get:

$$CC_{meme} = e - n + p + 1 + \sum_{j=1}^{p-1}(r_j - 1) + \sum_{k=1}^{p-1}(i_k - 1) \tag{3.1}$$

where $r_j$ is the number of exits from $j$-th connected component and $i_k$ is the number of entries to the $k$-th connected component. In case there exists only a single connected component, the equation becomes:

$$CC_{meme} = e - n + 2 + (r-1) + (i-1) = e - n + r + i \tag{3.2}$$

where $r$ is the number of exits and $i$ the number of entries to the CFG. Notice that eq. 3.1 and eq. 3.2 assume the existence of at least one exit and at least one entry, in which case we get the original Henderson-Seller equation. Applying this to eq. 3.2 results with:

$$CC_{compFull} = e - n + r + i = e - n + N_{op} + N_{sig} \tag{3.3}$$

where $N_{sig}$ denotes the number of signals and $N_{op}$ the number of operations on all ports of a given xtUML component. Notice that we did not differentiate between synchronous and asynchronous entries and exits assuming that they equally contribute to cyclomatic complexity. For details about this please refer to section 3.1.4 and figures 3.11 and 3.13.

Eq. 3.3 refers to the complete CFG of an xtUML component, including all of its content. The part of overall xtUML model cyclomatic complexity visualized by the component model can be calculated by the following equation:

$$CC_{comp} = N_{op} + N_{sig} \tag{3.4}$$

The Henderson-Sellers approach (and our alteration for multiple entries) assumes that a complete component CFG has at least one entry and at least one exit node. However, in xtUML, a component does not always need to have ports or messages on ports. Such components do not communicate with other components, their complete CFG is closed inside the component which is used only as a package. In that case, entry and exit nodes still exist but they are not explicitly exposed in the component model which, in that case, does not give us any information about cyclomatic complexity.

Notice that semantically equivalent interfaces may expose different cyclomatic complexities, depending on the level of abstraction used to create messages on the interface. On the figure 3.2, notice that the less abstract version visualizes more of the component cyclomatic complexity. A more general interface will have fewer messages, but it will also have a larger total number of parameters. For example, consider a simple calculator component: instead of

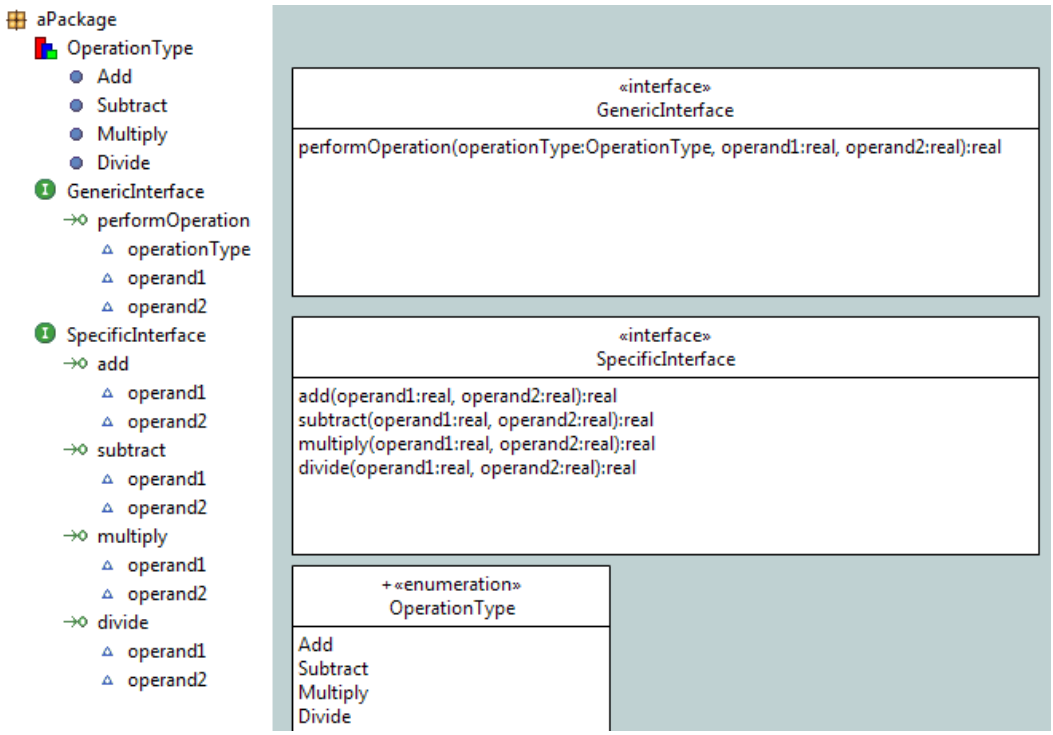



*Figure 3.2. An example of two functionally equivalent interfaces at different level of abstraction.*

four messages *add*, *subract*, *divide*, and *multiply* with two parameters, we can have a single message (*performCalculatorOperation*) with three parameters where the additional parameter encodes the calculator operation to perform. The total cyclomatic complexity of the component remains the same because the port message code in the case with single message will need to branch according to the value of the third parameter. This implies that, although eq. 3.3 seems to depend on the number of operations and signals on component interfaces, it actually does not, because its effect is compensated with the remainder of the equation (the $e - n$ part). At the same time, cylomatic complexity visualized by the component model is lower in case of a more general interface because only one interface operation is used. Despite the functional and cyclomatic equivalence of models, a model with specific interface is obviously simpler to comprehend because it states the interface functionality more clearly.

### 3.1.2 Cyclomatic complexity of classes

Similarly as the component model, the class model describes the structure, and not the behaviour of an xtUML model. As with a component model, this implies that a class model only partially visualizes the cyclomatic complexity of processing.

A class model visualizes the cyclomatic complexity of processing code with its operations and with the relations it exposes. Similarly to port operations, each class operation adds *two edges* and a *single node* to the CFG effectively increasing the cyclomatic complexity visualized by the class model by one (see figure 3.11).

Class relations also have an influence of the processing model. The processing code operates on class instances; it creates and relates them, selects them across relations, reads or changes their values, unrelates them and eventually deletes them. Single- or multi-hop instance selections across relations are a very important part of the processing model, because each relation (chain) exposes specific meaning within an application. Depending on the multiplicity and the conditionality of a relation traversed in a selection statement, it will be followed by a *i)* conditional loop (iterating across selected instances), *ii)* a conditional *if* branch (checking for emptiness of a selection variable), *iii)* both, a loop and an *if* branch (if the relation was conditional and multiple) or *iv)* neither, if the relation was unconditional and single. A loop for iteration across selected instances and/or a check for result emptiness are needed in order to evade runtime errors. This is how selections influence the cyclomatic complexity of processing code.

The general approach of calculating cyclomatic complexity visualized by a relation in a class model relies on the assumption that all directions of all relations in a class model are used in at least one selection statement in the processing code. For simple relations there are only two possible selection directions: from one class to another, and in reverse (see figure 3.3). The result of each of those selection directions may be conditional and/or multiple, depending on the flags on the remote end of the relation direction used in the selection. If the result of a selection may be empty, a single *if* branch (required to check the emptiness of the result) will increase cyclomatic complexity by 1. If a selection may result in a set of instances, a *for* loop that iterates through the instances will also increase the cyclomatic complexity by 1. In the worst case scenario, where both relation ends of a simple relation are conditional and multiple, **there are four additional control-flow paths added**. On the figure 3.3, a dog may or may not have an owner, implying that we need to check for emptiness when selecting dog's owner. In the reverse direction, each dog owner may own one or more dogs. This means that we do not have to check for emptiness, but we do have to iterate through all instances to apply the same processing to all owned dogs.

Each generalization relation (see figure 3.4) defines $2n$ selection directions: $n$ directions towards $n$ subclasses and $n$ in the reverse direction. However, the $n$ directions from subclasses towards a general class are always unconditional and single and do not require a conditional check or an iteration. The remaining $n$ directions towards subclasses are always conditional (they require an emptiness check with a conditional *if* branch) and single (they do not require a conditional loop iteration). This means that each generalization relation increases cyclomatic complexity by $n$ additional control-flow paths. On figure 3.4, when selecting a *person* related to a *male*, we do not have to check for emptiness or iterate because we are sure there will be exactly one instance. In reverse direction, when we select a *male* from a *person*, we have to check for emptiness because a person might be a *female*, in which case male selection results in an empty reference.

With associative relations there are three classes involved: two *related* classes and a

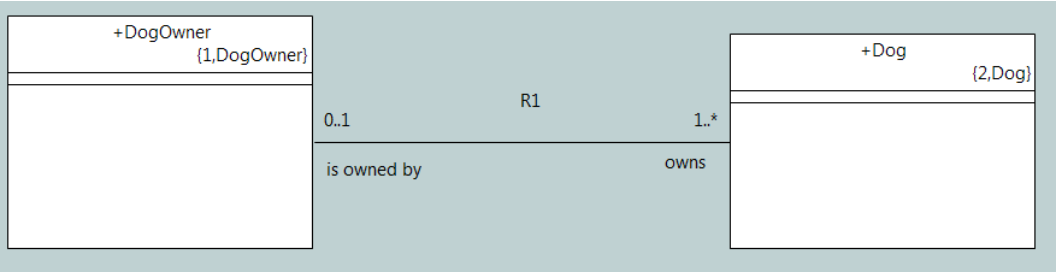



*Figure 3.3. An example of a simple xtUML relation.*

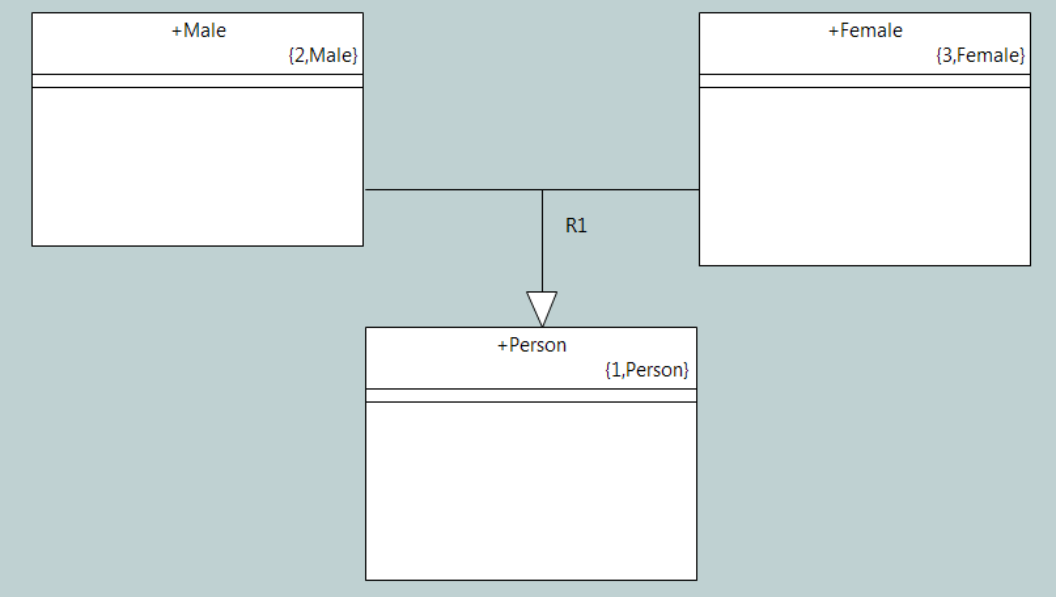



*Figure 3.4. An example of a generalization relation.*

single *associative* class. Each of these classes may be involved in a selection in two different ways, which means that there are six different selection directions, each with a different meaning. However, the multiplicity and the conditionality of some of those directions is not trivial to detect. For this reason, we will use a *replacement* model consisting of two simple relations, from each *related* class towards the associative class. We will refer to these two models as the *original* and the *replacement* class models.

Since the existence of an associative class instance assumes the existence of a pair of related classes, all selections starting from an associative instance always result with a single instance. This implies that relation ends towards related classes are unconditional and single. A selection starting from one of the related classes to another related class, and a selection from the same related class towards the associative class share the same conditionality. This simply means that if there is no associative class instance found, there should be no remote related class instance found as well.

The same rule applies to the multiplicity flag, but only in the case when the associative link is single. In this case, the multiplicity flag at an associative class end in the replacement class model is the same as multiplicity of the direction towards the remote related class in

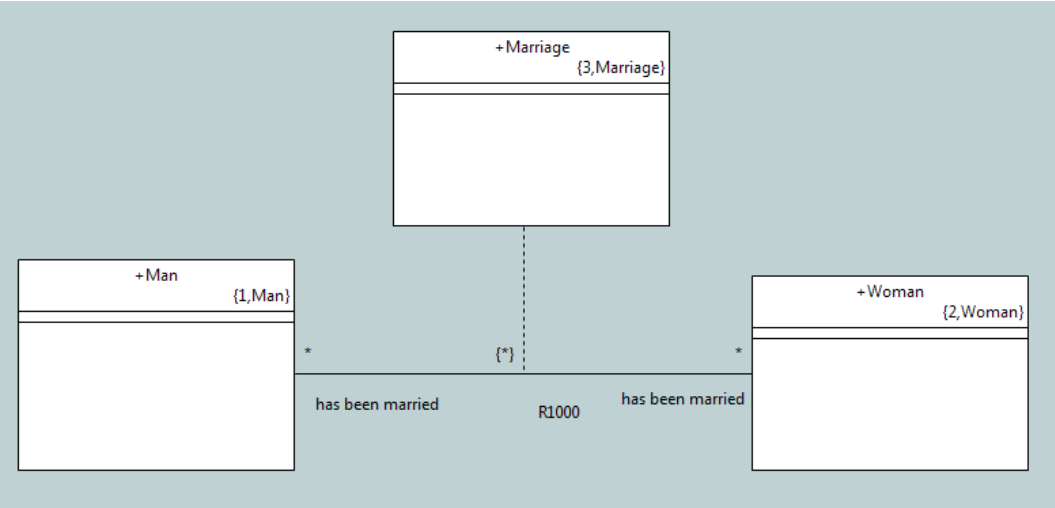



*Figure 3.5. An example of associative relation.*

the original model. However, when the associative link is multiple, the relation ends at the associative class on the replacement model are always multiple, regardless of the multiplicity in the original class model (see figure 3.6).

On figure 3.5, a man may have been married with more than one woman (*Man* $\rightarrow$ *Woman*). In addition, a man may have been involved in more than one marriage (*Man* $\rightarrow$ *Marriage*). However, since a man can be married to the same woman more than once (indicated by star in curly brackets), multiplicity of that selection is not the same as the one towards a women. This is not true regarding the conditionality: if a man has been involved in at least one marriage, he surely has been married to at least one woman. This means that two selection directions share conditionality, but not the multiplicity flags.

Now that we know how to determine the multiplicity and conditionality of all the selection directions across associative relation, we can determine its effect on cyclomatic complexity. It is important to emphasize that, from cyclomatic complexity perspective, not all branches are independent. This is a consequence of the fact that the conditionality flag of the selection direction towards associative class is always the same as the conditionality of the selection direction towards the remote related class. This actually lowers the cyclomatic complexity because, without any semantic change, one of the dependent branches can be eliminated and their processing can be merged. Table 3.1 summarizes the associative relation effect to processing code cyclomatic complexity.

### 3.1.3 Cyclomatic complexity of state machines

An xtUML state machine describes runtime behaviour of a class instance. This implies that, unlike component or class models, cyclomatic complexity of state machines directly influences the overall xtUML model cyclomatic complexity. In this section we describe two approaches for calculating cyclomatic complexity of xtUML state machines.

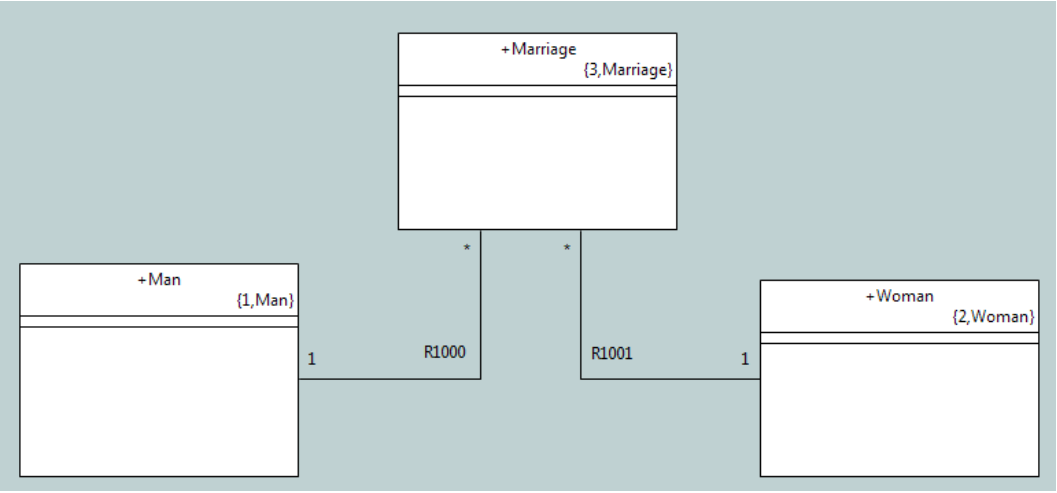



*Figure 3.6. A replacement class model for an associative relation.*

*Table 3.1. Summary of associative relation effect to cyclomatic complexity*

| Flag | Adds to cyclomatic complexity of CFG | Count | Maximal total effect |
|---|---|---|---|
| Conditionality on related class end | +1 | 2 | +2 |
| Multiplicity on related class end | +1 | 2 | +2 |
| Multiplicity of associative link | +2 | 1 | +2 |
| Total | | | +6 |

Calculating cyclomatic complexity of UML-based state machines is relatively straightforward. The basic formula is given with the following equation [51]:

$$CC_{sm1} = N_t - N_s + 2 \tag{3.5}$$

where $N_t$ is the number of transitions and $N_s$ the number of states within a state machine. Eq. 3.5 is obtained from the original McCabe's formula by considering the state machine graph as a CFG: each state is represented by a node and each state transition by an edge.

This is the most common way of constructing a CFG from an xtUML state machine. The problem with this approach is that it does not consider the complete xtUML state machine described by the state-event matrix. On figure 3.7, notice the *Event Ignored* case when event *OPEN* is received in *WORKING* state and *Can't Happen* case when event *CLOSE* is received in state *CLOSED*.

An additional problem is that, strictly, a state machine is not a complete control flow graph (CFG), but only a part of it. When discussing cyclomatic complexity, a CFG describes possible paths of a single, **uninterrupted** thread of execution. However, this is not how a state-machine actually works. From the modeler's perspective, each instance state machine

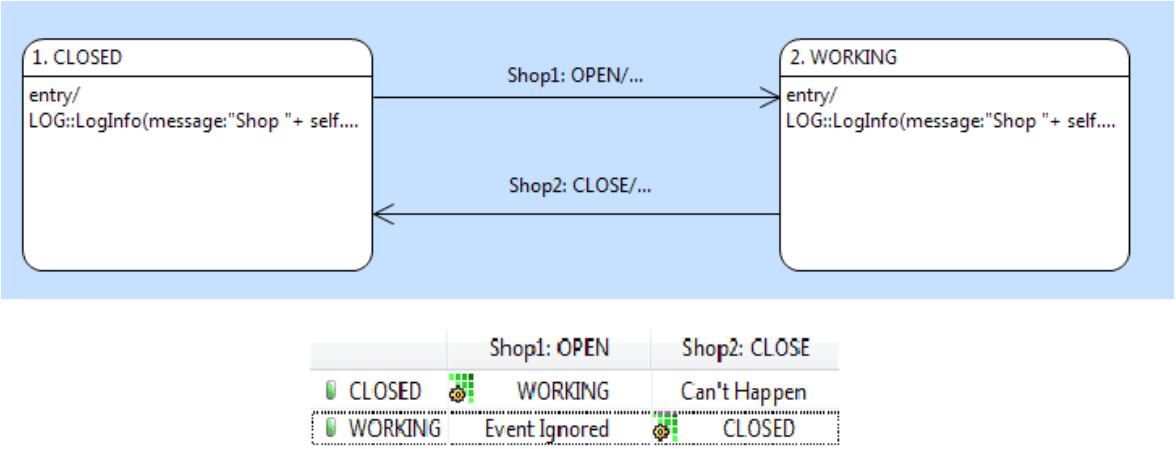



*Figure 3.7. State machine of the* Shop *class.*

```
1  BEGIN
2      SET running = true
3      SET currentState = CLOSED
4      WHILE running
5          SET event = dequeue event from the event pool
6          IF event is not empty
7              IF currentState is WORKING
8                  IF event is CLOSE
9                      execute WORKING->CLOSED transition effect behaviour
10                     SET currentState = CLOSED
11                     execute CLOSED state entry behavior
12                 ENDIF
13                 (Do nothing if event is OPEN)
14             ELSE IF currentState is CLOSED
15                 IF event is OPEN
16                     execute CLOSED->WORKING transition effect behaviour
17                     SET currentState = WORKING
18                     execute CLOSED state entry behavior
19                 ELSE IF event is CLOSE
20                     SET running = false
21                     SET currentState = ERROR
22                 ENDIF
23             ENDIF
24             IF currentState is final
25                 SET running = false
26             ENDIF
27         ENDIF
28         wait for some time before iterating again
29     ENDWHILE
30 END
```

*Figure 3.8. A pseudo-code description of the execution semantics of Shop instance state machines.*

gains execution control when it receives an event, resulting in the execution of a corresponding transition effect and state entry behaviour; a *Request-To-Completion* (RTC) step (see section 2.1.3 for details). After an RTC step execution finishes, the state machine execution

is **paused** until the next event is received. While waiting for the next event, the state machine temporarily loses execution control. Such behaviour is not inline with premise that CFG represents possible paths of uninterrupted execution. The control flow graph represented by a state machine should be extended to reflect the expected behaviour.

Since state-machines communicate asynchronously, a new event may be received while the instance is processing an RTC step. For this reason, there exists an *event pool* that stores events that are waiting to be processed, along with a dispatching mechanism that continuously checks for incoming events and triggers the corresponding state machine RTC executions.

Figure 3.8 shows a snippet of pseudo-code that describes the execution semantics of *Shop* instance state machines. In every while loop iteration, first the event pool is checked for any waiting events (line 5). In case an event exists, its event type is checked (lines 8, 15 and 19) and the corresponding RTC step (transition effect and the state entry behaviour) executed. In case of an *Event Ignored*, the loop simply skips an iteration without doing any work (line 13), while in case of a *Can't Happen* (lines 19 – 22), the loop is stopped, and the state machine is terminated.

By using this execution semantics, a complete CFG with a single entry and a single exit node can be constructed. A concrete example that shows a state machine CFG obtained through this execution semantics is shown in figure 3.9. The part of the graph outside the rectangle does not depend on the state machine and will be the same for any state machine because it represents the state machine infrastructure code. This implies that, for smaller state machines such as this one, cyclomatic complexity introduced by state machine execution infrastructure is comparable to the cyclomatic complexity of the state machine itself. However, for larger state machines this will not be the case.

In figure 3.9, the part within the dashed rectangle actually depends on the structure of the state machine. For each state in the state machine we have two *state* nodes in the CFG, which are connected to infrastructure nodes with two edges. For each non-ignored event in the state, we have an *event* node connected to the state nodes with two edges. An *event* node actually contains RTC execution as a synchronous invocation of *transition effect* and *state entry* actions. In case a state ignores any number of events it will also have a single direct edge between the two state nodes, regardless of the number of ignored events. We will use this rule and figure 3.9 to induce the general rule for calculating strict cyclomatic complexity of an xtUML state machine:

$$N_{edge} = 12 + 2N_{st} + 2\sum_{s=1}^{N_{st}} [N_{ev} - N_{ei}(s)] + N_{stei} + 2N_t \tag{3.6}$$

$$N_{node} = 8 + 2N_{st} + \sum_{s=1}^{N_{st}} [N_{ev} - N_{ei}(s)] \tag{3.7}$$

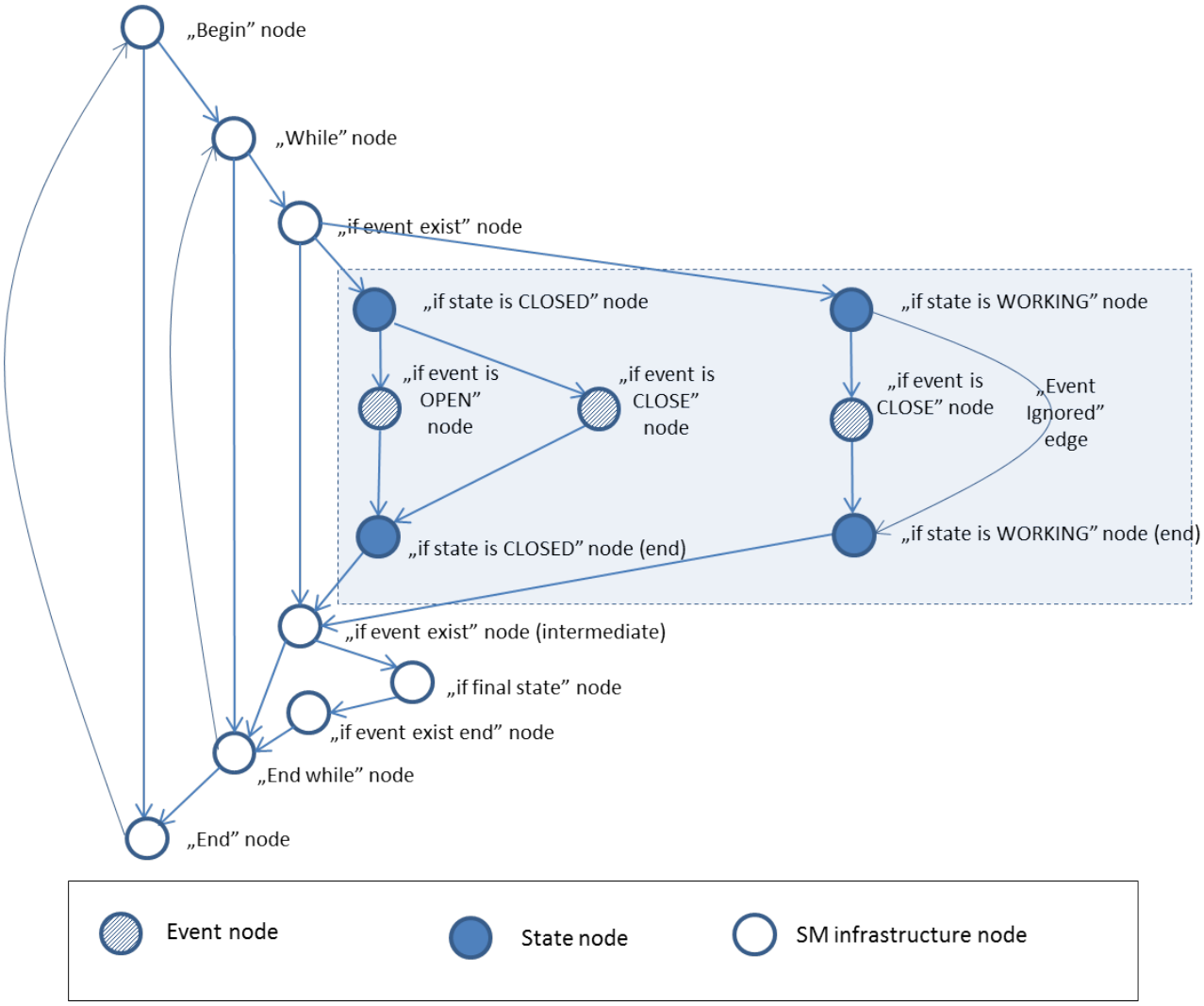



*Figure 3.9. A CFG created from the execution semantics of the Shop class instance state machine.*

$$CC_{sm2} = N_{edge} - N_{node} + 2 = 6 + \sum_{s=1}^{N_{st}} [N_{ev} - N_{ei}(s)] + N_{stei} + 2N_t \tag{3.8}$$

where $N_{st}$ is the number of states, $N_{ev}$ the number of events, $N_{ei}(s)$ the number of *Event Ignored* records the state $s$ has in the matching row of the state machine matrix, $N_{stei}$ the number of states that have at least one *Event Ignored* record in the matching row of the state machine matrix and $N_t$ the number of transitions. Since the number of transitions is actually the number of RTC steps and each RTC step contains (invocations of) exactly two bodies, each RTC step adds 2 to the overall cyclomatic complexity of a state machine (therefore the $2N_t$ in the equation). Notice that, if a state has more than one *Event Ignored* record in the matching row, it will still have only one "event ignored" edge in a CFG (see figure 3.9). This explains the $N_{stei}$ part of the equation. If each state has at most one *Event Ignored* record, $N_{stei}$ is equal to the total number of *Event Ignored* cases in the state machine. This results with the following equation:

$$
\begin{aligned}
CC_{sm2} &= 6 + \sum_{s=1}^{N_{st}} N_{ev} - \sum_{s=1}^{N_{st}} N_{ei}(s) + N_{stei} + 2N_t \\
&= 6 + \sum_{s=1}^{N_{st}} N_{ev} - N_{stei} + N_{stei} + 2N_t = 6 + N_{st} * N_{ev} + 2N_t
\end{aligned}
\tag{3.9}
$$

In eq. 3.9, we can observe that the state machine execution infrastructure adds a constant amount (6) to cyclomatic complexity. This constant becomes less significant as the state machine grows. In addition, such calculus for cyclomatic complexity is inline with the intuition because the product of the number of states and the number of events actually represents the number of possible linearly independent paths through the state machine, while the $2N_t$ denotes the number of actions we invoked in those paths.

Note also that the constant amount of 6 is introduced by the state machine infrastructure and does not describe complexity directly visible in the state machine model. Therefore, if we are trying to calculate the *visible* state machine complexity we can ignore the constant in our calculus:

$$
CC_{sm2vis} = \sum_{s=1}^{N_{st}} [N_{ev} - N_{ei}(s)] + N_{stei} + 2N_t \approx N_{st} * N_{ev} + 2N_t \tag{3.10}
$$

### 3.1.4 Cyclomatic complexity of processing code

An xtUML processing model represents textual processing instructions and is similar to more traditional programming languages. For this reason, from the cyclomatic complexity point of view, the processing model follows similar control- and data-flow rules as other traditional programming languages. This implies that standard, source-code level complexity metrics can easily be applied to the processing model.

In xtUML, code is organized into OAL actions (often called *bodies*) that can be associated with different model elements. There is more than one way to calculate cyclomatic complexity of a body and we need to make a set of decisions:

- Which basic approach to use?
- How to handle multiple calls of the same body (subroutine)?
- How to handle asynchronous calls?
- How to handle compound branching conditions?

In the following chapters we will analyse each of this questions and propose a single way to calculate cyclomatic complexity of an OAL body.

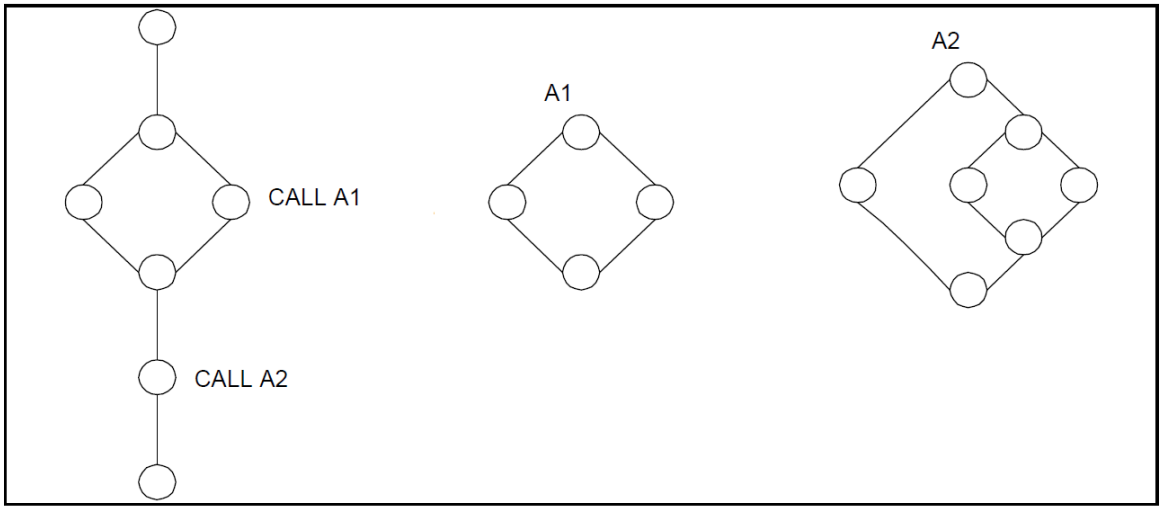



*Figure 3.10. McCabe's approach to cyclomatic complexity.*

### Choosing the basic approach

When choosing the basic approach for measuring cyclomatic complexity of processing code there are two options: the original McCabe's approach [1] which considers each subroutine separately, and the Henderson-Sellers approach [2] which creates a single connected component from all subroutines. In order to make this decision, we have to consider their effects on cyclomatic complexity.

In the original McCabe's approach, the CFG of each subroutine has to be strongly connected. This means that each subroutine should have its single entry and single exit node connected with an additional virtual edge. The resulting equation for McCabe's cyclomatic complexity of CFG with $p$ connected components is:

$$CC_{McCabe} = \sum_{i=1}^{p} [N_{edge}(i) - N_{node}(i) + 1 + 1] = N_{edge} - N_{node} + 2p \tag{3.11}$$

In Henderson-Sellers approach, a single control flow graph is created from all subroutines. To achieve this, a node containing a call to the subroutine is split into two nodes: one used to connect to the subroutine entry node and one used to connect back from the subroutine exit node. This means that, for each subroutine except the one we are merging into, we have to add a single additional node and two additional edges, and each subroutine increases cyclomatic complexity by 1. For this reason eq. 3.12 contains $(p-1)$. In order to make this complete CFG strongly connected, a single edge from the exit node to the entry node has to be added (thus the $+1$ in eq. 3.12).

$$CC_{HS} = N_{edge} - N_{node} + 1 + (p-1) + 1 = N_{edge} - N_{node} + p + 1 \tag{3.12}$$

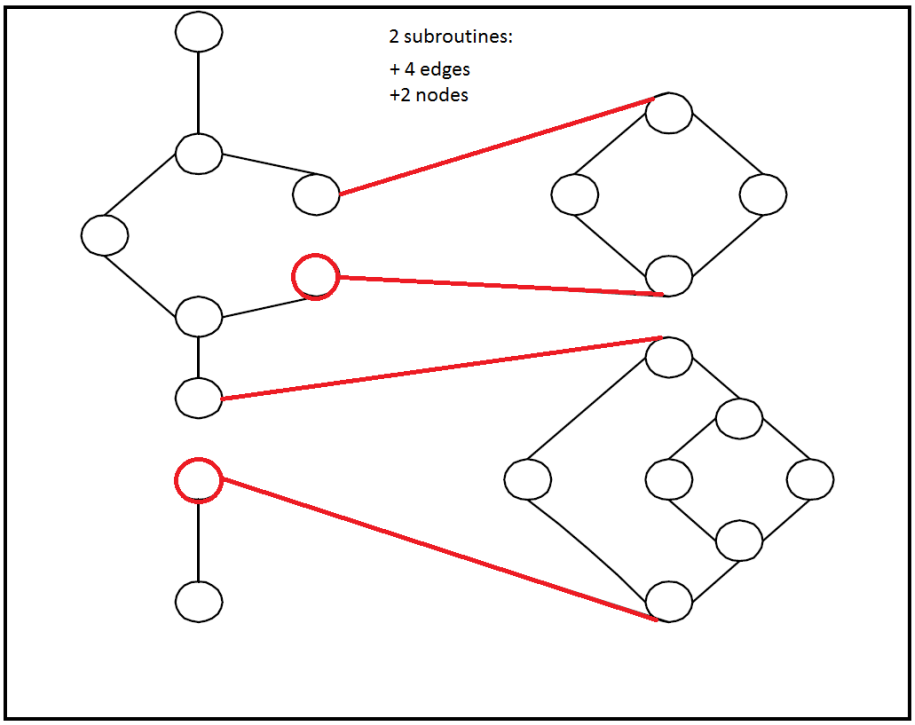



*Figure 3.11. Henderson-Sellers approach to cyclomatic complexity. Compare this graph with the one on figure 3.10*

$$CC_{HS} = \sum_{i=1}^{p}(d_i + 1) = D + 1 \tag{3.13}$$

It is important to emphasize that modularization has no effect on the Henderson-Sellers cyclomatic complexity (see figure 3.11). As the number of components ($p$) is reduced by merging them back into the caller's graph (eq. 3.12), the number of nodes $N_{node}$ is also reduced by the same amount (figure 3.11). This is not the case with the original McCabe's approach. Also notice that, since there is only a single control flow graph, Henderson-Sellers cyclomatic complexity can be easily calculated by counting the number of decisions in all subroutines (see eq. 3.13). For these reasons, in our approach, we use Henderson-Sellers's approach as a base for calculating cyclomatic complexity.

### Handling multiple synchronous calls of the same subroutine

The standard Henderson-Seller's approach (as well as the original McCabe's approach) ignores multiple synchronous calls of the same subroutine. The reason for this is that its standard usage is to estimate application testing effort which is not affected by the number of times a subroutine is called. Since our goal is to use cyclomatic complexity to estimate cognitive complexity or understandability, this is not good enough, because multiple calls to the same subroutine influence the program's cognitive complexity and understandability. For this reason, we modify the Henderson-Seller's approach to also take into account the number

of subroutine calls.

We modify eq. 3.12 in the following way: the expression $p-1$ should be replaced with the total number of subroutine calls $N_{call}$, because cyclomatic complexity will be increased by 1 for each subroutine call, and not for each subroutine definition (see figure 3.12). This leads to the following equation:

$$CC_{allBodies} = N_{edge} - N_{node} + 1 + N_{call} + 1 = N_{edge} - N_{node} + N_{call} + 2 \tag{3.14}$$

Eq. 3.14 assumes the existence of a CFG and is not suitable for practical uses. For this reason, we have to adapt eq. 3.13 as well. Since eq. 3.13 assumes that each subroutine (except the main one that calls all others) is called exactly once, we need to increase the cyclomatic complexity by $N_{call} - (p-1)$ (because $p-1$ of them are already included). This results with the following equation:

$$CC_{allBodies} = D + 1 + (N_{call} - (p-1)) = D + N_{call} - p + 2 \tag{3.15}$$

With this, we have replaced the original assumption that each subroutine is called *exactly once* with the assumption that a subroutine is called for each occurrence of a call expression (implying $N_{call} \geq p$ in the equation).

This approach takes into account multiple subroutine calls similarly to Shepperd's approach [39]. The difference is that Shepperd extends the original McCabe's approach, which makes the calculus somewhat more complex. In this approach we are using Henderson-Sellers approach as a base, but instead for each subroutine definition, we are incrementing the overall complexity by 1 for each subroutine *call*. This difference can be clearly seen on figure 3.12.

In case a subroutine is called only once, modularization is done only for abstraction sake and our approach is equivalent to Henderson-Sellers approach. However, if there are multiple calls of the same subroutine (in case modularization is done because of reuse), cyclomatic complexity increases, but not as much as it would increase if modularization is not done at all (in which case the calling CFG would contain as many subroutine CFGs as there are calls to the subroutine). This is different from the Henderson-Sellers approach in which cyclomatic complexity remains the same regardless of the reasons for modularization. We consider that our approach is inline with the intuition that cognitive complexity does not increase with modularization itself, but does reduce overall complexity if there are multiple calls to the same subroutine.

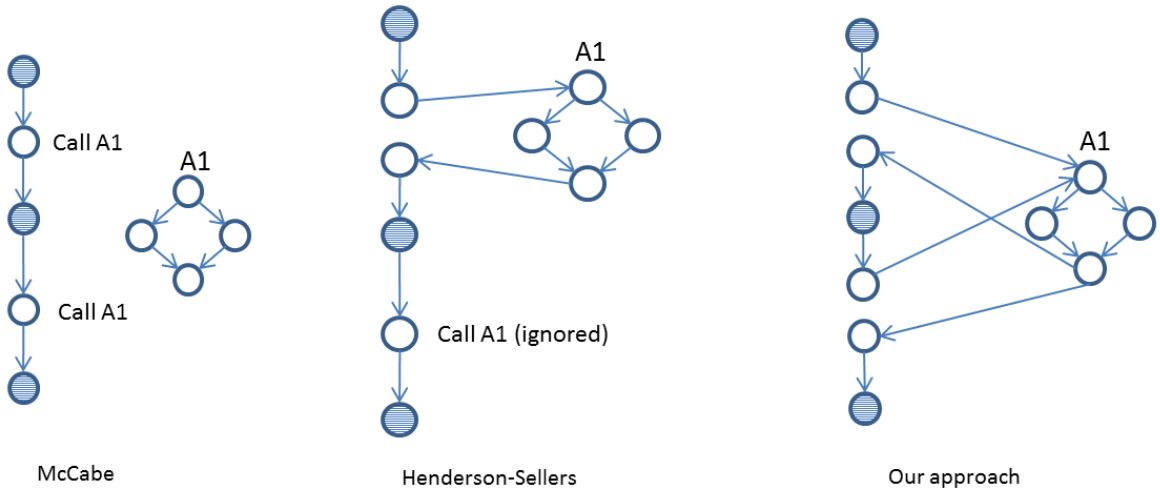



*Figure 3.12. Different approaches for handling multiple calls when calculating cyclomatic complexity.*

### Handling asynchronous calls

When dealing with component ports, we have already seen that communication can be both synchronous and asynchronous. In that case, we decided to treat asynchronous communication as synchronous communication where it is not necessary to wait for the execution control to return. This means that the effects on the CFG for those two types of communications is somewhat different (figure 3.13). Notice however that, despite the different effect to the CFG, they equally contribute to the cyclomatic complexity: synchronous calls introduce two edges, but also add one additional node, unlike asynchronous calls. Practically, this means that we can reuse eq. 3.14 (and 3.15) for both, synchronous and asynchronous calls. From understandability perspective, this is justified because both, synchronous and asynchronous calls contribute equally to cognitive complexity. Unlike many other languages, xtUML does not require modeller to explicitly synchronize access to shared resources. Asynchronous calls in xtUML are different from synchronous ones only with regard the assumption that processing immediately following asynchronous call (within the same body) must not assume that asynchronous call has already been processed.

Asynchronous calls (invocations) in OAL language are represented by the event and signal sending statements. In addition to that, upon the creation of a class instance that defines the state machine, the state machine will be started. This will be treated as the third type of asynchronous invocation. Notice that the creation of class instance that does not have a state machine will not count as an asynchronous invocation.

### Handling compound branching conditions

A single branch with a compound condition can be split into multiple branches with simple conditions. This effectively increases the cyclomatic complexity. However, the initial com-

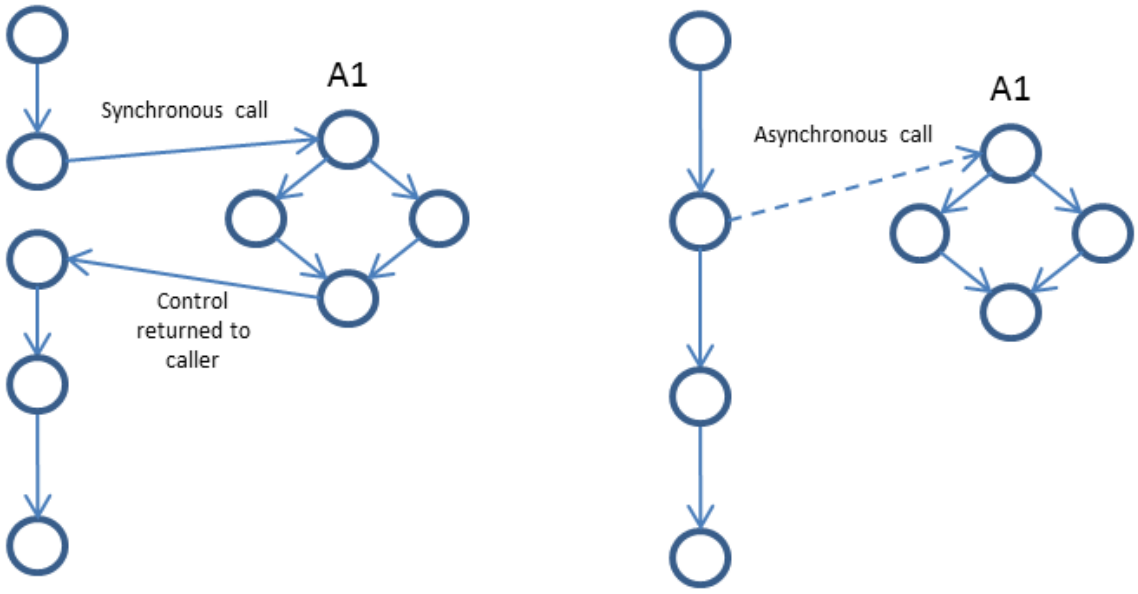



*Figure 3.13. Handling of asynchronous communication and its effect to cyclomatic complexity.*

plex condition can also be abstracted away into a single boolean flag which will have the same cyclomatic complexity as a single conditional branch. Since we are observing cyclomatic complexity from the modelling perspective, we assume the modeller will use the most abstract alternative for presenting compound conditions. Practically, this means we will use the number of decision points as $D$ in the equation for calculating cyclomatic complexity and not the number of conditions.

### 3.1.5 Calculating the overall cyclomatic complexity

A complete xtUML model is obtained by semantically integrating four different models: component, class, state machine, and processing models. So far, we have been analysing the cyclomatic complexity of each of those models separately, with minimal considerations towards other models or the xtUML model as a whole. In this section, we will describe our approach to calculating cyclomatic complexity of a complete xtUML model.

The first step in calculating cyclomatic complexity is constructing a CFG. In case of a complete xtUML model, the CFG construction starts with components, since they represent the basic building blocks of xtUML systems. Components contain classes, which contain operations and act as wrappers around state machines. For this reason, constructing a CFG graph of the whole model starts with constructing a CFG of a single xtUML component.

An incoming port message, whether it is synchronous or asynchronous, has its implementation on a port. Component ports are static constructs that do not have to be created, while most of the component functionality is distributed across class instances. This means that the main task of a port message implementation is to find the correct class instance (or

create a new one) and forward the message to it. Typically, but not mandatory, if the incoming port message is asynchronous, an asynchronous message (an event) is passed to the instance state machine. Similarly, if the incoming port message is synchronous, usually a synchronous operation on the selected instance is invoked.

In any case, a class instance represents a context (i.e. provides contextual data) used by behaviors in operations and state machines. This implies that, in order to create a CFG of an xtUML component, we can observe operations and state machines as standalone entities, without their container classes. From cyclomatic complexity perspective, which is not concerned with data complexity, classes, their instances, and the data context they provide, are not relevant. They are only used to logically organize those behaviours and they have no effect on cyclomatic complexity of an xtUML component. Practically, this means that class model cyclomatic complexity can be ignored when calculating complexity of a complete xtUML component. This is inline with the fact that cyclomatic complexity of a class model is included in the cyclomatic complexity of processing code and does not need to be explicitly taken into account.

As a base for calculating cyclomatic complexity wrapped within a component, we use eq. 3.3, where an xtUML component is considered as an MEME CFG constructed from a set of synchronously communicating bodies and asynchronously communicating state machines. Except with edges that represent asynchronous invocations (event and signal sending), the synchronous and asynchronous part of a component's CFG are interconnected with edges introduced by instance creation statements. In case a class defines a state machine, the instance creation statement starts the execution of a state machine. The total number of edges and nodes within a component in eq. 3.3 is therefore determined as a sum of all edges and nodes in all bodies and state machines wrapped in a component. While eq. 3.15 provides this number for *all* bodies within a component, eq. 3.8 gives cyclomatic complexity for a *single* state machine. This means that we need to sum the number of edges and nodes for all state machines within a component. However, notice that, when observing state machines in the context of a component, they become integrated in a single component CFG. For this reason, a virtual edge connecting each exit and each entry node of a state machine CFG is not needed. Therefore, we need to decrement the overall component cyclomatic complexity for each state machine found in the component. Applying this to eq. 3.3 results with the following equation:

$$\begin{aligned} CC_{comp} &= CC_{allBodies} + CC_{allSm} + N_{op} + N_{sig} \\ &= CC_{allBodies} + \sum_{i=1}^{N_{sm}} CC_{sm} - N_{sm} + N_{op} + N_{sig} \end{aligned} \tag{3.16}$$

where $CC_{allBodies}$ is given with eq. 3.15, $CC_{sm}$ is given with eq. 3.8, $N_{sm}$ represents the number of state machines defined in a component, $N_{op}$ and $N_{sig}$ represent the number of interface operations and signals found on all component ports. In the rest of the thesis we

use eq. 3.16 for calculating the total cyclomatic complexity of an xtUML component.

### 3.1.6 Calculating the distribution of cyclomatic complexity

Complexity in an xtUML model can be distributed: *i) horizontally*, between the elements of the same type, and *ii) vertically*, between different types of models. An example of *horizontal* complexity distribution is determining how complexity is distributed across different components of the system, or among different classes within a single component. *Vertical* complexity, on the other hand, compares complexities exposed on component, class, state machine, and processing model levels. This separation between *vertical* and *horizontal* complexity is specific for model-driven technologies, and does not have an equivalent in traditional software development metrics.

In our approach, for calculating *vertical* distribution of cyclomatic complexity, we use cyclomatic complexities of each model relative to the complexity of the complete xtUML model. Remember that class and component model complexity only visualize a subset of procedural model complexity and that they do not influence the total xtUML model cyclomatic complexity. There is, however, a value in analysing the degree of visually exposed complexity, so we include it in the process. Since cyclomatic complexity depends on the number of execution (control) paths, we expect that processing model cyclomatic complexity will dominate over cyclomatic complexities of other layers. In addition, we can also compare the cyclomatic complexity expressed through graphical models (components, classes, and state machines) with the complexity of the textual model expressed in the processing model.

*Horizontal* complexity can be calculated on several layers: on component level, among different components of an application, on class level, among the classes of a component and also on a body level, among the all bodies within the application (figure 3.14). Theoretically, complexity distribution can also be done on the state machine level, but information about distribution of state machine complexity is included in horizontal distribution on class level. As it has limited added value, horizontal distribution on state-machine level will not be considered in this dissertation. On system level, horizontal complexity distribution can be calculated between different components of the system, including all the complexities of all state machines and bodies within a component. This provides information about the distribution of cyclomatic complexity among different components. Inside a single component, complexity distribution can be analyzed across different classes, by taking into account the complexity of their state machines and bodies. Although component complexity may also be located outside its classes (ports, functions, bridges), the majority of cyclomatic complexity will be wrapped within classes. The distribution of complexity among classes will provide useful information about key classes within the component and their relative complexity. The contribution of a single class to the overall cyclomatic complexity (given with eq. 3.16)

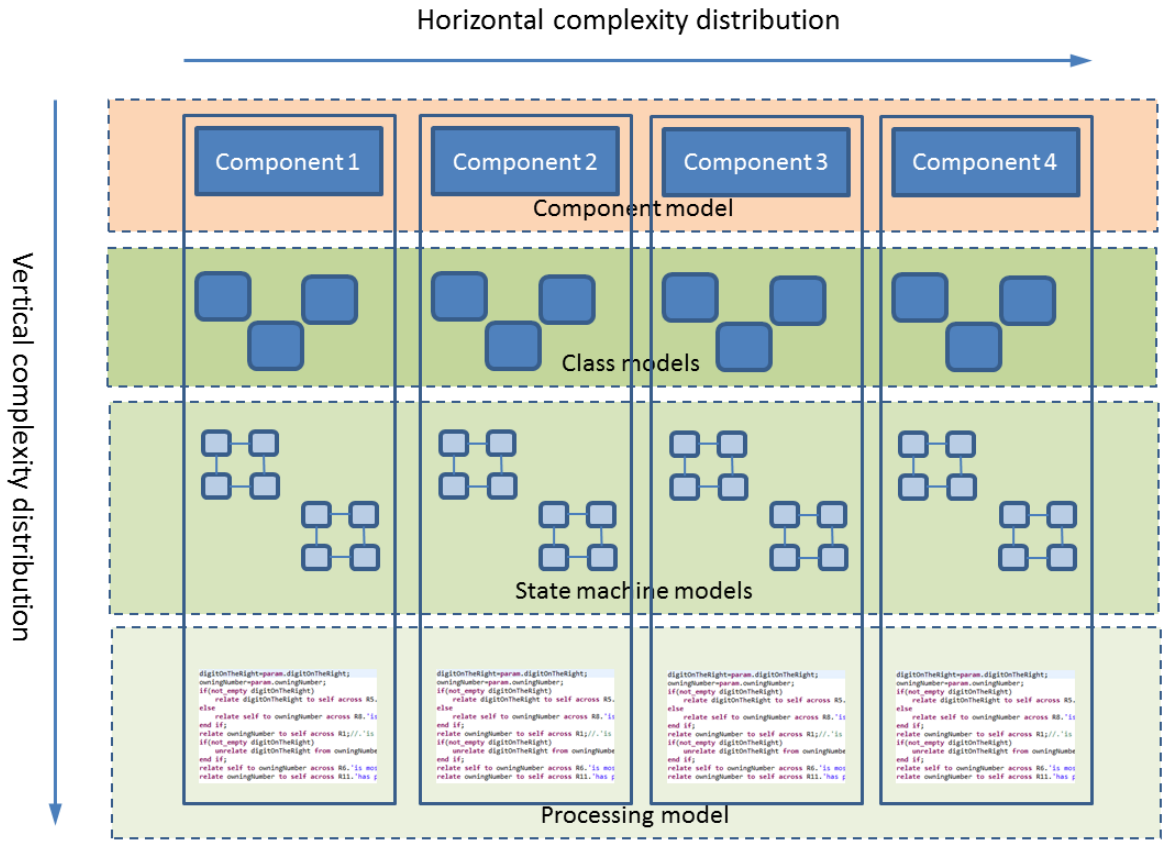



*Figure 3.14. Horizontal and vertical complexity distribution.*

can be calculated using the following equation:

$$\begin{aligned} CC(c) &= CC_{allBodies}(c) + [CC_{sm}(c) - 1] \\ &= D(c) + N_{call}(c) - p(c) + [CC_{sm}(c) - 1] \end{aligned} \tag{3.17}$$

where $D(c)$ is the total number of decisions in all bodies of class $c$, $N_{call}(c)$ the total number of synchronous and asynchronous calls in all bodies contained in the class, $p(c)$ the number of bodies defined in the class and the $CC_{sm}(c)$ is the cyclomatic complexity of an instance state machine. If a class does not define a state machine, the eq. 3.17 becomes:

$$CC(c) = CC_{allBodies}(c) = D(c) + N_{call}(c) - p(c) \tag{3.18}$$

On the lowest level, we will analyse the distribution of processing complexity across different bodies, taking into account the number of decisions and calls contained within each body ($b$).

$$CC(b) = D(b) + N_{call}(b) - p(b) = D(b) + N_{call}(b) - 1 \tag{3.19}$$

It is important to stress that equations 3.17, 3.18 and 3.19 represent mathematical contribution to the overall cyclomatic complexity (given with eq. 3.16), not the actual cyclomatic complexity of the respective class or body. The key difference is in interpretation of the

$p$ value, which **should** be interpreted as the number of *different* subroutines *called* from observed body (or class). Such interpretation prevents expression $N_{call} - p$ to be negative because there is at least as many calls as there is calls to different bodies. However, when applied to a single body, that interpretation is not applicable because the same subroutine may be called from multiple bodies, but its effect to the value of $p$ can only be taken into account once. Typically, it is added to the value of $p$ of the body in which the first call to the subroutine is found. This makes contribution of a body dependant on the order in which we process the bodies, which is not acceptable. Mathematical contribution to overall cyclomatic complexity, as described in equations 3.17, 3.18 and 3.19, does not have such a problem. The $p$ value in those equations is interpreted as the number of body *definitions* found in the observed class or body. With such interpretation, the expression $N_{call} - p$ may even have the negative value if a class contains more bodies than calls (invocations) within those bodies. For a single body, as indicated in eq. 3.19, the value of $p$ is always 1 because the body itself represents a single body definition.

## 3.2 Entropy as a measure of xtUML component model complexity

The main idea behind entropy as a measure of model complexity is that a model is observed as an information source. In that case, the entropy of a model can be calculated using classical Shannon's formula for entropy[45]. This approach has been used by several authors for calculating complexity of software on source code level[43] [44]. The key issue when calculating the entropy of an information source is symbol selection. Harisson [43] used source code *operators* as symbols. He noted that *program complexity is inversely proportional to the average information content of its operators*. This implies that, a larger number of source code symbols leads to higher software complexity. Kim et. al.[44] had a somewhat different approach which could be applied only to object-oriented source code. They constructed intra- and inter- class dependency graphs with (data and function) class members as nodes and read/write relations as arcs between nodes. The symbols used in entropy calculations are nodes in the graph.

In this thesis, we will use a similar idea of a model as an information source and will use entropy to calculate model complexity. Similarly as Harrison [43], we will use model elements as our symbols and frequency of their appearance in the model to calculate their probability and entropy. However, a precise definition of what will be considered as a *model element* is not trivial and needs elaboration.

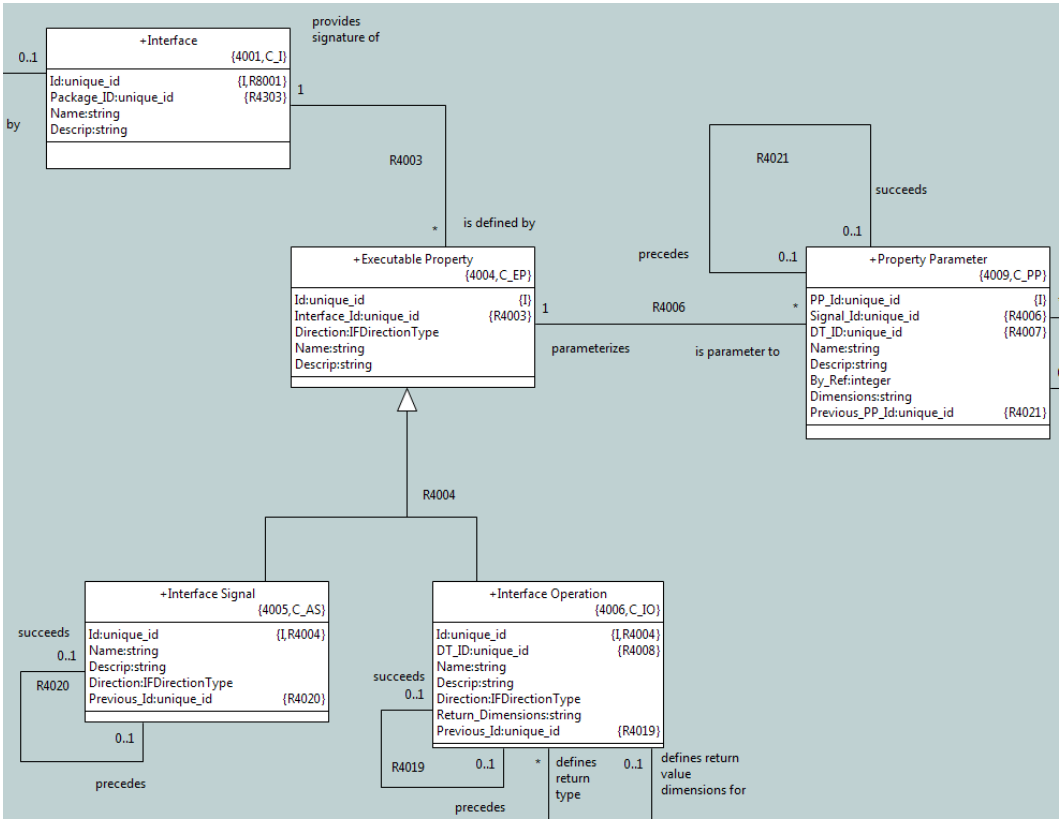



*Figure 3.15. A part of Bridgepoint xtUML meta-model describing interfaces.*

### 3.2.1 Model elements

Each model can be observed as a populated meta-model. Similarly to the *Abstract Syntax Tree* (AST) in programming languages, a meta-model defines the structure and rules for creating valid models. Meta-models are typically specified using class models. Simply put, we are using a class model to describe a class model itself (as well as the other xtUML sub-models). The classes that describe the xtUML language itself are called *meta-classes*. In our calculus, each meta-class will represent a model *element type* and each meta-class *instance* will be a model element[1].

In order to calculate probability of each model element type, we will use the number of instances of the element type in a given model. The probability of an element type is then used in order to calculate its entropy.

For example, when we create an interface in our model, an instance of meta-class *Interface* is created behind the scenes. Similarly, by adding a new interface message to that interface, we are actually creating a new instance of an *Executable Property* meta-class and associate it to the corresponding instance of *Interface* class (see figure 3.15). An *Executable Property* can be either an *Interface Signal* or an *Interface Operation* and may have many *Property Parameters*. It is important to stress that the populated xtUML

[1] In remaining of this chapter, phrases *meta-class* and *model element type* will be used interchangeably. The same applies for phrases *meta-class instance* and *model element*.

meta-model **completely** describes all structural and behavioural details of an xtUML model. Complete xtUML meta-model itself is therefore a very large class model consisting of 435 (meta)classes and 847 (meta)relations distributed across a number of packages[2].

From perspective of a BridgePoint user, an xtUML application model is a set of interlinked text snippets and figures. However, behind the scenes, each xtUML application model is actually a tree of xtUML meta-model instances or *populated meta-model*. If more information about meta-models is required please refer to [53].

When calculating entropy of an xtUML model, it is required to know the probability of all model element types (meta-classes). The probability of each model element type (meta-class) is calculated as a ratio between the number of elements of given type (the number of instances of given meta-class) and the number of all elements in the model (the number of instances of all meta-classes). Complete model entropy is then given with:

$$CE = \sum_{i=1}^{M} p_i log_2 \frac{1}{p_i} = \sum_{i=1}^{M} \frac{N_i}{N_T} log_2 \frac{N_T}{N_i} \tag{3.20}$$

where $N_i$ represents the number of elements of $i$-th type, $M$ represents the number of element types in the meta-model and $N_T$ represents the total number of elements in the model.

### 3.2.2 Vertical distribution of entropy

As an input to calculation of vertical distribution of entropy across model layers, we will use distribution of model elements across those layers. First, we will categorize all element types (meta-classes) by the layer they belong to. For example, class attributes and operations are part of class model so we consider all attributes and operations (together with other elements) as part of *CLASS* layer. Notice that not all model element types can be assigned to a model layer because there are elements that can appear in all layers or that do not belong to any of the layers. All such elements are categorized to category *OTHER*. Since we do not introduce any new symbols, the total model entropy of vertically distributed model ($CE_{vd}$), remains the same as the total model entropy ($CE$, given with eq. 3.20):

$$CE = CE_{vd} = \sum_{j=1}^{L} CE_j = \sum_{j=1}^{L} \sum_{i=1}^{M_j} \frac{N_i}{N_T} log_2 \frac{N_T}{N_i} \tag{3.21}$$

where $CE_j$ represents the sum of entropies of all elements from layer $j$, $M_j$ represents the number of meta-classes associated to $j$-th layer, $N_i$ represents the number of instances of $i$-th meta-class in $j$-th layer and $N_T$ the total number of elements in the model. In this equation, an layer entropy is calculated as a sum of entropies of all element types in that layer.

[2]For more details please refer to BridgePoint tool meta-model [52] which can be accessed from the welcome screen in the tool. Since the BridgePoint is currently the only xtUML tool, BridgePoint meta-model is actually xtUML meta-model.

Notice the difference between this approach and the case in which we calculate the entropy of a layer by counting the elements in each layer and using it in order to calculate probability that an element belongs to a layer:

$$CE_{vd} \neq CE'_{vd} = \sum_{j=1}^{L} \frac{N_j}{N_T} log_2 \frac{N_T}{N_j} \tag{3.22}$$

where $N_j$ represents the number of elements in $j$-th layer anf $L$ represents the number of layers. In that case we have only 5 symbols corresponding to 5 different layers of the xtUML model. As we decided to use meta-classes as symbols, we will ignore this approach and only use eq. 3.21 when calculating the entropy of a layer.

### 3.2.3 Horizontal distribution of entropy across classes

In order to distribute model elements horizontally, across classes, we need to associate each element with the (application) class it belongs to. This is done in addition to base distribution by the element type (meta-class). Notice that only a subset of model element types in the meta-model can be associated to a class. For example, an attribute, an operation, a state, a transition, a statement, an expression or a variable may belong to a class, but a component, a port, a package or an interface cannot. Although such distribution does not include all model elements, it is still interesting because classes are considered as the most important xtUML elements and contain the majority of model elements[3] (see figure 3.16).

Similarly as we did when calculating entropy of an xtUML layer, when calculating the entropy of a class, we will sum up the entropies of all the element belonging to that class:

$$CE_{hdc} = \sum_{k=1}^{C} CE_k = \sum_{k=1}^{C} \sum_{i=1}^{M_c} \frac{N_i}{N_T} log_2 \frac{N_T}{N_i} \tag{3.23}$$

where $CE_k$ represents the sum of entropies of all elements within $k$-th application class, $C$ represents the number of application classes, $N_i$ represents the number of instances of $i$-th meta-class, and $M_c$ represents the number of meta-classes that can be associated with a class. Notice that $N_T$ represents the total number of instances of all meta-classes, not only those that can be associated to classes.

Unlike vertical distribution, horizontal distribution introduces new symbols. The reason for this is the fact that, in order to calculate horizontal entropy distribution, we need to split the population of model elements of the same type according to the (application) class they belong. Although vertical distribution includes a larger number of elements (actually all model elements), the number of symbols is larger when we observe only the elements that can be associated with a class (distributed across classes). For example, when calculating vertical distribution, we count all *create* statements in the model and consider *all* of them to

[3] When it comes to distribution of complexity and application logic, xtUML design rules favor classes over other elements.

represent the same symbol. However, when calculating horizontal distribution, we are interested how those statements are distributed across classes. We consider *create* statements in the *Calculator* class as a separate symbol from those in the *Number* class. In that way, the number of symbols increases by factor of $C$ which represents the number of application classes. Since the total entropy depends on the number of symbols, the total amount of entropy of elements within classes will therefore be larger then the the total amount of vertically distributed entropy of all elements within a model.

### 3.2.4 Horizontal distribution of entropy across bodies

Similar approach is used to calculate entropy distribution across bodies. A body is a model element that contains processing statements. Each processing model statement must be part of exactly one body. In that sense, a body is very similar to a subroutine (procedure or a method). A body, with all its statements, can be associated to other model elements, practically holding the part of processing model associated with that element. For example, class operations and states are not directly associated with their processing statements but have their own instance of body which holds them.

Instead of splitting a population of elements of a certain type across application classes, we will split it according to the bodies they belong to (in addition to the basic distribution according to the model element type). Notice that only a relatively small subset of element types within meta-model may be associated with a body. For example, a statement, an expression or a variable can belong to a body, but a class, a relation, a port or a component cannot. Although a small subset of element *types* is included in this distribution, it is still interesting because the majority of model *elements* are located in the bodies (see figure 3.16).

Body entropy is then calculated as a sum of entropies of all the element types in that body:

$$CE_{hdb} = \sum_{n=1}^{B} CE_n = \sum_{n=1}^{B} \sum_{i=1}^{M_b} \frac{N_i}{N_T} log_2 \frac{N_T}{N_i} \tag{3.24}$$

where $CE_n$ represents the sum of entropies of all elements within $n$-th body, $B$ represents the number of bodies within the model, $N_i$ represents the number of instances of $i$-th meta-class, and $M_b$ represents the number of meta-classes that can be associated to a body. Notice that $N_T$ represents the total number of instances in the model, and not only those that can be associated to bodies.

### 3.2.5 Entropy as a complexity metric: conclusion

Using the model element types (meta-classes) defined in the meta-model as symbols and their cardinality (number of instances) for the calculation of application model entropy is a novel

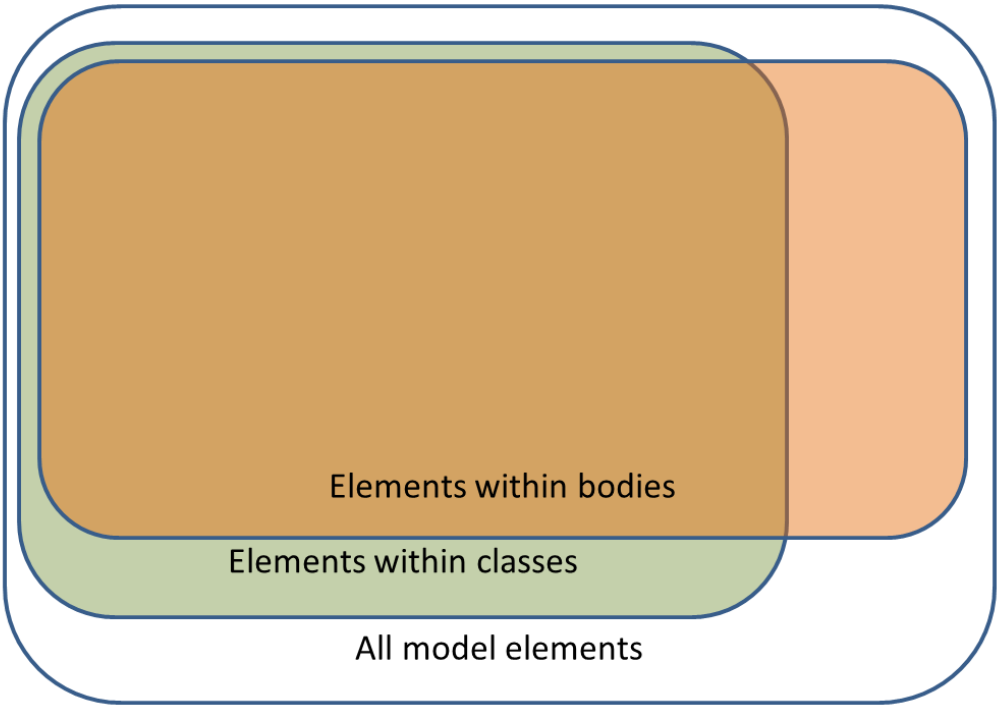



*Figure 3.16. Distribution of elements in xtUML models.*

approach presented in this thesis. Entropy as a complexity metric represents a logarithmic-scale size metric and as such has limited practical value. However, similarly as the LOC metric in the source-code level, the number of model elements (required for entropy calculus) may be well suited as the *referent size metric* for the models. This is especially important because LOC metric is poor choice as a metric in model-based software-development.

When comparing the size of two models (conforming to the same meta-model), the number of meta-class instances gives an intuitive measure of their relative complexity since it specifies the number of elements required to completely describe the application model. Similarly as comparing the LOC metric for applications written in different languages, comparing the number of elements for models conforming to different meta-models (i.e. models of different type), does not have much sense.

The only drawback of using number of meta-model instances as a referent size metric of a model is that it cannot be explicitly counted as the lines of code can. This limitation is, however, easily avoided since instance counting represents a relatively simple additional feature in the tooling that operates on the populated meta-model of the application.

## 3.3 Data complexity of xtUML models

Data complexity is a vague term that may imply several different complexity metrics. In this section, we will cover two data complexity metrics that can be applied to xtUML models. Before analysing these metrics, we will explain the data types and data modelling in xtUML.

### 3.3.1 Introduction to data types in xtUML

As in most object-oriented languages, at design time, users in xtUML can specify their own data types. Most frequently, but not exclusively, this is done in form of classes or data structures. In xtUML, classes and data structures have almost complementary usages. Classes are used to describe the concepts in a domain; they can declare relations between the instances and those relations may expose very specific meaning. It is not strange to have more than one relation between same pair of classes, expressing similar, but slightly different, semantics. At design time, classes are specific for, and visible only within, the component they are defined in. As a consequence, classes cannot be used as parameter types on interface messages. This means that, at runtime, class instances cannot be passed to some other component. This limitation in xtUML is intentional as components are intended to act as wrappers for subject matter domains which should be clearly separated. Furthermore, this limitation simplifies the language because components are independent of each other and can truly be considered as *black boxes*.

Data structures are used in a different way: typically, they are defined globally and allow data exchange between the components or with outside, non-modelled, parts of the application[4]. Unlike classes, data structures cannot be related, except using implicit containment relations when a member of the data structure is typed by some other data structure. In addition, classes may also have attributes typed by a data structure.

Except the data structures and classes, xtUML allows definition of enumerations and user-defined primitive types. Enumerations in xtUML have well-known semantics as in many other languages. User-defined primitive types are similar to *typedef* construct in the C language: they are based, and take values of, some other primitive type. There are, however, only 4 basic primitive types in xtUML: boolean, real, integer and string. Except adding another name to a primitive type, a user-defined type is often a subject of a separate translation rule. For example, by default an *integer* type in the model is translated into a *long* type in C++ language. However, a user defined type *byte* which uses integer type as its base, may be translated as a *char* type in C++. By allowing user-defined primitives, xtUML allows easier mapping to richer type systems required by target implementation languages. Memory optimization of that kind are not a concern in xtUML modelling, so, except the name, there is no other differences between the user-defined type and it's core primitive type.

### 3.3.2 Data type complexity

Complexity of a data type measures the *complexity of a data type definition* as *the number of its members and relations*. In xtUML, data modelling is usually done using a class model and our focus will be on complexity of a class definition. Although a definition of a structured

[4]For details about communication with non-modelled parts of the application, please refer to BridgePoint tool help [52]

data type formally does not belong to a class model, its data type complexity will be added to data complexity of a class when there is a class attribute typed by the data structure. A class data type complexity depends on the number of its primitive attributes, the number of relations class is involved in, as well as the number of primitive members within all its structured members. Mathematically, this is described with following recursive equation:

$$CD_t(c) = N_p + N_r + \sum_{i=1}^{N_s} CD_t(s_i) \tag{3.25}$$

where $CD_t(c)$ represents the data type complexity of a class $c$, $N_p$ represents the number of primitive members in a class, $N_r$ represents the number of relations a class is involved in, $N_s$ represents the number of structured members in a class and $CD_t(s_i)$ represents the data type complexity of $i$-th structured member in a class, also calculated using the equation 3.25. Of course, when applying the recursive formula to structured member, $N_r$ will always be zero. Notice that equation does not handle multiplicity of attributes or relations which implies that an array of integers contributes to the data type complexity equally as a plain integer member.

According to the xtUML specification, classes cannot have other classes as types of their attributes, since that implies hidden relations; relations not exposed visually in the class model. Unlike the common practice in object-oriented languages, a relation in xtUML class model is not represented with a class attribute (therefore the $N_r$ in eq. 3.25). This also implies that a relation contributes to data type complexity of each class involved in the relation.

### 3.3.3 Data flow complexity

Data flow complexity measures the complexity of the data processing performed at runtime. This typically includes the number of times data items are assigned and/or used during processing. Data flow complexity is inspired by compiler optimization techniques and is first described by Oviedo [49]. He measured the data flow complexity as the number of *def-use* (definition-usage) pairs in the code. However, an xtUML model consists of several layers and Oviedo's metric is applicable only to the processing model. In order to be able to investigate vertical data flow complexity distribution, we need to be able to apply the metric to other layers as well. We will use *the number of variable (re)definitions* as the data flow complexity metric because it can be applied to other xtUML models as well. As it ignores variable usages, our metric can be considered as simplification of Oviedo's original metric.

At runtime, from perspective of data flow complexity, an xtUML model can be observed as a set of bodies communicating synchronously or asynchronously with each other. Each body, including those of states and transitions, may have some input and output data flows which implies that entire data flow complexity of an xtUML model may be measured on a body level. In other words, data flow complexity primarily affects the processing model

which specifies details of data manipulation done at runtime. Other xtUML layers may also expose data flow complexity, but data flow complexity expressed there is taken into account on body level. This abstraction of an xtUML model is somewhat simpler from the one used when calculating cyclomatic complexity of a complete xtUML model. In order to create a control flow graph (CFG) out of a complete xtUML model, we could not eliminate the state machine layer since it describes dynamic (runtime) behaviour and introduces additional cyclomatic complexity. Unlike in cyclomatic complexity, state machine layer does not introduce any real data flow complexity that is not already expressed in the body layer. Regardless to that, it would be beneficial to see how much of the data flow complexity is visualized and how it is distributed vertically across component, class and state-machine models.

In a component model, data flow complexity will be expressed as the number of parameters on **incoming** port messages. Data flow complexity of outgoing port messages, is not ignored however; it is taken into account in the remote component in which those messages are considered as incoming. Each incoming port message (signal or operation) has associated a non-empty body that specifies processing instructions to be executed upon message reception. However, on component level, we know only the number of body parameters. Each parameter represents a *variable definition* on the entry to the body and contributes to the data flow complexity on the processing model level accordingly. A parameter in a component model only visualizes the actual parameter in the body, so it is does not contribute to the absolute data flow complexity. All parameters on all incoming messages of all ports on the component need to be taken into account[5]:

$$CD_f(comp) = \sum_{i=1}^{N_p} \sum_{j=1}^{N_{im}} N_{prm}(j) \tag{3.26}$$

where $N_p$ represents the number of ports on a component, $N_{im}$ the number of incoming messages on a given port and $N_{prm}(j)$ the number of parameters on $j$-th message.

Similarly to incoming port operations, class operations have non empty bodies so their parameters represent a *variable definitions* in the body. Class model contribution to the data flow complexity will also be ignored when calculating the total data flow complexity of the models as it is taken into account in the data flow complexity of the processing model. We will consider that each parameter of each operation in each class *visualizes* one single variable definition from the body layer. For a class $c$ we than have:

$$CD_f(c) = \sum_{j=1}^{N_{op}} N_{prm}(j) \tag{3.27}$$

where $N_{op}$ is the number of operations in a class and $N_{prm}(j)$ is the number of parameters in $j$-th operation of the class. When calculating vertical distribution of data flow complexity

[5]Remember that even outgoing ports may have incoming messages as the port direction does not imply the messages direction

all classes within a component should be taken into account.

On state machine level, events are main carriers of data so data flow complexity is defined with the number of the *event data items*. They represent parameters of the event that triggered a transition or state entry and have the same semantics as parameters of class operations or port messages. Therefore we can say that each event data item within a **non-empty** state or transition body, counts as a single variable definition. However, it is perfectly normal that an event does not have any data items, in which case, the processing of an event does not imply any variable definition on the body level. Similarly as in class operation and incoming port messages, each event data item on each transition or state with non-empty body will not count as separate variable definition but will rather be considered to visualize the same variable definition from the body layer.

State machines in xtUML allow both, transitions and states, to define processing instructions (have non-empty bodies), but they are often empty. Non-empty transitions and states are, on state machine model, clearly indicated (see figure 3.17). This means that, unlike in class operations and incoming port messages which almost exclusively have non-empty bodies, we cannot assume the existence of a body for transitions or states. When counting variable definitions (event data items) visualized by the state machine model, we should include only those transitions and states that actually have non-empty bodies. Although they share many similarities regarding the data flow complexity, counting the number of event data items in transition and state bodies is not identical. In xtUML state machines, events are assigned to transitions so, within a transition body, we always know exactly which event triggered the transition and which event data items (if any) we have on our disposal. However, this is not always true for state bodies. Each state may have more than one incoming transition, which may be triggered by different events. As a consequence, within a state body, we cannot be sure which event actually led to state body execution. This implies that, for states having multiple incoming transitions triggered by different events, only a subset of event data items common to all incoming events will be available in the state body. If events do not have common data items, there will be no data items reachable in a state body. For example, on figure 3.17, only *amount* parameter is reachable within *Producing* state because it is the only event data item common to both events. This should be taken into account when calculating data complexity of a state (body). For a state machine *sm* we then have:

$$CD_f(sm) = \sum_{i=1}^{N_{Tneb}} N_{edi}(i) + \sum_{j=1}^{N_{Sneb}} N_{cedi}(j) \tag{3.28}$$

where $N_{Tneb}$ represents the number of transitions with non-empty bodies, $N_{edi}(i)$ represents the number of event data items available in the body of $i$-th transition, $N_{Sneb}$ represents the number of non-empty states within a state machine and $N_{cedi}(j)$ is the number of event data items *common* to all transitions incoming to the $j$-th state. When calculating the vertical distribution of data flow complexity, all state-machines within all classes in a component

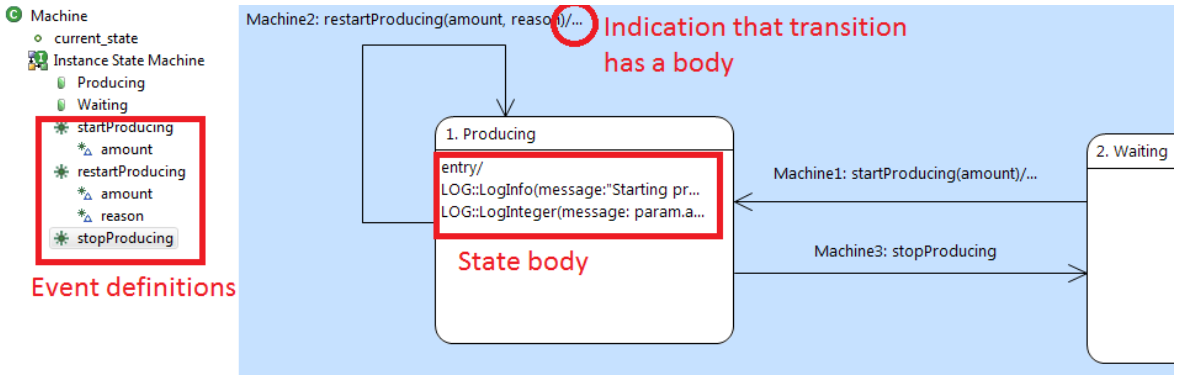



*Figure 3.17. An example of a state with multiple incoming transitions triggered by different events.*

should be taken into account.

As already mentioned, data flow complexity of a body layer in an xtUML model represents the complete data flow complexity of the model. In order to measure it we will count the number of (re)definitions in each body. In OAL, the language used to textually specify the processing model, there are four ways to (re)define a value within a body of OAL code:

- **Instance creation statement** creates a new instance of a given class with all its primitive members initialized to default values. Typically, a reference to the instance is stored in a variable.
- **Assignment statement** assigns a value to a variable or to an instance member. It may be interpreted as a declaration with an initialization (if the variable of same name has not be used yet) or as a simple redefinition (if there is a variable of the same name already defined)[6]. Such definitions may have any type, primitive, data structure or class, depending on the type of the right-hand side expressions.
- **Selection statement** is used to select an instance or an instance set from the population of some class or across a chain of relations ending with that class. The result of a selection is stored in a variable of corresponding type (class instance reference or a set of class instance references).
- **Parameter definitions** in a body are implicitly defined (no formal definition of a parameter exists in a body). Parameter values are accessed using keyword *param* before the name of the parameter or event data item.

Therefore, for a non-empty body $b$, data flow complexity can be calculated using the following equation:

$$CD_f(b) = N_{ics}(b) + N_{as}(b) + N_{ss}(b) + N_{prm}(b) \tag{3.29}$$

[6]OAL language used to textually specify processing model does not allow explicit declaration and supports only implicit variable typing.

```
1 //Instance creation
2 create object instance a of A;
3
4 //selection from instances
5 select any b from instances of B;
6
7 //selection across a chain
8 select one c related by a->B[R1]->C[R2];
9
10 //Assignment statements
11 a.age = 18;
12 temp = param.name;
```

*Figure 3.18. Statements in OAL language that affect the number of definitions.*

where $N_{ics}(b)$ represents the number of instance creation statements, $N_{as}(b)$ represents the number of assignment statements, $N_{ss}(b)$ represents the number of selection statements and $N_{prm}(b)$ is the number of parameters (or event data items) available in body $b$. When calculating the total data flow complexity of processing model (also total data flow complexity) of an xtUML model, all non-empty bodies should be taken into account.

# 4 Calculating procedure of xtUML complexity metrics

In this chapter we will explain the implementation of xtUML complexity metrics. In order to understand how the software calculating metrics actually works, we need to elaborate the translation process used by the BridgePoint tool as well as the details of the meta-model relevant for this calculus.

## 4.1 BridgePoint translation process

The translation process translates a populated xtUML meta-model into textual files that represent the source code in the target implementation language. This process is performed by the translation engine, a special software which uses several inputs. The most obvious one, an xtUML application model[1], specifies the complete functionality of the application[2]. A model that enters the translation process is assumed to be tested and verified to meet all functional requirements.

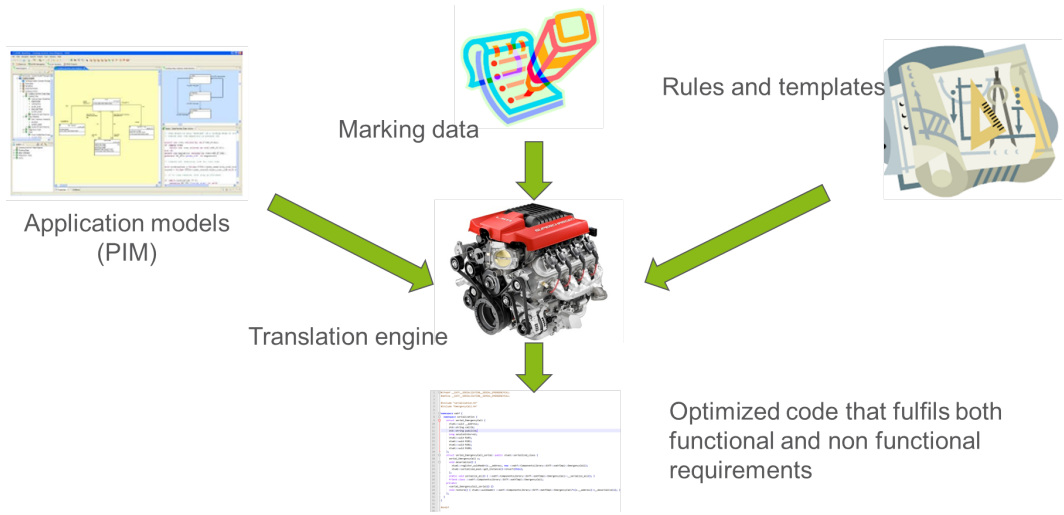



*Figure 4.1. The translation process used by the BridgePoint tool.*

The second input to the translation process is the marking data. It is used for customizing the code generation process. Using *marks*, the translation rules allow variation points in the

[1] Any model conforming to xtUML meta-model, sometimes also called *Platform Independent Model* or PIM

[2] The complete xtUML application model specifies semantics (i.e. structure and behaviour), but also the graphical information of an xtUML model. Since the translation process ignores graphical information, it is removed before for optimization reasons

code generation process. The customization of code generation process can have a global effect, such as marking for low memory footprint or they can have a very limited effect such as changing the translation rule for the type of the parameter on a component interface.

The most important input to translation process are translation rules and textual templates. Both the rules and the templates are specified using the *Rule Specification Language* (RSL)[3]. RSL follows a old structured programming paradigm (based on functional decomposition) and is syntactically very similar to the OAL processing language. The similarity between the languages is not accidental because both languages share class model navigation as their core functionality[4]. In order to support textual templates required for generation of source code files, RSL has some additional features not available in OAL. Figure 4.2 shows an example of RSL template for C++. An example of RSL queries can be seen on figure 4.3.

```
/*******************************************************************************
 *
 * O B J E C T   C L A S S   ( B O D Y )
 *
 * NAME: ${class_inst.UML_Name}
 *
 * NUMBER: ${class_inst.UML_Number}
 *
 * KEY LETTERS: ${class_inst.UML_Key_Letters}
 *
${Description}\
 *
 * Generated by EMC-C++ ${trans_inst.Version}
 *
 *******************************************************************************/

#define __EMC_CPP_BUILD ${trans_inst.Build_Number}

#include "${class_inst.CPP_Name}.${trans_inst.CPP_Header_Extension}"

#include "${trans_inst.CPP_Select_Include_File}"

using namespace xtuml;

#include "${cont_inst.CPP_Include_Into_Body}"
${Include_Externals}\
${Operations}\
${Instance_State_Actions}\
${Class_State_Actions}\
${Instance_Transition_Actions}\
${Class_Transition_Actions}\

// Constructor
${class_inst.CPP_Qualified_Name}::${class_inst.CPP_Name}()${Constructor_Initializer_List} {
${Instance_Registrations}\
}
```

*Figure 4.2. An example of RSL template file for C++ language.*

The two roles of RSL actually describe the two phases of the translation process. The first phase, *querying*, searches the populated meta-model of an application for interesting

[3]More information about RSL can be obtained in BridgePoint help[54].

[4]In case of OAL, navigation is performed on application class model, while RSL specifies navigation on the populated meta-model.

```
.//
.function find_classes_in_component
.param inst_ref comp_inst
    .select many attr_result from instances of O_OBJ where false
    .select many root_pkg_set related by comp_inst->PE_PE[R8003]->EP_PKG[R8001]
    .for each root_pkg in root_pkg_set
        .invoke cips = find_classes_recursively(root_pkg)
        .assign class_in_pkg_set = cips.result
        .assign attr_result = attr_result|class_in_pkg_set
    .end for
.end function
.//
.function find_classes_recursively
.param inst_ref pkg_inst
    .select many class_set related by pkg_inst->PE_PE[R8000]->O_OBJ[R8001]
    .select many internal_pkg_set related by pkg_inst->PE_PE[R8000]->EP_PKG[R8001]
    .for each internal_pkg in internal_pkg_set
        .invoke ns = find_classes_recursively(internal_pkg)
        .assign nested_classes = ns.result
        .assign class_set = class_set|nested_classes
    .end for
    .assign attr_result = class_set
.end function
```

*Figure 4.3. An example of RSL queries.*

information that can be extracted from the model. Depending on the generated target, this information can be, for example, the name of some interface or the number of attributes in a class. In the second phase, *generation*, the information extracted from a model is complemented with textual templates providing the actual output of the translation process. Notice that such translation process surpasses the main use case of source code generation and can also be used to generate application documentation, enforce good modelling practices or calculate model metrics.

The calculus of xtUML metrics discussed in this thesis is formalized as an RSL translation process in which the querying phase is used to obtain information required by metric's equation. After all required information is collected, a metric value is calculated and exported as pure text. For some metrics, that require statistical analysis, it was not practical to use RSL for the complete calculus. In such cases, information collected from a model is exported to a *Comma-Separated-Values* (CSV) file for further processing in other tools.

## 4.2 Implementation of xtUML cyclomatic complexity

In this section we will explain details of RSL implementation for cyclomatic complexity of xtUML models. Understanding of RSL code requires familiarity with xtUML meta-model, so relevant parts of the meta-model are provided in figures accompanying the text.

### 4.2.1 Vertical distribution of cyclomatic complexity

Implementation of vertical cyclomatic complexity goes as follows: First we calculate the cyclomatic complexity of components (see function *calculate_cc_comp_port* on figure 4.4) by counting the number of operations and signals on all ports in the component (see eq. 3.4). Starting from an instance of a component (C_C meta-class), we navigated across *Component Port* (C_PO), *Interface Reference* (C_IR), *Interface*(C_I), *Executable Property*(C_EP) and *Interface Signal* (C_AS) meta-classes across relations 4010, 4016, 4012, 4003 and 4004 respectively. The number of selected instances is obtained by the *cardinality* operator on the set. Similar navigation is done for interface operations (C_IO) meta-class. This part of meta-model can be seen on figure 4.6.

```
413 .//-------------------------------------------------------------------------------
414 .// Component complexity = Nop + Nsig
415 .//-------------------------------------------------------------------------------
416 .function calculate_cc_comp_port
417 .param inst_ref comp_inst
418     .assign attr_result = 0;
419     .select many signal_set related by comp_inst->C_PO[R4010]->C_IR[R4016]->C_I[R4012]->C_EP[R4003]->C_AS[R4004]
420     .select many operation_set related by comp_inst->C_PO[R4010]->C_IR[R4016]->C_I[R4012]->C_EP[R4003]->C_IO[R4004]
421     .assign op_num = cardinality operation_set
422     .assign sig_num = cardinality signal_set
423     .//print "Component ${comp_inst.Name} has ${op_num} operations and ${sig_num} signals."
424     .assign attr_result = (op_num + sig_num)
425     .//print "Component ${comp_inst.Name} has (port) cyclomatic complexity: ${attr_result}"
426 .end function
427 .//
```

*Figure 4.4. RSL code for calculating cyclomatic complexity of component ports.*

This was followed by calculating class cyclomatic complexity. The cyclomatic complexity of a class is calculated by counting operations of the class and adding the complexity introduced by the relations the class is involved in. First we had to find all the classes in the component (function *find_classes_in_component* on figure 4.5) and then, for each class, count the operations it has. Operations of a class are represented with instances of O_TFR meta-class and reached by selecting across relation R115 (see figure 4.5). For details about cyclomatic complexity introduced by relations please refer to function *calculate_cc_comp_class* in *CyclomaticComplexityVertical.arc* on [55].

```
505     .//Contribution of operations to complexity
506     .invoke cls = find_classes_in_component(comp_inst)
507     .assign class_set = cls.result
508     .for each class in class_set
509         .select many class_op_set related by class->O_TFR[R115]
510         .assign class_op_num = cardinality class_op_set
511         .print "Class ${class.Name} has operations ${class_op_num}"
512         .assign op_num = op_num+class_op_num
513     .end for
514     .assign attr_result = attr_result + op_num
515 .end function
516 .//
```

*Figure 4.5. Counting the number of operations in classes.*

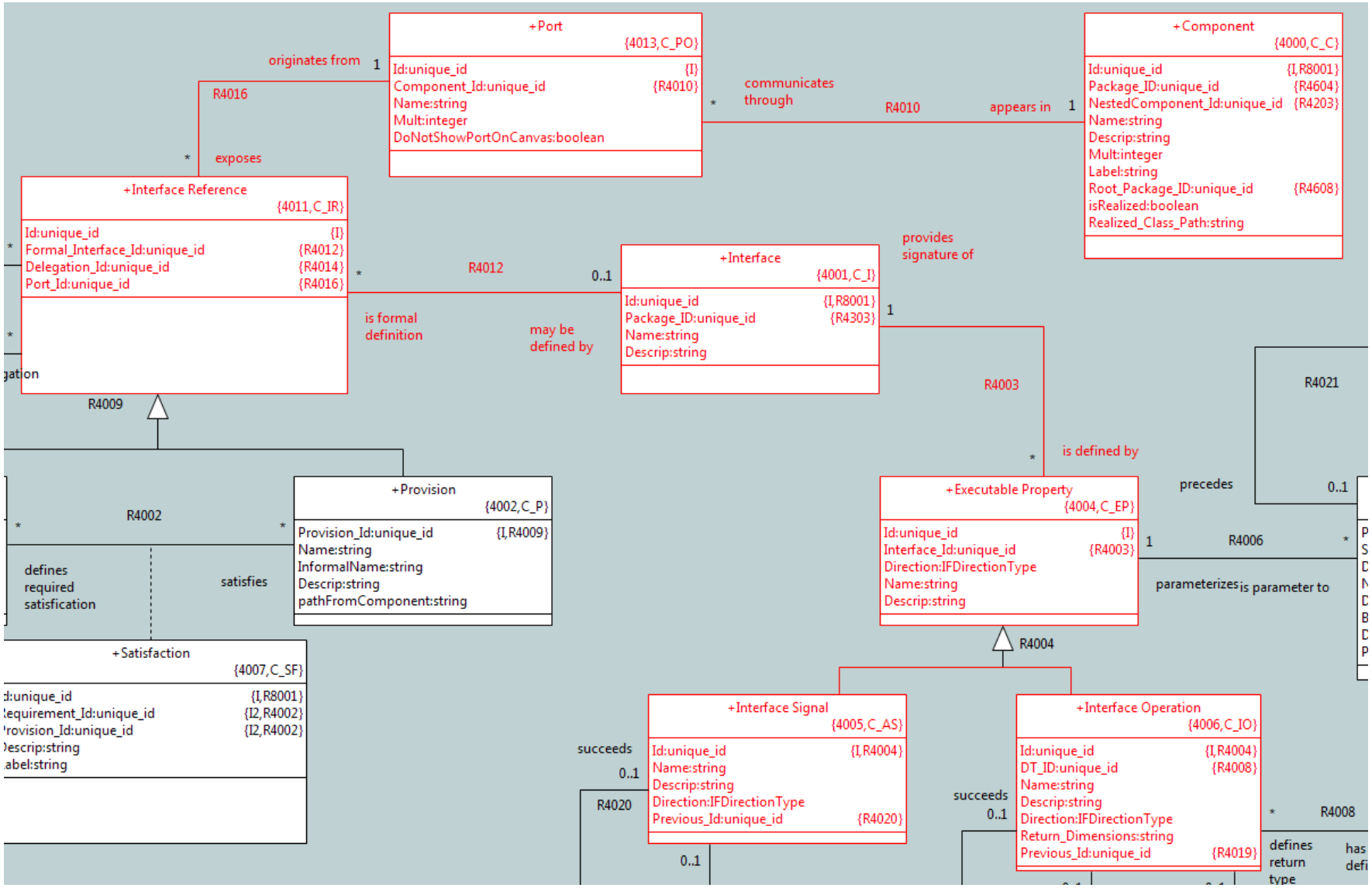



*Figure 4.6. Part of xtUML meta-model describing components.*

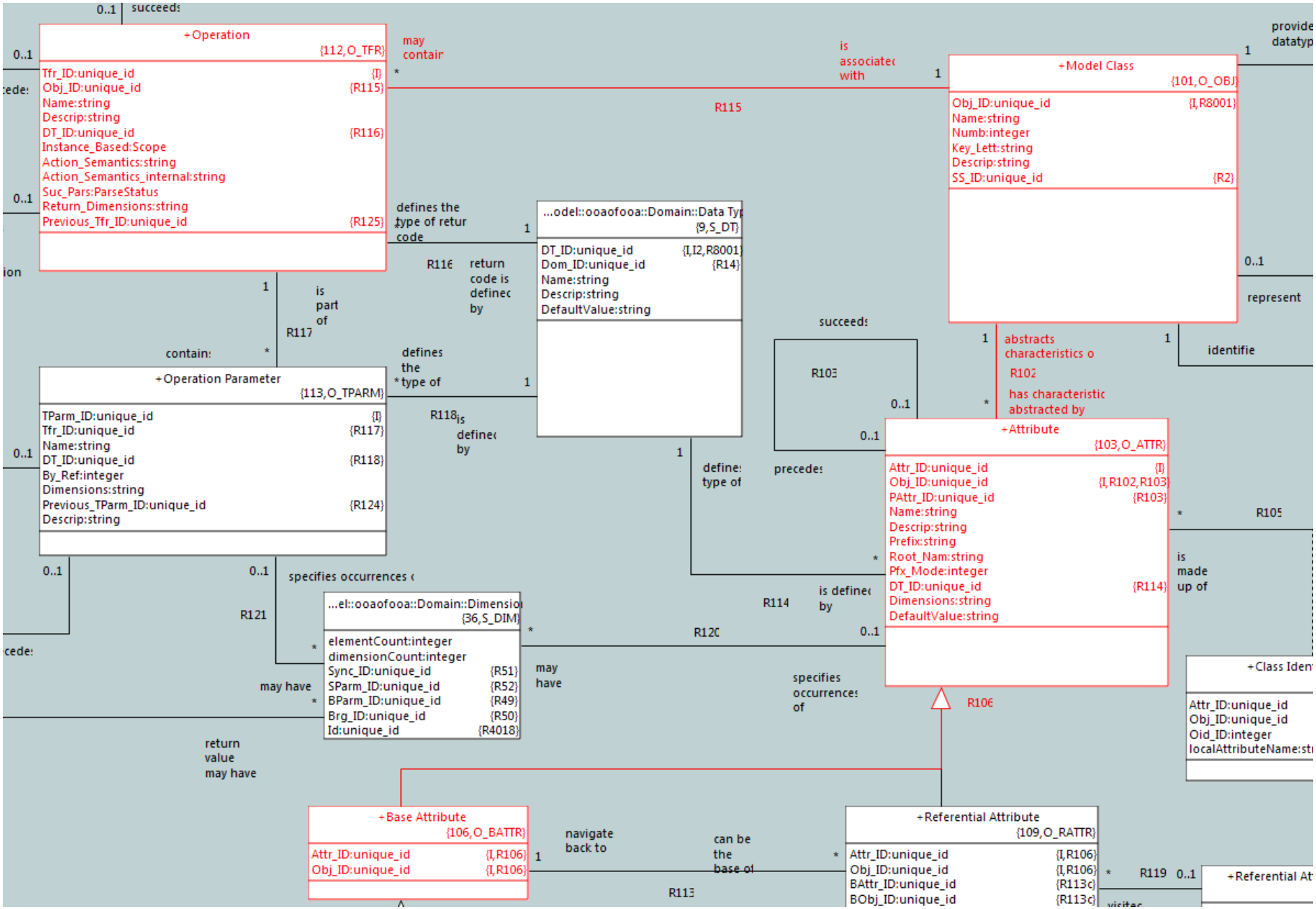



*Figure 4.7. Part of xtUML meta-model describing classes.*

As a base for calculating cyclomatic complexity of state machines we used equation 3.8. Depending on whether the constant 6 representing the infrastructure complexity is taken into account or not, we used two different functions: *calculate_cc_comp_sm_graphical* and *calculate_cc_comp_sm_complete* (see figure 4.8). Both functions first search for all classes within a component and then, for each class, call the *calculate_cc_sm* function, but with different value for the second parameter. The complete cyclomatic complexity of state machine layer is calculated as a sum of cyclomatic complexities of all classes in the component. The result that includes infrastructure complexity (the result of *calculate_cc_comp_sm_complete* function) is used to calculate the total cyclomatic complexity of a component, while the version without infrastructure (*calculate_cc_comp_sm_graphical* function) is used to calculate cyclomatic complexity visualized by the state machine layer.

```
517 .//-------------------------------------------------------------------------------
518 .// Graphical state machine cyclomatic complexity for all state machines in component
519 .// Does not include 6 that include infrastructure
520 .//-------------------------------------------------------------------------------
521 .function calculate_cc_comp_sm_graphical
522 .param inst_ref comp_inst
523     .assign attr_result = 0;
524     .//defined in in q.general.utils.arc
525     .invoke cic = find_classes_in_component(comp_inst)
526     .assign class_set = cic.result
527     .for each class in class_set
528         .invoke csc = calculate_cc_sm(class, false)
529         .assign cc_sm_class = csc.result
530         .//print "Class ${class.Name} has graphical state machine cyclomatic complexity
531         .assign attr_result = attr_result + cc_sm_class
532     .end for
533 .end function
534 .//-------------------------------------------------------------------------------
535 .// Complete state machine cyclomatic complexity for all state machines in component
536 .// Includes  cyclomatic complexity of infrastructure (6)
537 .//-------------------------------------------------------------------------------
538 .function calculate_cc_comp_sm_complete
539 .param inst_ref comp_inst
540     .assign attr_result = 0;
541     .//defined in in q.general.utils.arc
542     .invoke cic = find_classes_in_component(comp_inst)
543     .assign class_set = cic.result
544     .for each class in class_set
545         .invoke csc = calculate_cc_sm(class, true)
546         .assign cc_sm_class = csc.result
547         .//print "Class ${class.Name} has graphical state machine cyclomatic complexity
548         .assign attr_result = attr_result + cc_sm_class
549     .end for
550 .end function
```

*Figure 4.8. Two ways to calculate cyclomatic complexity of state machine layer within a component.*

Cyclomatic complexity of a state machine in a single class is calculated by selecting a set of states in a state machine. For each state (row in the state-event matrix), we then select all entries in the row (see figure 4.9, line 210) by selecting all instances of *SM_SEME* meta-class across relation R503. In a similar way (line 213), we select the *Event Ignore* entries for that state (*SM_EIGN* meta-class across relation R504). Contribution of each state to cyclomatic

```
197 .function calculate_cc_sm
198 .param inst_ref class
199 .param boolean includeSmInfrastructure
200     .assign attr_result = 0
201     .select one ism related by class->SM_ISM[R518]->SM_SM[R517]
202     .if(not_empty ism)
203         .if(includeSmInfrastructure)
204             .assign attr_result = 6
205         .end if
206         .select many state_set related by ism->SM_STATE[R501]
207         .//Contribution of each state
208         .for each state in state_set
209             .assign state_contrib = 0
210             .select many state_entry_set related by state->SM_SEME[R503]
211             .assign state_entry_num = (cardinality state_entry_set)
212             .//print "State ${state.Numb} in class ${class.Name} has ${state_entry_num} ent
213             .select many state_ei_entry_set related by state->SM_SEME[R503]->SM_EIGN[R504]
214             .assign state_ei_entry_num = (cardinality state_ei_entry_set)
215             .//print "State ${state.Numb} in class ${class.Name} has ${state_ei_entry_num}
216             .assign state_non_ei_entry_num = (state_entry_num - state_ei_entry_num)
217             .//print "State ${state.Numb} in class ${class.Name} has ${state_non_ei_entry_
218             .//If state has at most one event ignored entry, state contribution to cyclomat
219             .//complexity is equal to number of entries in the row (which is equal to numbe
220             .//of received events). Otherwise, state contributes with one more than there i
221             .//non-event-ignored entries
222             .if(state_ei_entry_num<=1)
223                 .assign state_contrib = state_entry_num
224             .else
225                 .assign state_contrib = state_non_ei_entry_num + 1
226             .end if
227             .//print "State ${state.Numb} in class ${class.Name} contributes with ${state_c
228             .assign attr_result = attr_result + state_contrib
229         .end for
230         .//Contribution of transitions: State machine has CFG that for each RTC has 2 calls
231         .//transition effect call and state entry body call. This means there are 2 calls p
232         .//Since there are as many RTCs and there is transitions. state machine, without it
233         .//implies 2Nt invocations
234         .select many transition_set related by ism->SM_TXN[R505]
235         .assign num_txn =  cardinality transition_set
236         .//print "Class ${class.Name} has state machine with ${num_txn} transitions (RTC st
237         .assign attr_result = attr_result + (2*num_txn)
238         .//print "Class ${class.Name} has state machine cyclomatic complexity: ${attr_resul
239     .else
240         .//print "Class ${class.Name} has not instance state machine."
241     .end if
242 .end function
```

*Figure 4.9. Cyclomatic complexity calculus for a single state machine.*

complexity of a state machine is equal to the number of events if there is at most one *Event Ignore* entry (line 223). Alternatively, the contribution is equal to the number of non-*Event Ignore* entries increased by one (line 225). Each transition in the state machine (selection to *SM_TXN* meta-class across R505 in line 234) contributes to cyclomatic complexity with 2.

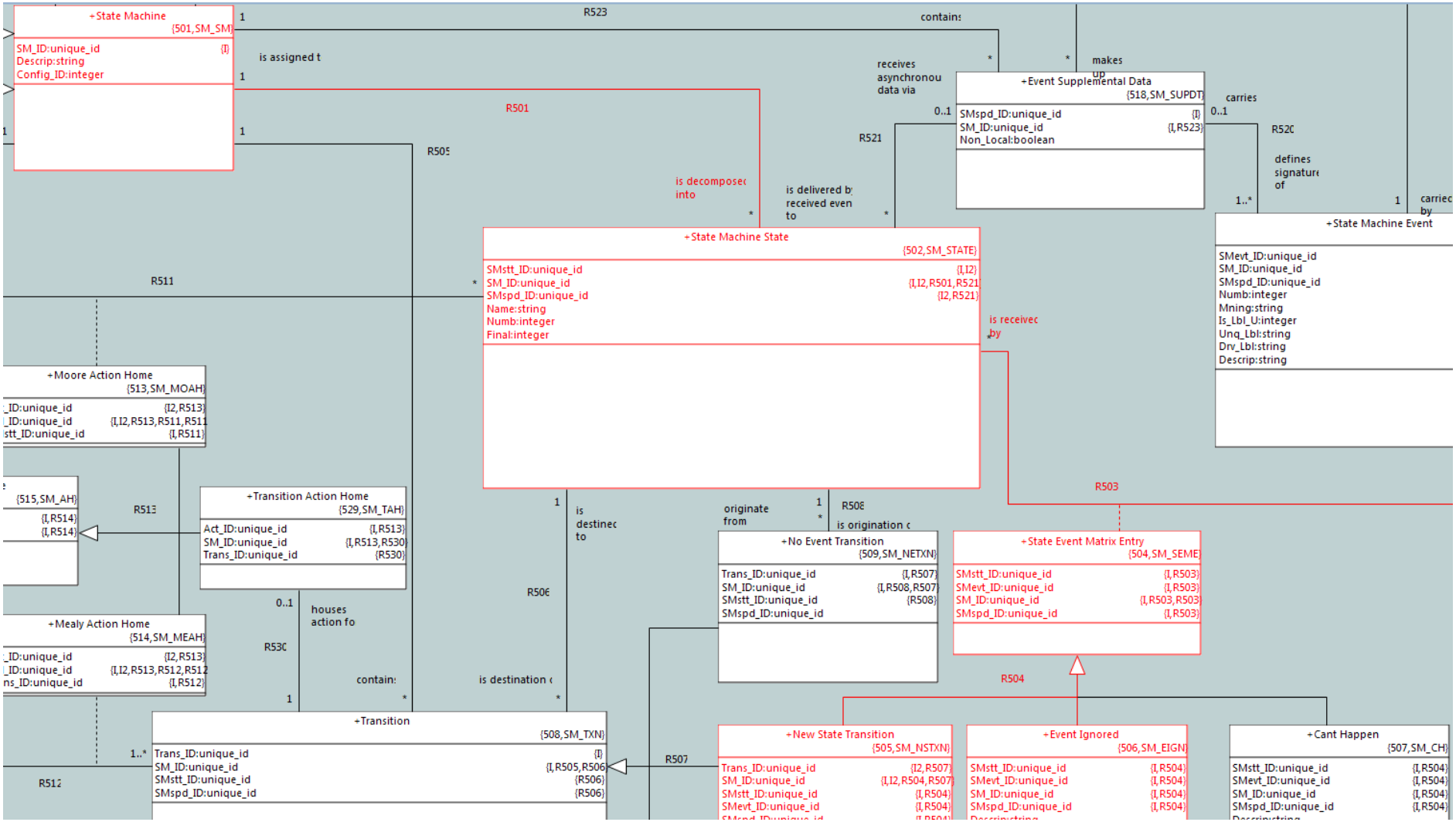



*Figure 4.10. Part of xtUML meta-model describing state-machines.*

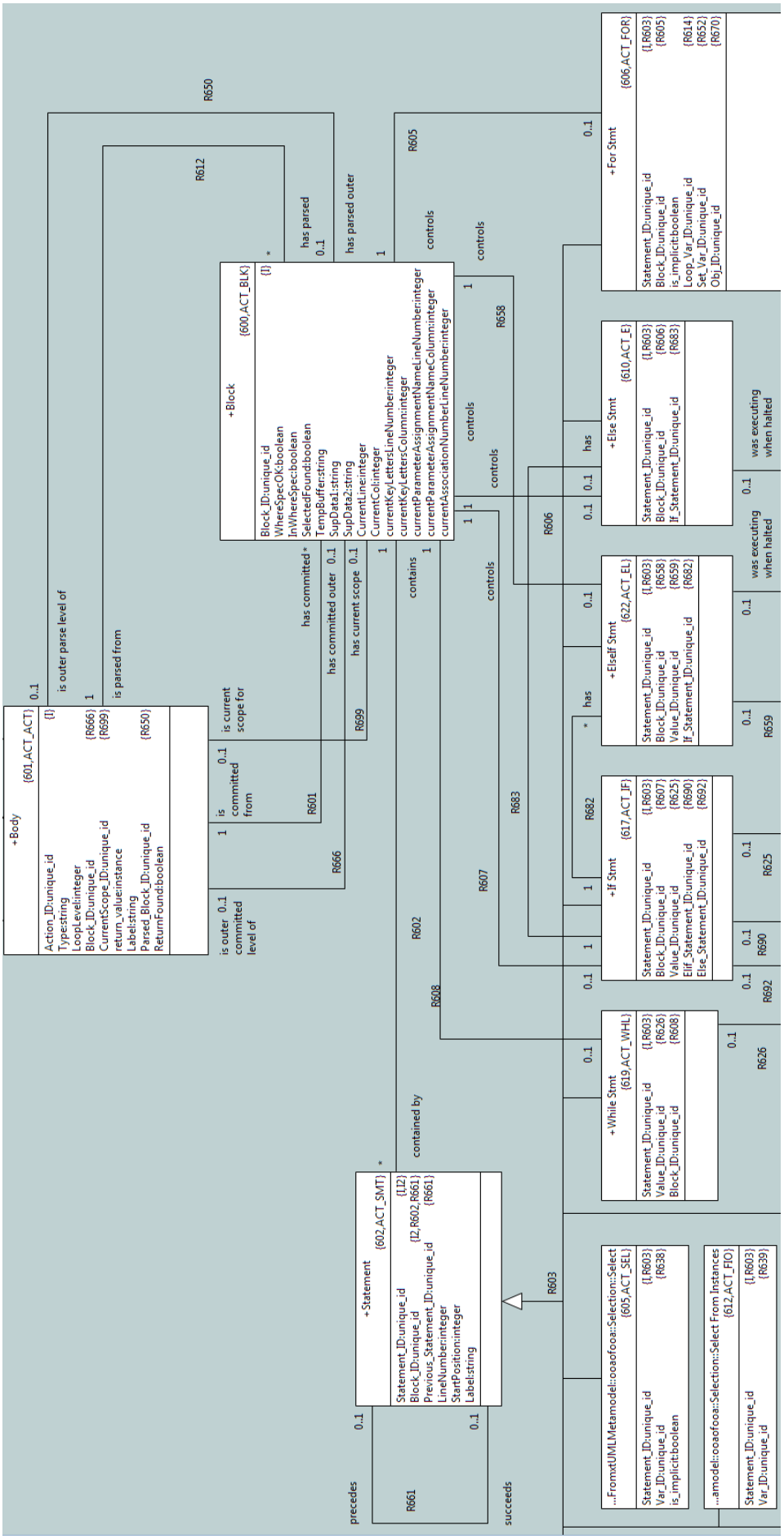



*Figure 4.11. Part of xtUML meta-model describing bodies.*

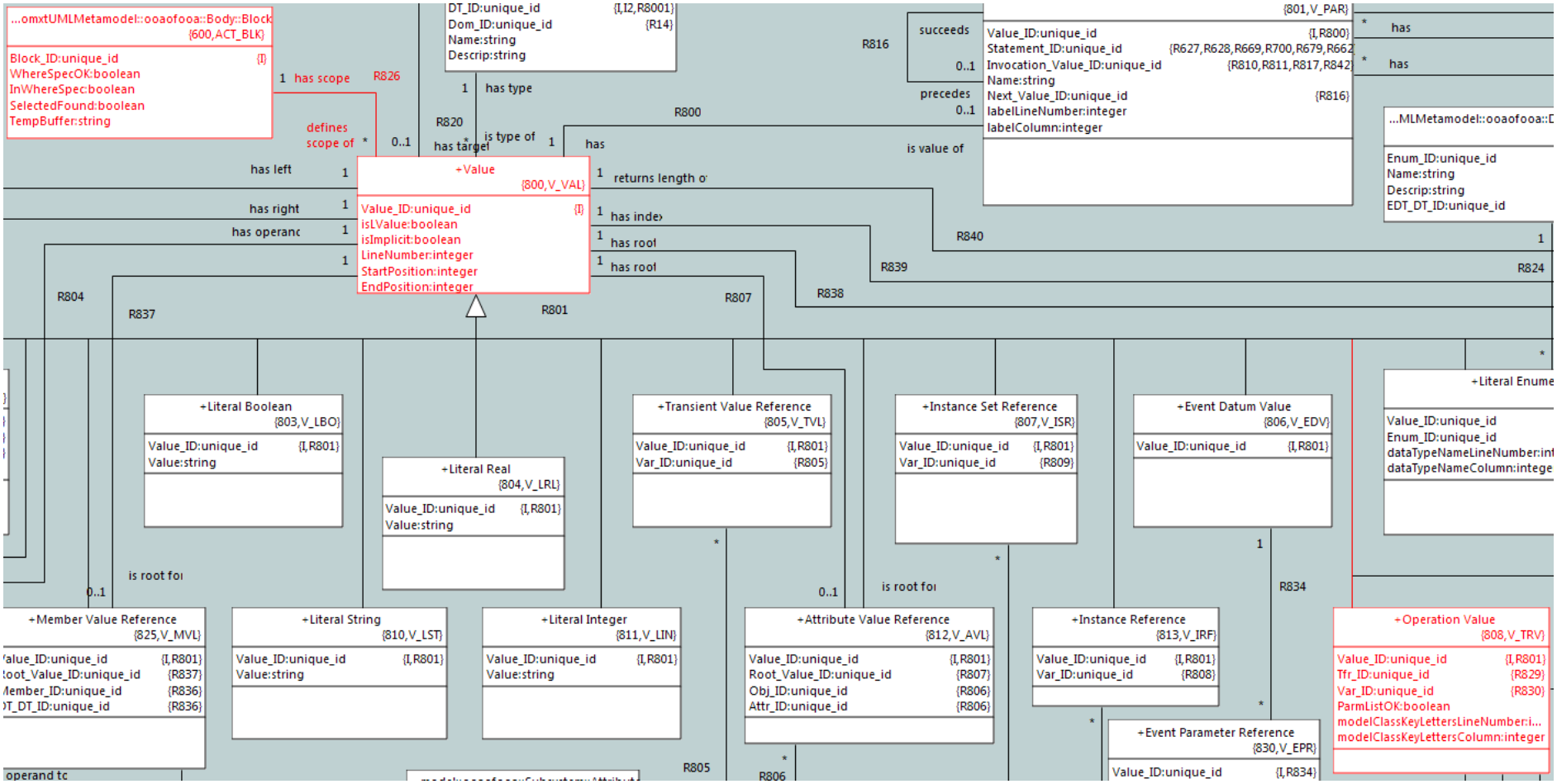



*Figure 4.12. Part of xtUML meta-model describing values (expressions).*

Equation 3.15 was starting point when calculating cyclomatic complexity of all bodies. The first step when calculating cyclomatic complexity of all bodies within a component is to find all bodies (line 558 in figure 4.13). Each statement of OAL code is presented with an instance of an *ACT_SMT* meta-class. The set of all bodies (*ACT_ACT* instances) is then filtered to keep only the bodies which actually contain OAL code. This is done by a selection starting from a set of bodies to *ACT_SMT* across *ACT_BLK* via relations R601 and R602 and back across the same chain to the *ACT_ACT* meta-class. The selection chain across an OAL statement (*ACT_ACT*) meta-class and back ensures that only bodies that contain at least one statement are selected. For each non-empty body we then count the number of decisions and calls as required by eq. 3.15 (see figure 4.14).

```
551 .//-------------------------------------------------------------------------------
552 .// Processing code cyclomatic complexity:  = D + Ncall - p + 2
553 .//-------------------------------------------------------------------------------
554 .function calculate_cc_comp_body
555 .param inst_ref comp_inst
556     .assign attr_result = 0;
557     .//defined in in q.general.utils.arc
558     .invoke bs = find_all_bodies_within_component(comp_inst)
559     .assign body_set = bs.result
560     .assign body_num = cardinality body_set
561     .//print "Component ${comp_inst.Name} has ${body_num} bodies."
562     .select many non_empty_body_set related by body_set->ACT_BLK[R601]->ACT_SMT[R602]->ACT_BLK[R602]->ACT_ACT[R601]
563     .assign non_empty_body_num = cardinality non_empty_body_set
564     .print "Component ${comp_inst.Name} has ${non_empty_body_num} non-empty bodies."
565     .for each body in non_empty_body_set
566         .invoke cdc = count_body_decisons_and_calls(body)
567         .assign decisions_and_calls= cdc.result
568         .print "Body ${body.Label} has decisions and calls:${decisions_and_calls}"
569         .assign attr_result = attr_result+decisions_and_calls
570     .end for
571     .assign attr_result = attr_result-non_empty_body_num
572     .assign attr_result = attr_result+2
573 .end function
```

*Figure 4.13. Calculating cyclomatic complexity of all bodies within a component.*

The decisions in a body are counted by counting the number of *while* (*ACT_WHL*), *if* (*ACT_IF*), *else if* (*ACT_EL*) and *for* (*ACT_FOR*) statements (lines 322, 326, 330 and 334 respectively). Figure 4.11 shows the relevant part of the meta-model. In a similar way, the number of synchronous and asynchronous invocation statements appearing in the body is taken into account. Since synchronous invocations may appear as part of expression in the statement, we also had to count the statement containing such invocation expressions. The relevant part of the meta-model is shown in figure 4.12. Relation R826 on that figure relates a body block (*ACT_BLK*) with all its expressions (*V_VAL*). Other invocation expressions, for example *Operation Value* (*V_TRV*), are subclasses of the *Value* meta-class.

After calculating cyclomatic complexity of each layer, the vertical distribution of complexity is obtained as a ratio between the cyclomatic complexity of a layer and the total cyclomatic complexity of a component (given with eq. 3.16).

```
.function count_body_decisons_and_calls
.param inst_ref body
    .assign attr_result = 0
    .assign decisons = 0
    .assign calls = 0
    .select many stmt_set related by body->ACT_BLK[R601]->ACT_SMT[R602]
    .assign stmt_num =  cardinality stmt_set
    .//print "Body of ${body.Type} ${body.label} has total of ${stmt_num} statements."
    .//Decision statements
    .select many whl_set related by body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_WHL[R603]
    .assign whl_num =  cardinality whl_set
    .assign decisons = decisons + whl_num
    .//print "Body of ${body.Type} ${body.label} has ${whl_num} while statements."
    .select many if_set related by body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_IF[R603]
    .assign if_num =  cardinality if_set
    .assign decisons = decisons + if_num
    .//print "Body of ${body.Type} ${body.label} has ${if_num} if statements."
    .select many elif_set related by body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_EL[R603]
    .assign elif_num =  cardinality elif_set
    .assign decisons = decisons + elif_num
    .//print "Body of ${body.Type} ${body.label} has ${elif_num} elif statements."
    .select many for_set related by body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_FOR[R603]
    .assign for_num =  cardinality for_set
    .assign decisons = decisons + for_num
    .//print "Body of ${body.Type} ${body.label} has ${for_num} for statements."
    .//Sync commuincations (calls)
    .select many brg_invk_set related by body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_BRG[R603]
    .assign brg_invk_num =  cardinality brg_invk_set
    .assign calls = calls + brg_invk_num
    .//print "Body of ${body.Type} ${body.label} has ${brg_invk_num} bridge invocation statements."
    .select many fnc_invk_set related by body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_FNC[R603]
    .assign fnc_invk_num =  cardinality fnc_invk_set
    .assign calls = calls + fnc_invk_num
```

*Figure 4.14. Calculating cyclomatic complexity of a single body.*

### 4.2.2 Horizontal distribution of cyclomatic complexity

Horizontal distribution of cyclomatic complexity is formalized in *CyclomaticComplexityHorizontal.arc* file and can be found on [55]. Unlike vertical distribution, horizontal distribution generates a *Comma-Separated-Values* (*CSV*) files that are later used by Microsoft Excel for further statistical analysis. Horizontal distribution is done on two layers, across classes and bodies which corresponds to *generate_cc_comp_class* and *generate_cc_comp_body_decisions_and_calls* functions in the figure 4.15.

```
    .select any comp_inst from instances of C_C where selected.Name==comp_name
    .if(not_empty comp_inst)
        .//-------------------------------------------------------------------------------
        .//Cyclomatic complexity: horizontal distribution
        .print "================================================================================"
        .print "Generating class cyclomatic complexity for horizontal distribution:"
        .invoke t1 = generate_cc_comp_class(comp_inst, project_location)
        .assign Lines = t1.body
        .include "${project_location}/utils/t.part.general"
        .emit to file "C:/temp/${comp_inst.Name}_class.csv"
        .print "================================================================================"
        .print "Generating component body cyclomatic complexity for horizontal distribution:"
        .invoke t2 = generate_cc_comp_body_decisions_and_calls(comp_inst, project_location)
        .assign Lines = t2.body
        .include "${project_location}/utils/t.part.general"
        .emit to file "C:/temp/${comp_inst.Name}_bodies.csv"
    .else
```

*Figure 4.15. Generating CSV files needed for horizontal distribution across classes and bodies.*

```
429 .function generate_cc_comp_class
430 .param inst_ref comp_inst
431 .param string project_loc
432     .invoke cic = find_classes_in_component(comp_inst)
433     .assign class_set = cic.result
434     .assign cc_all_classes = 0
435     .assign class_number = 0
436     .for each class in class_set
437         .assign class_number = class_number + 1
438         .invoke ccc = calculate_cc_class_complete(class)
439         .assign cc_class = ccc.result
440         .assign key = "${class.Name}"
441         .assign value = "${cc_class}"
442 ${key},${value}
443         .assign cc_all_classes = cc_all_classes + cc_class
444     .end for
445 .end function
446 .//
447 .function generate_cc_comp_body_decisions_and_calls
448 .param inst_ref comp_inst
449 .param string project_loc
450     .assign attr_result = 0;
451     .invoke bs = find_all_bodies_within_component(comp_inst)
452     .assign body_set = bs.result
453     .assign body_num = cardinality body_set
454     .//print "Component ${comp_inst.Name} has ${body_num} bodies."
455     .select many non_empty_body_set related by body_set->ACT_BLK[R601]->ACT_SMT[R602]->ACT_BLK[R602]->ACT_ACT[R601]
456     .assign non_empty_body_num = cardinality non_empty_body_set
457     .//print "Component ${comp_inst.Name} has ${non_empty_body_num} non-empty bodies."
458     .for each body in non_empty_body_set
459         .invoke cbc = count_body_decisons_and_calls(body)
460         .//CCbody = D+Ncall-1
461         .assign cc_current_body = cbc.result-1
462         .assign key =  "${body.Type}_${body.label}"
463         .assign value = "${cc_current_body}"
464 ${key},${value}
465         .assign attr_result = attr_result+cc_current_body
466     .end for
467     .assign totalDecisionsAndCalls = attr_result
468     .assign attr_result = ((attr_result-non_empty_body_num)+2)
469     .assign key =  "TOTAL NUMBER OF NON_EMPTY BODIES: "
470     .assign value = "${non_empty_body_num}"
471 ${key},${value}
472     .assign key =  "TOTAL CC OF BODIES (decisions and calls): "
473     .assign value = "${totalDecisionsAndCalls}"
474 ${key},${value}
475 .end function
```

*Figure 4.16. Functions producing literal text during generation of CSV files.*

Functions in RSL may have multiple return values that are accessed using the dot operator on the function's *return structure* (the *t1* in line 504 and *t2* in line 510 on figure 4.15). An implicit return value, *body*, returns all literal text produced by the function. The literal text produced by a function includes all lines in a function definition that do not start with a dot, including spaces, newlines and tabs. Notice that the literal text produced by a function can, and usually does, contain values of variables (${key} and ${value} on figure 4.16). This feature of RSL enables complex textual operations and calculations which can spread across a hierarchy of functions. Except the literal text, the functions can also include textual templates (line 512 on figure 4.15). Prior the inclusion of a template, all variables used by the template should be set. Upon inclusion, the text of the template is added to the implicit textual buffer together with values of the variables it uses. Finally, when the text in textual buffer is complete and there are no additional templates to be included, it's content can be

emitted to a file and written to the file system (*emit* statements in lines 507 and 513 on figure 4.15).

Function *generate_cc_comp_class* first looks for all classes in the component and then, for each class, calculates its contribution to the overall cyclomatic complexity according to eq. 3.17. A class contribution (function *calculate_cc_class_complete* on figure 4.17) consists of contribution of all its bodies and a contribution of its state machine, if it exists. Contribution of a state machine is calculated using the *calculate_cc_sm* function from figure 4.8. The function *count_class_decisions_and_calls* on figure 4.18 shows how decisions and calls in a class are counted. First, we had to find a set of all bodies in a class. This is done using selections across several different relation chains:

- **Class operation bodies**: starting from an instance of class, we traverse the relation R115 to *Operation* (*O_TFR*), then relation R696 to *Operation Body* (*ACT_OPB*) and finally relation R698 to *Body*(*ACT_ACT*) meta-class
- **Derived attribute bodies**: starting from an instance of class, we traverse the relation R102 to *Attribute* (*O_ATTR*), then relation R106 to *Base Attribute* (*ACT_BATTR*) then relation R107 to *Derived Base Attribute* (*O_DBATTR*), then relation R693 to *Derived Attribute Body* (*(ACT_DAB)*) and finally relation R698 to *Body* (*ACT_ACT*) meta-class
- **State bodies**: starting from an instance of class, we traverse the relation R518 to *Instance State Machine* (*SM_ISM*), then relation R517 to *State Machine* (*SM_SM*), then relation R515 to *Action* (*SM_ACT*), then relation R691 to *State Action Body* (*ACT_SAB*) and finally relation R698 to *Body* (*ACT_ACT*) meta-class
- **Transition bodies**: starting from an instance of class, we traverse the relation R518 to *Instance State Machine* (*SM_ISM*), then relation R517 to *State Machine* (*SM_SM*), then relation R515 to *Action* (*SM_ACT*), then relation R688 to *Transition Action Body* (*ACT_TAB*) and finally relation R698 to *Body* (*ACT_ACT*) meta-class

Notice that all previously listed relation chains start and end with same meta-classes, but the path, as well as the result, is different. For each of those bodies, we then invoke *count_body_decisions_and_calls* function shown on figure 4.14 and summarize the result for all the bodies (lines 307-310 on figure 4.18). This information is then used, together with state machine contribution, to calculate the contribution of a class to the overall cyclomatic complexity.

After calculating the contribution of a class to the overall cyclomatic complexity, its value, together with the class name, is written as plain text to implicit *body* return value of the function (line 442 in figure 4.16). Upon return of the function call, that value will be written to implicit textual buffer and emitted to a *CSV* file (see figure 4.15).

Similarly as the function *generate_cc_comp_class* for classes, the function *generate_cc_comp_body_decisions_and_calls* on figure 4.16 maps component bodies to their

contribution to cyclomatic complexity and writes this data to a CSV file. It first looks for bodies within a component and then filters out the empty ones (lines 451 and 455). For each non-empty body, we then count decisions and calls using function *count_body_decisions_and_calls* (see figure 4.14 and eq. 3.19). The data written to a CSV file is statistically analysed in Microsoft Excel tool and the results are presented in the next chapter.

```
211 .function calculate_cc_class_complete
212 .param inst_ref cls_inst
213     .assign attr_result = 0
214     .invoke csc = calculate_cc_sm(cls_inst, true)
215     .assign cc_class_sm = csc.result
216     .invoke cdc = count_class_decisons_and_calls(cls_inst)
217     .assign class_decisions_and_calls = cdc.result
218     .invoke cbc = count_class_bodies(cls_inst)
219     .assign class_bodies = cbc.result
220     .assign attr_result = (class_decisions_and_calls - class_bodies) + cc_class_sm
221     .print "Class ${cls_inst.Name} (with state machine and bodies) contributes to cyclomatic complexity: ${attr_result}"
222 .end function
```

*Figure 4.17. Calculating class contribution to overall cyclomatic complexity.*

```
279 .function count_class_decisons_and_calls
280 .param inst_ref cls_inst
281     .assign attr_result = 0
282     .assign body_num = 0
283     .assign allDecisionsAndCalls = 0
284     .select many body_set from instances of ACT_ACT where false
285     .//Class operation bodies
286     .select many op_body_set related by cls_inst->O_TFR[R115]->ACT_OPB[R696]->ACT_ACT[R698]
287     .assign ob_num =  cardinality op_body_set
288     .//print "Class ${cls_inst.Name} has class operation bodies: ${ob_num}"
289     .assign body_set = body_set|op_body_set
290     .//Derr attr bodies
291     .select many derattr_body_set related by cls_inst->O_ATTR[R102]->O_BATTR[R106]->O_DBATTR[R107]->ACT_DAB[R693]->ACT_ACT[R698]
292     .assign dab_num =  cardinality derattr_body_set
293     .//print "Class ${cls_inst.Name} has derived attributes bodies: ${dab_num}"
294     .assign body_set = body_set|derattr_body_set
295     .//State bodies
296     .select many state_body_set related by cls_inst->SM_ISM[R518]->SM_SM[R517]->SM_ACT[R515]->ACT_SAB[R691]->ACT_ACT[R698]
297     .assign sb_num =  cardinality state_body_set
298     .//print "Class ${cls_inst.Name} has state bodies: ${sb_num}"
299     .assign body_set = body_set|state_body_set
300     .//Transition bodies
301     .select many txn_body_set related by cls_inst->SM_ISM[R518]->SM_SM[R517]->SM_ACT[R515]->ACT_TAB[R688]->ACT_ACT[R698]
302     .assign tb_num =  cardinality txn_body_set
303     .//print "Class ${cls_inst.Name} has transition bodies: ${tb_num}"
304     .assign body_set = body_set|txn_body_set
305     .assign body_num =  cardinality body_set
306     .//
307     .for each body in body_set
308         .invoke dac = count_body_decisons_and_calls(body)
309         .assign allDecisionsAndCalls = allDecisionsAndCalls + dac.result
310     .end for
311     .assign attr_result = allDecisionsAndCalls
312     .//print "Class ${cls_inst.Name} has processing model complexity of: ${attr_result}"
313 .end function
```

*Figure 4.18. Counting decisions and calls in all bodies within a class.*

## 4.3 Implementation of entropy complexity metric

Entropy as a complexity metric is explained in section 3.2. The main challenge when calculating an entropy is symbol selection. In case of xtUML, we used model element types

(meta-classes) as symbols and the number of instances (model elements) to determine their probability. Since there are 435 classes in the meta-model, creating an RSL script that will export a short name of each meta-class and the number of its instances into a CSV file is not a minor task. In following sections we will see how we solved this problem and simplified the calculus of entropy metrics.

### 4.3.1 Vertical distribution of entropy complexity metric

The xtUML meta-model is available as an application model from the *welcome* screen in BridgePoint tool. Having xtUML meta-model as any other application model in the workspace enables us to treat it the same way: we can use it as input to the translation process. We created a simple RSL script that operates on the xtUML meta-model in order to generate another, much larger, RSL script. The main template used by the first RSL script (see figure 4.19) is relatively simple and consists of several lines:

- Selection statement that selects all instance of given meta-class
- Counting the number of all instances of a meta-class using *cardinality* operator
- If the number of meta-class instances is not zero, a line containing short name of a meta-class (key letters actually) and the number of found instances is written as literal text to the implicit textual buffer

After applying this template to to all 435 meta-classes from xtUML meta-model, we got much larger script (see figure 4.20). The second script is then applied to application models used in the experiment. Generated CSV files were then used as input for further analysis in the Microsoft Excel tool.

```
${indent}.select many ${cls.Key_Lett}_inst_set from instances of ${cls.Key_Lett}
${indent}.assign  ${cls.Key_Lett}_num = cardinality ${cls.Key_Lett}_inst_set
${indent}.print "Metaclass ${cls.Name} (${cls.Key_Lett}) has $${${cls.Key_Lett}_num} instances."
${indent}.if(not (${cls.Key_Lett}_num==0))
${cls.Key_Lett} $${${cls.Key_Lett}_num}
${indent}.end if
```

*Figure 4.19. The main template used by the first RSL script (used to generate the second RSL script).*

When calculating distribution of entropy across xtUML model layers, we also needed to associate each meta-class with the layer it belongs to. As already discussed before, for some meta-classes it was not clear to which layer they belong. All such classes are considered to belong to OTHER category. This categorization was impossible to automate, and is done manually for each meta-class. An entropy of each layer is then calculated using this categorization and the number of instances of each meta-class.

```
C_C ${C_C_num}
    .end if
    .select many C_DG_inst_set from instances of C_DG
    .assign  C_DG_num = cardinality C_DG_inst_set
    .print "Metaclass Delegation (C_DG) has ${C_DG_num} instances."
    .if(not (C_DG_num==0))
C_DG ${C_DG_num}
    .end if
    .select many C_EP_inst_set from instances of C_EP
    .assign  C_EP_num = cardinality C_EP_inst_set
    .print "Metaclass Executable Property (C_EP) has ${C_EP_num} instances."
    .if(not (C_EP_num==0))
C_EP ${C_EP_num}
    .end if
    .select many C_IO_inst_set from instances of C_IO
    .assign  C_IO_num = cardinality C_IO_inst_set
    .print "Metaclass Interface Operation (C_IO) has ${C_IO_num} instances."
    .if(not (C_IO_num==0))
C_IO ${C_IO_num}
    .end if
    .select many C_RID_inst_set from instances of C_RID
    .assign  C_RID_num = cardinality C_RID_inst_set
    .print "Metaclass Interface Reference In Delegation (C_RID) has ${C_RID_num} instances."
    .if(not (C_RID_num==0))
C_RID ${C_RID_num}
    .end if
    .select many C_IR_inst_set from instances of C_IR
    .assign  C_IR_num = cardinality C_IR_inst_set
    .print "Metaclass Interface Reference (C_IR) has ${C_IR_num} instances."
    .if(not (C_IR_num==0))
C_IR ${C_IR_num}
    .end if
    .select many C_AS_inst_set from instances of C_AS
    .assign  C_AS_num = cardinality C_AS_inst_set
    .print "Metaclass Interface Signal (C_AS) has ${C_AS_num} instances."
    .if(not (C_AS_num==0))
C_AS ${C_AS_num}
```

*Figure 4.20. The second, generated, RSL script used on actual application models.*

Comparison of application model with regard to entropy complexity is presented in the next chapter. All scripts, templates and excel files used are accessible in *entropyMetricsCalculation\vertically* directory on repository [55].

### 4.3.2 Horizontal distribution of entropy complexity

Calculating the horizontal distribution of entropy across classes introduced an additional challenge of associating each model element (instance of meta-class) to the application class it belongs. Of course, this was not possible to do for all meta-classes because only a small subset of meta-classes can be considered *to belong* to a class. However, since classes are main containers of application complexity in xtUML, a large share of total instances in the model is included. In order to associate a model element to its class, we had to identify a chain of relations that describes a "belongs to" relation between the model element type (meta-class) and a *Model Class* (*O_OBJ*) meta-class. This process was impossible to completely automate as this relation chain differs for each meta-class. However, similarly as when calculating vertical distribution of entropy, we partially generated the script used on application models. During generation of that script we applied a simple template to all meta-

classes (figure 4.21). After generated, the script was manually edited. For each meta-class that can be considered to *belong* to an application class, we uncommented the text from the template and completed the selection statement. That statement actually selects all instances of observed meta-class related to the given application class.

```
${indent}${indent}.//select many ${mmClass.Key_Lett}_set related by class->...->${mmClass.Key_Lett}[Rxxx]
${indent}${indent}.//assign ${mmClass.Key_Lett}_set_cardinality = cardinality ${mmClass.Key_Lett}_set
${indent}${indent}.//print "${mmClass.Key_Lett} $${${mmClass.Key_Lett}_set_cardinality} $${class.Name}"
${indent}.//${mmClass.Key_Lett} $${${mmClass.Key_Lett}_set_cardinality} $${class.Name}
```

*Figure 4.21. A template with commented RSL lines used for partial generation of script for horizontal distribution of entropy.*

Figure 4.22 shows an excerpt from the script dealing with bodies and statements. We first found all bodies of a class (line 41) and then, starting from that set of bodies, we selected all instance of a given meta-class. As can be seen in line 51, an instance of *Statement* (*ACT_STM*) can be related to class, but an instance of *Provided Signal Body* (*ACT_PSB*) (line 67), which is part of a port, cannot (and is therefore left uncommented). Each line of literal text in the script contains a meta-class name, its cardinality in a given application class and the name of application class it belongs to. This is done for each class in the application model and for each meta-class that can be associated with a class.

A similar approach is used when calculating horizontal distribution of elements across bodies. The most important step here was to identify a relation chain that associates observed meta-class to the *Body* (*ACT_ACT*) meta-class. The difference from the script shown on figure 4.22 is the starting part of selection lines. Instead looking for all class bodies and then selecting instances of each meta-class associated to that set of bodies (and implicitly to the given class), we are starting from a single body and then selecting instances of each meta-class associated to that body. Of course, a very small subset of meta-classes can be associated to a body, but, since bodies include majority of model elements, a large share of total elements is included. Each literal text line in the script contains a meta-class name, its cardinality in the body and the name of that body. This is done for each body in the application model and for each meta-class that can be associated with a body. Since there are much more bodies then classes in application models, generated CSV file is much larger for horizontal distribution across bodies than the one across classes.

CSV files generated by the scripts are used as input for further analysis required for measuring the entropy metric. This process was done using Microsoft Excel tool and the results are presented in the next chapter. All scripts, templates and Excel files used are accessible in *entropyMetricsCalculation\horizontally* directory on repository [55].

```
36    .select many class_set from instances of O_OBJ
37    .for each class in class_set
38    .print "Looking for metaclass instances related to class ${class.Name}"
39       .//---------------------------------------------------------------------
40       .//Body
41       .invoke alcb = get_all_class_bodies(class)
42       .assign ACT_ACT_set = alcb.result
43       .//
44       .assign ACT_ACT_set_cardinality = cardinality ACT_ACT_set
45       .print "ACT_ACT ${ACT_ACT_set_cardinality} ${class.Name}"
46 ACT_ACT ${ACT_ACT_set_cardinality} ${class.Name}
47       .select many ACT_BLK_set related by ACT_ACT_set->ACT_BLK[R601]
48       .assign ACT_BLK_set_cardinality = cardinality ACT_BLK_set
49       .print "ACT_BLK ${ACT_BLK_set_cardinality} ${class.Name}"
50 ACT_BLK ${ACT_BLK_set_cardinality} ${class.Name}
51       .select many ACT_SMT_set related by ACT_ACT_set->ACT_BLK[R601]->ACT_SMT[R602]
52       .assign ACT_SMT_set_cardinality = cardinality ACT_SMT_set
53       .print "ACT_SMT ${ACT_SMT_set_cardinality} ${class.Name}"
54 ACT_SMT ${ACT_SMT_set_cardinality} ${class.Name}
55       .select many ACT_BRK_set related by ACT_ACT_set->ACT_BLK[R601]->ACT_SMT[R602]->ACT_BRK[R603]
56       .assign ACT_BRK_set_cardinality = cardinality ACT_BRK_set
57       .print "ACT_BRK ${ACT_BRK_set_cardinality} ${class.Name}"
58 ACT_BRK ${ACT_BRK_set_cardinality} ${class.Name}
59       .//select many ACT_BRB_set related by class->...->ACT_BRB[Rxxx]
60       .//assign ACT_BRB_set_cardinality = cardinality ACT_BRB_set
61       .//print "ACT_BRB ${ACT_BRB_set_cardinality} ${class.Name}"
62    .//ACT_BRB ${ACT_BRB_set_cardinality} ${class.Name}
63       .//select many ACT_POB_set related by class->...->ACT_POB[Rxxx]
64       .//assign ACT_POB_set_cardinality = cardinality ACT_POB_set
65       .//print "ACT_POB ${ACT_POB_set_cardinality} ${class.Name}"
66    .//ACT_POB ${ACT_POB_set_cardinality} ${class.Name}
67       .//select many ACT_PSB_set related by class->...->ACT_PSB[Rxxx]
68       .//assign ACT_PSB_set_cardinality = cardinality ACT_PSB_set
69       .//print "ACT_PSB ${ACT_PSB_set_cardinality} ${class.Name}"
70    .//ACT_PSB ${ACT_PSB_set_cardinality} ${class.Name}
71       .//select many ACT_ROB_set related by class->...->ACT_ROB[Rxxx]
72       .//assign ACT_ROB_set_cardinality = cardinality ACT_ROB_set
73       .//print "ACT_ROB ${ACT_ROB_set_cardinality} ${class.Name}"
74    .//ACT_ROB ${ACT_ROB_set_cardinality} ${class.Name}
75       .//select many ACT_RSB_set related by class->...->ACT_RSB[Rxxx]
76       .//assign ACT_RSB_set_cardinality = cardinality ACT_RSB_set
77       .//print "ACT_RSB ${ACT_RSB_set_cardinality} ${class.Name}"
78    .//ACT_RSB ${ACT_RSB_set_cardinality} ${class.Name}
79       .select many ACT_CTL_set related by ACT_ACT_set->ACT_BLK[R601]->ACT_SMT[R602]->ACT_CTL[R603]
80       .assign ACT_CTL_set_cardinality = cardinality ACT_CTL_set
81       .print "ACT_CTL ${ACT_CTL_set_cardinality} ${class.Name}"
82 ACT_CTL ${ACT_CTL_set_cardinality} ${class.Name}
```

*Figure 4.22. Partially generated script that relates model elements to the class it belongs.*

## 4.4 Implementation of data complexity metric

Data complexity metrics are explained in chapter 3.3. This chapter will explain how xtUML data type and data flow complexities are implemented.

```
    .select any comp_inst from instances of C_C where selected.Name==comp_name
    .if(not_empty comp_inst)
        .print "Starting..."
        .invoke cic = find_classes_in_component(comp_inst)
        .assign class_set = cic.result
        .assign total_primitive_values = 0
        .for each class in class_set
            .assign cls_primitive_values = 0
            .select many attr_set related by class->O_ATTR[R102]
            .for each attr in attr_set
                .if(attr.Name!="current_state")
                    .select one cdt_type related by attr->S_DT[R114]->S_CDT[R17]
                    .if(not_empty cdt_type)
                        .print "Class ${class.Name} has member ${attr.Name} of core data type!"
                        .assign cls_primitive_values = cls_primitive_values + 1
                    .end if
                    .//
                    .select one enum_attr_type related by attr->S_DT[R114]->S_EDT[R17]
                    .if(not_empty enum_attr_type)
                        .print "Class ${class.Name} has attr ${attr.Name} of enumerated data type!"
                        .assign cls_primitive_values = cls_primitive_values + 1
                    .end if
                    .//
                    .select one irdt_attr_type related by attr->S_DT[R114]->S_IRDT[R17]
                    .if(not_empty irdt_attr_type)
                        .print "ERROR: Class ${class.Name} has attr ${attr.Name} of instance reference data type!"
                    .end if
                    .//
                    .select one sdt_attr_type related by attr->S_DT[R114]->S_SDT[R17]
                    .if(not_empty sdt_attr_type)
                        .print "ERROR: Class ${class.Name} has attr ${attr.Name} of structured data type!"
                        .invoke sdtmc = count_structured_type_primitive_members(sdt_attr_type)
                        .assign cls_primitive_values = cls_primitive_values + sdtmc.result
                    .end if
                .end if
            .end for
            .print "${class.Key_Lett} ${cls_primitive_values}"
            .//Count the number of relations
            .select many rel_set related by class->R_OIR[R201]->R_REL[R201]
            .assign rel_num = cardinality rel_set
            .print "Class ${class.Name} is involved in ${rel_num} relations."
            .assign cls_primitive_values = cls_primitive_values + rel_num
${class.Key_Lett} ${cls_primitive_values}
            .assign total_primitive_values = total_primitive_values + cls_primitive_values
        .end for
        .print "TOTAL ${total_primitive_values}"
TOTAL ${total_primitive_values}
        .print "DONE!"
        .select any sys from instances of S_SYS
        .print "Sys.ID = ${sys.Sys_ID}"
        .emit to file "C:/Temp/${sys.Name}_DataTypeComplexityPerClasses.csv"
```

*Figure 4.23. Calculating distribution of data type complexity across classes.*

### 4.4.1 Distribution of data type complexity metric

In xtUML, data modelling is usually done using a class model, but can also be done using data structures. Since usage of data structures is strongly discouraged (except in the cases when they are required), data type complexity metrics will only be applied to a class model. This also implies that we will only measure the total data type complexity and horizontal distribution across classes. The only case when complexity of a structured data type (given with equation 3.25) will be taken into account is when it appears as a type of a class attribute. In that case, data structure complexity affects class data type complexity and cannot be ignored.

Figure 4.23 shows how the data type complexity is implemented. For each class we first select a set of its attributes (line 81). For attributes of core type (*S_CDT*, line 84) or enumerated data type (*S_EDT*, line 90) we increment the data type complexity. If an attribute type is *Instance Reference Data Type* (*S_IRDT*, line 96) it will be ignored and error will be printed. The reason for this is the fact that classes should not have other classes as type of their attributes. Such attributes represent unidirectional relations that are not shown on class model as all other relations. Although such practice is technically allowed by the BridgePoint tool, it is considered as a bad practice. If an attribute is of structured type (*S_SDT*, line 101), data type complexity is increased by the number of primitive fields hierarchically contained within the structured type (line 104). In a case when structured data type has a field typed by other structured type, the function *count_structured_type_primitive_members* has a recursive call to itself (line 27 on figure 4.24). Relevant part of meta-model can be seen on figure 4.25.

```
2 .function count_structured_type_primitive_members
3 .param inst_ref sdt
4     .select one sdt_dt related by sdt->S_DT[R17]
5     .assign attr_result = 0
6     .select many member_set related by sdt->S_MBR[R44]
7     .for each member in member_set
8         .select one core_member_type related by member->S_DT[R45]->S_CDT[R17]
9         .if(not_empty core_member_type)
10            .print "Structure data type ${sdt_dt.Name} has member ${member.Name} of core data type!"
11            .assign attr_result = attr_result + 1
12        .end if
13        .//
14        .select one enum_member_type related by member->S_DT[R45]->S_EDT[R17]
15        .if(not_empty enum_member_type)
16            .print "Structure data type ${sdt_dt.Name} has member ${member.Name} of enumerated data type!"
17            .assign attr_result = attr_result + 1
18        .end if
19        .//
20        .select one irdt_member_type related by member->S_DT[R45]->S_IRDT[R17]
21        .if(not_empty irdt_member_type)
22            .print "ERROR: Structure data type ${sdt_dt.Name} has member ${member.Name} of instance reference data type!"
23        .end if
24        .select one sdt_member_type related by member->S_DT[R45]->S_SDT[R17]
25        .if(not_empty sdt_member_type)
26            .print "ERROR: Structure data type ${sdt_dt.Name} has member ${member.Name} of structured data type!"
27            .invoke sdtmc = count_structured_type_primitive_members(sdt_member_type)
28            .assign attr_result = attr_result + sdtmc.result
29        .end if
30    .end for
31    .print "Structure data type ${sdt_dt.Name} has ${attr_result} primitive members"
32 .end function
```

*Figure 4.24. Recursive function for counting the number of primitive fields within a structured type.*

Finally, the number of relations and their effect on data complexity of a class is calculated in lines 111-114 (figure 4.23). Starting from an instance of a class, we select all relations it is involved in by traversing the associative relations R201 (see figure 4.26). The complete RSL source code for distribution of data type complexity acros classes can be found in file *DataTypeComplexity.arc* in *dataComplexity\DataTypeComplexity* directory on repository [55]

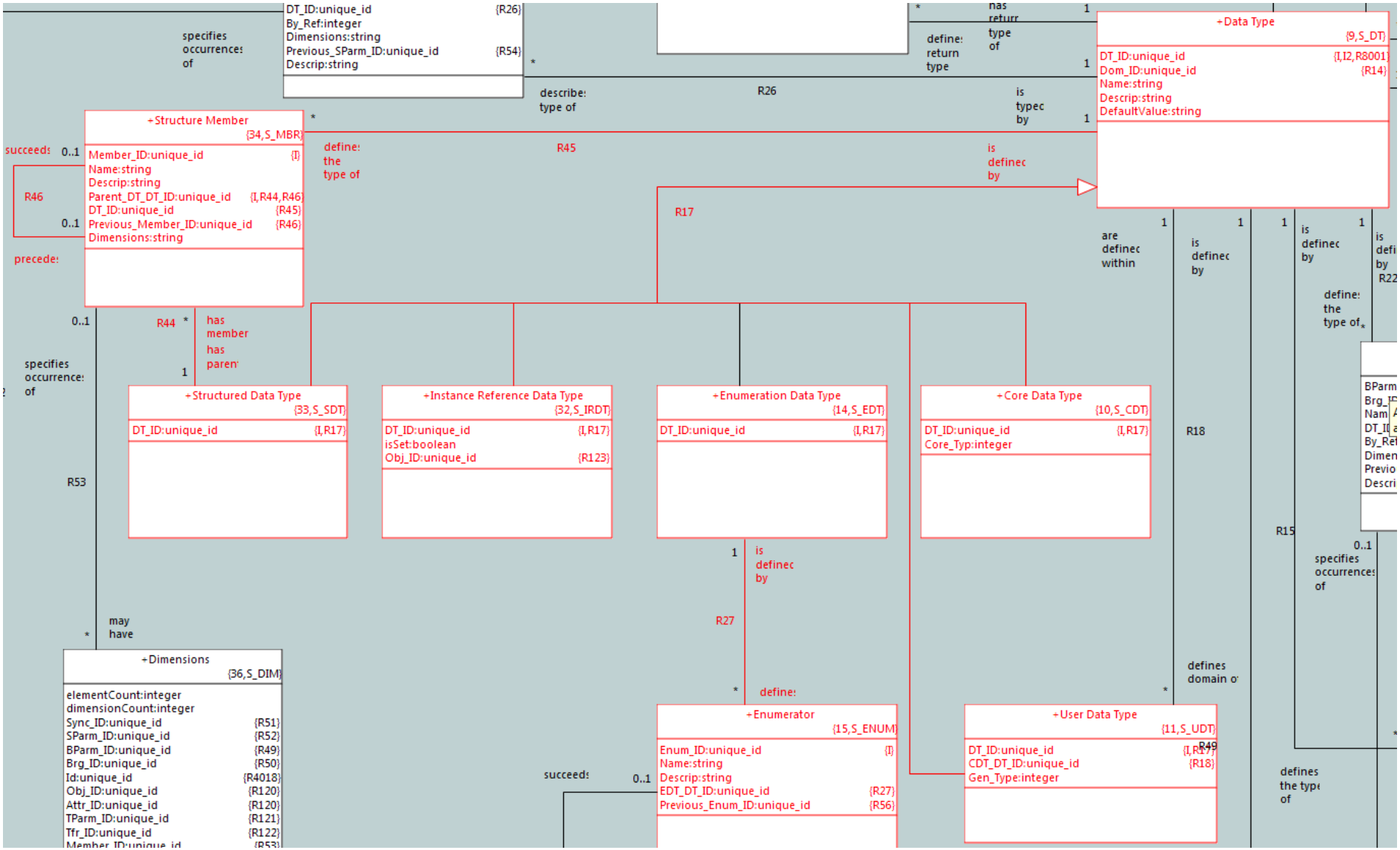



*Figure 4.25. Part of xtUML meta-model describing data types.*

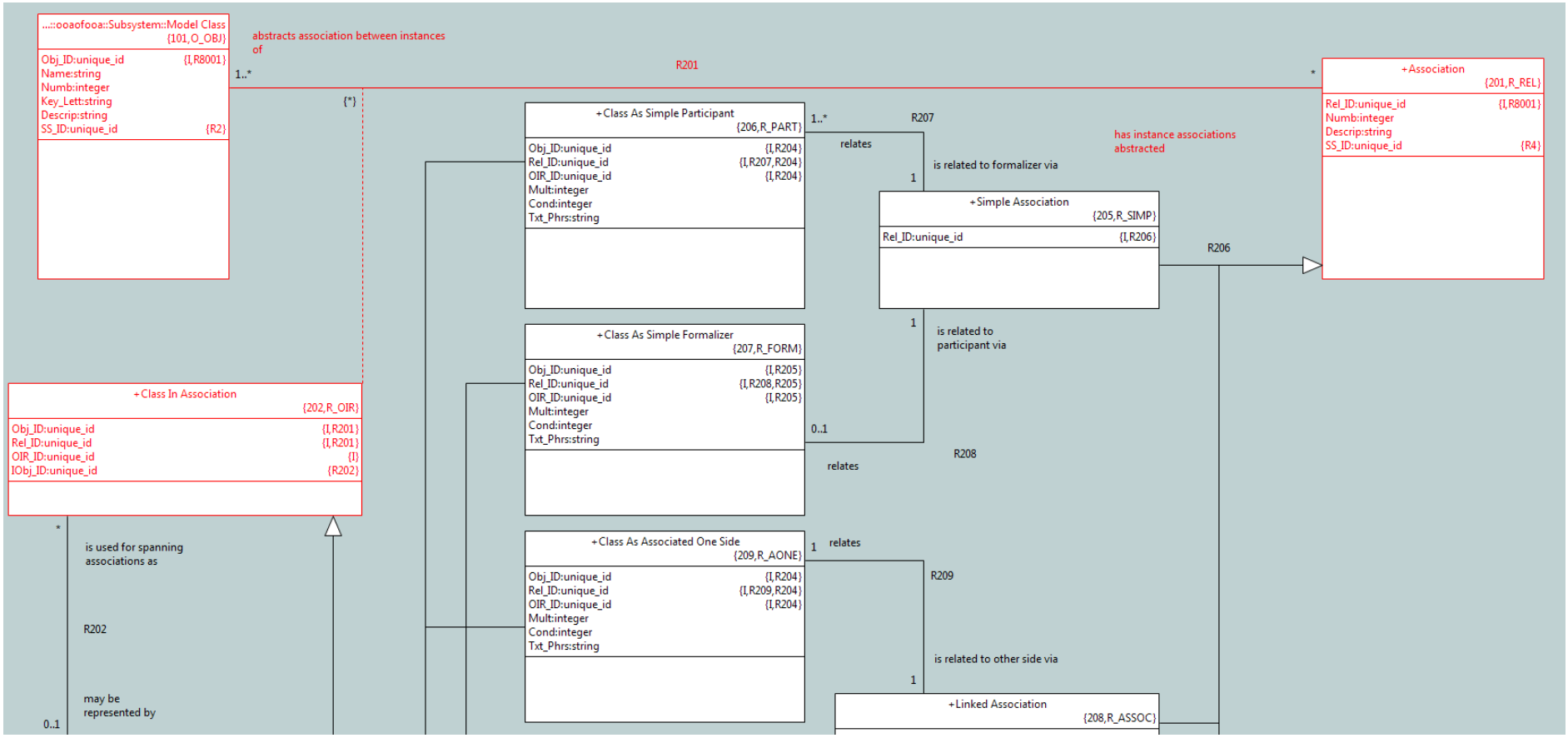



*Figure 4.26. Part of xtUML meta-model describing relations.*

### 4.4.2 Vertical distribution of data flow complexity metric

Data flow complexity metric is explained in details in section 3.3.3. The main idea is that, from the perspective of data flow complexity, xtUML model can be observed as a set of bodies that communicate synchronously or asynchronously with each other. This implies that other xtUML models, such as component, class and state machine model, do not affect the total model data flow complexity. Similarly as in other metrics, those models partially visualize the data flow complexity. Vertical distribution of data flow complexity measures the amount of total data flow complexity that is shown on those models.

Figure 4.27 shows the RSL code that calculates the distribution of data flow complexity across different layers of an xtUML model. The output of that code is a simple CSV file with five lines (check the literal text lines on the figure 4.27) indicating the absolute amount of data flow complexity of each layer.

```
280     .select any comp from instances of C_C where selected.Name==comp_name
281     .if(not_empty comp)
282         .//Component data flow complexity
283         .invoke cdfc = calculate_all_comp_port_data_flow_complexity(comp)
284         .assign comp_port_param_num = cdfc.result
285         .print "Component ${comp.Name} on port UI has ${comp_port_param_num} parameters on incoming messages."
286 COMP,${comp_port_param_num}
287         .//Class data flow complexity
288         .invoke ccldfc = calculate_all_comp_class_data_flow_complexity(comp)
289         .assign class_operation_param_num = ccldfc.result
290         .print "Component ${comp.Name} has ${class_operation_param_num} parameters on all operations of all classes."
291 CLASS,${class_operation_param_num}
292         .//State machine data flow complexity
293         .invoke csmdfc = calculate_all_comp_state_machine_data_flow_complexity(comp)
294         .assign sm_evtdti_num = csmdfc.result
295         .print "Component ${comp.Name} has ${sm_evtdti_num} event data items on all transitions and states with non-empty bodies."
296 SM,${sm_evtdti_num}
297         .//Body data flow complexity
298         .invoke cbdfc = calculate_all_comp_body_data_flow_complexity(comp)
299         .assign body_def_num = cbdfc.result
300         .print "Component ${comp.Name} has ${body_def_num} definitions in all its bodies."
301 BODY,${body_def_num}
302         .assign visual_df = (comp_port_param_num + class_operation_param_num) + sm_evtdti_num
303 VISUAL,${visual_df}
304     .end if
305     .print "DONE!"
306     .select any sys from instances of S_SYS
307     .print "Sys.ID = ${sys.Sys_ID}"
308     .emit to file "C:/Temp/${sys.Name}_DataFlowComplexity_vertical.csv"
```

*Figure 4.27. Calculating vertical distribution of data flow complexity.*

Function *calculate_all_comp_port_data_flow_complexity* calculates data flow complexity of a component model by counting the parameters on all incoming port messages (operations and signals, see figure 4.28). It first selects all ports (*C_PO*) of the component (line 150) and then, for each port, it selects all incoming signals (*C_AS*) across relations R4016, R4012, R4003 and R4004. Finally, all its parameters are selected (line 157) and counted. This procedure could have been much shorter, by using only one selection starting from a component, but the traversed chain would be too long and too complex for understanding. A similar selection process is repeated for incoming operations (*C_IO*). The sum of all parameters of all incoming messages is written to *attr_result* variable which is implicitly returned

from the function[5].

Function *calculate_all_comp_class_data_flow_complexity* calculates data flow complexity of all classes within a component by counting the parameters on all their operations. The function first looks for all classes within a component (line 172), and then searches for their operations (*O_TFR*, line 175). Finally, for each operation we count the parameters (*O_TPARAM*) and return their total number.

```
147 .function calculate_all_comp_port_data_flow_complexity
148 .param inst_ref comp
149     .assign attr_result = 0
150     .select many port_set related by comp->C_PO[R4010]
151     .for each port in port_set
152         .//IFDirectionType::ServerClient = 1; IFDirectionType::ClientServer = 0
153         .//
154         .//Port incomming signal parameters
155         .select many in_sig_set related by port->C_IR[R4016]->C_I[R4012]->C_EP[R4003]->C_AS[R4004] where selected.Direction==0
156         .for each in_sig in in_sig_set
157             .select many sig_param_set related by in_sig->C_EP[R4004]->C_PP[R4006]
158             .assign attr_result = attr_result + (cardinality sig_param_set)
159         .end for
160         .//Port incomming operation parameters
161         .select many in_op_set related by port->C_IR[R4016]->C_I[R4012]->C_EP[R4003]->C_IO[R4004] where selected.Direction==0
162         .for each in_op in in_op_set
163             .select many op_param_set related by in_op->C_EP[R4004]->C_PP[R4006]
164             .assign attr_result = attr_result + (cardinality op_param_set)
165         .end for
166     .end for
167 .end function
168 .//
169 .function calculate_all_comp_class_data_flow_complexity
170 .param inst_ref comp
171     .assign attr_result = 0
172     .invoke ac = find_classes_in_component(comp)
173     .assign class_set = ac.result
174     .for each class in class_set
175         .select many op_set related by class->O_TFR[R115]
176         .for each op in op_set
177             .select many param_set related by op->O_TPARM[R117]
178             .assign op_param_num = cardinality param_set
179             .assign attr_result = attr_result + op_param_num
180             .//print "Operation ${op.Name} in class ${class.Name} has ${op_param_num} parameters!"
181         .end for
182     .end for
183 .end function
```

*Figure 4.28. Calculating data flow complexity visualized by class and component model.*

Function *calculate_all_comp_state_machine_data_flow_complexity* calculates the data flow complexity for all state machines within a component. The main carriers of data in a state machine are events which can be parametrized by event data items (*SM_EVDTI*). Event data items are available as parameters in the body of transitions and states. However, since a state can receive more than one event with different event data items, we can only access a set of event data items common to all incoming events. First, a set of non-empty states in the component is selected (line 191) and then, for each state, a set of incoming events (line 193). In the lines 197-207, a common set of event data items for all incoming events of a state is found by performing set intersection operation (line 204). The calculus for a transition event data items in the following lines is much simpler (lines 212-218) since they can only be triggered by a single event.

[5] All variables that start with *attr_* are accessible after function returns. In this way, RSL supports arbitrary number of return values.

```
.function calculate_all_comp_state_machine_data_flow_complexity
.param inst_ref comp
    .assign attr_result = 0
    .invoke ab = find_all_bodies_within_component(comp)
    .assign all_bodies = ab.result
    .select many non_empty_bodies related by all_bodies->ACT_BLK[R601]->ACT_SMT[R602]->ACT_BLK[R602]->ACT_ACT[R601]
    .select many non_empty_state_set related by non_empty_bodies->ACT_SAB[R698]->SM_ACT[R691]->SM_AH[R514]->SM_MOAH[R513]->SM_STATE[R511]
    .for each non_empty_state in non_empty_state_set
        .select many recvd_evt_set related by non_empty_state->SM_TXN[R506]->SM_NSTXN[R507]->SM_SEME[R504]->SM_SEVT[R503]->SM_EVT[R525]
        .//Declare the variable used in the loop
        .select many common_evtdti_set from instances of SM_EVTDI where false
        .assign isFirstProcessedEvent = true
        .for each recvd_evt in recvd_evt_set
            .select many evdti_set related by recvd_evt->SM_EVTDI[R532]
            .if(isFirstProcessedEvent)
                .//If this is first event to be processed, initialize the common evdti set to its evtdti set
                .assign common_evtdti_set = evdti_set
            .else
                .//Else find intersection of the current event evdti set and common evtdti set
                .assign common_evtdti_set = common_evtdti_set&evdti_set
            .end if
        .end for
        .assign common_evtdti_num = (cardinality common_evtdti_set)
        .print "State ${non_empty_state.Name} receives ${common_evtdti_num} common event data items."
        .assign attr_result = attr_result + common_evtdti_num
    .end for
    .//
    .select many non_empty_txn_set related by non_empty_bodies->ACT_TAB[R698]->SM_ACT[R688]->SM_AH[R514]->SM_TAH[R513]->SM_TXN[R530]
    .for each non_empty_txn in non_empty_txn_set
        .select many txn_evtdi_set related by non_empty_txn->SM_NSTXN[R507]->SM_SEME[R504]->SM_SEVT[R503]->SM_EVT[R525]->SM_EVTDI[R532]
        .assign txn_evtdi_num = cardinality txn_evtdi_set
        .print "Transition behavior parametrized with ${txn_evtdi_num} event data items."
        .assign attr_result = attr_result + txn_evtdi_num
    .end for
.end function
```

*Figure 4.29. Calculating data flow complexity visualized by state machine model.*

Data flow complexity of a complete model is calculated by calculating the data flow complexity of all its non-empty bodies. The function *calculate_all_comp_body_data_flow* on figure 4.30 calculates the body data flow complexity for a component, which, in our case, represents the whole application model. As usual, we first found all bodies within a component (line 224) and then filtered out the empty ones (line 226). For each non-empty body the function *calculate_data_flow_complexity_for_single_body* is invoked which calculates data flow complexity of a body. The function counts the number of variable definitions within that body, including implicit parameter and event data item definitions. This practically means that it counts:

- **All variable-defining statements:** assignment statements (*ACT_AI*, line 239), instance creation statements (*ACT_CR*, line 243) and all selection statements (*ACT_SEL*, *ACT_FIO*, *ACT_FIW* in lines 247, 251 and 255)
- **Parameters** in case a body is a provided signal (line 260), a provided operation(line 265), a function (line 270) or a class operation (line 275)
- **Common event data items** for all incoming events in case the body is a state body (lines 280-293, not shown on the figure)
- **Event data items** if a body is transition body (line 297, not shown on the figure).

Notice that figure 4.30 does not show the complete RSL code for the function and that some lines are not shown entirely (lines 226, 260 and 265). If needed, please refer to *dataComplexity\DataFlowComplexity\Vertically* directory on repository [55] which contains the *DataFlowComplexity_vertical.arc* RSL script as well as the excel files created from exported CSV files.

```
.function calculate_all_comp_body_data_flow_complexity
.param inst_ref comp
    .assign attr_result = 0
    .invoke ab = find_all_bodies_within_component(comp)
    .assign all_bodies = ab.result
    .select many non_empty_body_set related by all_bodies->ACT_BLK[R601]->ACT_SMT[R602]->ACT_BLK[R602]->ACT_ACT[R
    .for each non_empty_body in non_empty_body_set
        .invoke sb = calculate_data_flow_cmplexity_for_single_body(non_empty_body)
        .assign attr_result =attr_result+sb.result
    .end for
.end function
.//
.function calculate_data_flow_cmplexity_for_single_body
.param inst_ref body
    .assign attr_result = 0
    .select one non_empty_body related by body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_BLK[R602]->ACT_ACT[R601]
    .if(not_empty non_empty_body)
        .//Assignment
        .select many assign_stmt_set related by non_empty_body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_AI[R603]
        .assign assign_stmt_num =  cardinality assign_stmt_set
        .assign attr_result =attr_result+assign_stmt_num
        .//Create
        .select many create_stmt_set related by non_empty_body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_CR[R603]
        .assign create_stmt_num =  cardinality create_stmt_set
        .assign attr_result =attr_result+create_stmt_num
        .//select
        .select many select_stmt_set related by non_empty_body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_SEL[R603]
        .assign select_stmt_num =  cardinality select_stmt_set
        .assign attr_result =attr_result+select_stmt_num
        .//select_fio
        .select many select_fio_stmt_set related by non_empty_body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_FIO[R603]
        .assign select_fio_stmt_num =  cardinality select_fio_stmt_set
        .assign attr_result =attr_result+select_fio_stmt_num
        .//select_fiw
        .select many select_fiw_stmt_set related by non_empty_body->ACT_BLK[R601]->ACT_SMT[R602]->ACT_FIW[R603]
        .assign select_fiw_stmt_num =  cardinality select_fiw_stmt_set
        .assign attr_result =attr_result+select_fiw_stmt_num
        .//
        .//Parameter count: provided signals
        .select many provided_port_signal_parameter_set related by non_empty_body->ACT_PSB[R698]->SPR_PS[R686]->S
        .assign provided_port_signal_parameter_num = cardinality provided_port_signal_parameter_set
        .assign attr_result =attr_result+provided_port_signal_parameter_num
        .//
        .//Parameter count: provided operations
        .select many provided_port_operation_parameter_set related by non_empty_body->ACT_POB[R698]->SPR_PO[R687]
        .assign provided_port_operation_parameter_num = cardinality provided_port_operation_parameter_set
        .assign attr_result =attr_result+provided_port_operation_parameter_num
        .//
        .//Parameter count: functions
        .select many function_parameter_set related by non_empty_body->ACT_FNB[R698]->S_SYNC[R695]->S_SPARM[R24]
        .assign function_parameter_num = cardinality function_parameter_set
        .assign attr_result =attr_result+function_parameter_num
        .//
        .//Parameter count: opers
        .select many oper_parameter_set related by non_empty_body->ACT_OPB[R698]->O_TFR[R696]->O_TPARM[R117]
```

*Figure 4.30. Calculating data flow complexity within bodies (total data flow complexity).*

### 4.4.3 Horizontal distribution of data flow complexity metric

Data flow complexity of a class includes all data flow complexity of all its bodies. However, it does not include the data flow complexity visualized by a class, because that data complexity is already included in the data complexity of its bodies.

Figure 4.31 shows the function *calculate_data_flow_complexity_for_class* which calculates the data flow complexity of a class. In order to calculate data flow complexity distri-

bution across application classes, this function is invoked for each class in a component and the data is exported to CSV file. After finding all class bodies (lines 274-292), function *calculate_data_flow_complexity_for_single_body* is invoked for each body. The same function is used for vertical distribution of data flow complexity (see figure 4.30).

Implementation of horizontal distribution of data complexity across bodies is even simpler. First we find all bodies within a component, and then, for each body, we invoke the function *calculate_data_flow_complexity_for_single_body*. The result of each invocation is written to the implicit textual buffer which is, after the function has been invoked on all bodies, emitted to a CSV file for further analysis.

Complete RSL source code for horizontal distribution across classes and bodies can be found in *DataFlowComplexity_horizontal_by_classes.arc* and *DataFlowComplexity_horizontal_by_bodies.arc* files respectively on repository [55]. It also includes an additional excel files with results that are presented in the next chapter.

```
269 .function calculate_data_flow_complexity_for_class
270 .param inst_ref cls_inst
271     .assign attr_result = 0
272     .select many body_set from instances of ACT_ACT where false
273     .//Class operation bodies
274     .select many op_body_set related by cls_inst->O_TFR[R115]->ACT_OPB[R696]->ACT_ACT[R698]
275     .assign ob_num =  cardinality op_body_set
276     .//print "Class ${cls_inst.Name} has class operation bodies: ${ob_num}"
277     .assign body_set = body_set|op_body_set
278     .//Derr attr bodies
279     .select many derattr_body_set related by cls_inst->O_ATTR[R102]->O_BATTR[R106]->O_DBATTR[R107]->ACT_DAB[R693]->ACT_ACT[R698]
280     .assign dab_num =  cardinality derattr_body_set
281     .//print "Class ${cls_inst.Name} has derived attributes bodies: ${dab_num}"
282     .assign body_set = body_set|derattr_body_set
283     .//State bodies
284     .select many state_body_set related by cls_inst->SM_ISM[R518]->SM_SM[R517]->SM_ACT[R515]->ACT_SAB[R691]->ACT_ACT[R698]
285     .assign sb_num =  cardinality state_body_set
286     .//print "Class ${cls_inst.Name} has state bodies: ${sb_num}"
287     .assign body_set = body_set|state_body_set
288     .//Transition bodies
289     .select many txn_body_set related by cls_inst->SM_ISM[R518]->SM_SM[R517]->SM_ACT[R515]->ACT_TAB[R688]->ACT_ACT[R698]
290     .assign tb_num =  cardinality txn_body_set
291     .//print "Class ${cls_inst.Name} has transition bodies: ${tb_num}"
292     .assign body_set = body_set|txn_body_set
293     .//
294     .for each body in body_set
295         .invoke dfc = calculate_data_flow_cmplexity_for_single_body(body)
296         .assign attr_result = attr_result + dfc.result
297     .end for
298     .//print "Class ${cls_inst.Name} has data flow complexity of: ${attr_result}"
299 .end function
```

*Figure 4.31. Calculating horizontal distribution of data flow complexity across classes.*

# 5 Hypothesis and experiment setup

The goal of our experiment is to test the influence of horizontal and vertical distribution of xtUML model complexity on the understandability of software models. For this reason, we specify our hypothesis in the following way:

- **Null hypothesis ($H_0$):** The distribution of complexity does not affect the understandability of xtUML models.
- **Alternative hypothesis ($H_a$):** The distribution of complexity significantly affects the understandability of xtUML models.

In order to test this hypothesis, we have used three different xtUML models of the same application. All three models implement the same set of requirements; they have the same interface and functionality, and they pass the same set of 30 test cases. The only difference is in their internal implementation, which results with the fact that every application has a significantly different horizontal and vertical distribution of complexity. Our goal was to study the effect of complexity distribution on the understandability of xtUML models. To measure how well our test subjects understand each of the three xtUML models, we used an online questionnaire.

## 5.1 Study objects

The experiment objects were three functionally equivalent calculator applications developed with xtUML. Each application is composed out of a single xtUML component, the *SimpleCalculator* component. Each application has its own version of the *SimpleCalculator* component, but all component versions have the same interface and pass the same tests (meaning they are functionally equivalent). The three components are intentionally modelled differently, in order to demonstrate different ways to distribute application complexity:

1. *Model 1* uses the structured programming paradigm which relies on functional decomposition and data structures. It makes no use of object-oriented concepts such as classes, nor does it use state machines.
2. *Model 2* heavily relies on classes, but it does not use state machines.

3. *Model 3* uses both classes and state-machines.

All three applications, each with a model and a test suite, can be found at the *xtUMLProjects* directory at the public git repository available at [55]. In the following sections, we will compare the three models according to various dimensions.

### 5.1.1 Comparing the naming conventions used by models

One of the most important factors influencing software understandability is consistency and quality of naming conventions. Since in this work, we are not interested in the relation between naming conventions and software understandability, we need to eliminate it as a factor. To verify that our models do not differ considerably in that sense, we used Laitinen's language theory idea [5], in which the notion of a *language* refers to the set of symbols and identifiers used in some (software) document. He identified two main rules for comparing such languages:

- Smaller languages, in terms of number of elements, are easier to understand than larger ones. The main idea here is that each symbol has associated semantics in the context of the language, which needs to be understood.
- It is easier to understand closely related languages than more distantly related languages. The closeness of two languages is determined by the number of common symbols (and semantics) they share. This implies that no absolute measure of language understandability exists, only relative measures are possible.

*Table 5.1. Number of common words in the languages of three models and the language of the specification*

| | **Model 1 language** | **Model 2 language** | **Model 3 language** | **Specification language** |
|---|---|---|---|---|
| **Model 1 language** | **315** | 262 | 261 | **118** |
| **Model 2 language** | 262 | **332** | 305 | **123** |
| **Model 3 language** | 261 | 305 | **334** | **122** |
| **Specification language** | **118** | **123** | **122** | **270** |

Our idea was to extract the languages of the three models and compare it with the language of the specification, with regard to these two rules. In order to be able to do this, we used individual words from the names of the model elements as the language symbols of the models. The language of the specification refers to the set of symbols extracted from materials used for student training. Table 5.1 shows the number of common words between

the languages. Notice that the numbers on the diagonal of the table represent the size of corresponding language. If we compare the languages of the models and the specification language (last row or column), we can see that there is almost no difference between the models: models 1, 2 and 3 have 118, 123 and 122 words in common with the specification language, respectively. If we now observe the size of those languages (table diagonal), we can see that they are almost the same in size as well: models 1, 2 and 3 have in total 315, 332 and 334 words, respectively. From this we can conclude that there is no considerable difference in the consistency and quality of naming conventions in three models we used in our experiment.

### 5.1.2 Comparing the models in terms of LOC

The number of lines of code (LOC) used to implement the software is a referent metric for application size (and complexity) in traditional software development. In executable modelling, this metric losses its significance because it can only be used to measure size (and complexity) of processing model. However, the processing layer takes a significant share in the overall complexity of an xtUML model and it is interesting to compare the models used in our study according to the number of lines of code they use to implement the same functionality.

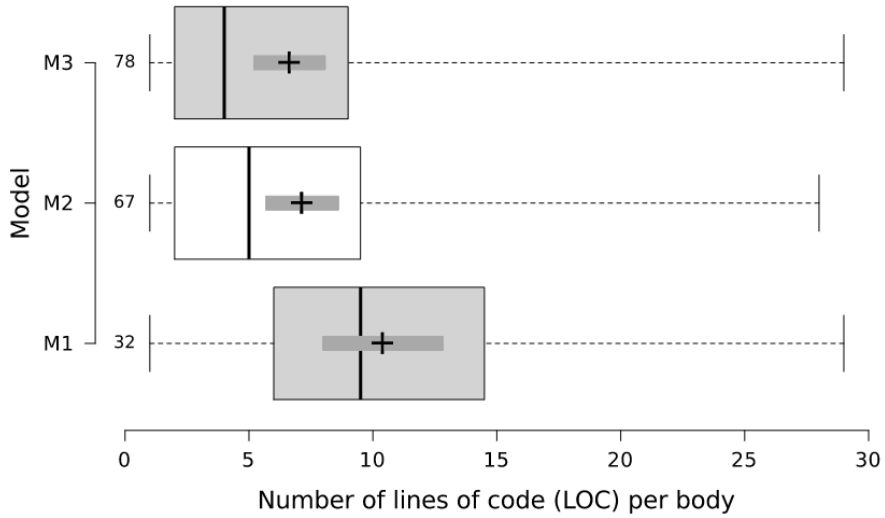



*Figure 5.1. Horizontal distribution of LOC across bodies.*

The difference in internal structure resulted in the fact that three models also differ in their total LOC as well as its distribution (shown in Table 5.2 and figure 5.1). Model 1 implements the desired functionality in only 333 LOC, while Model 3 uses 518 LOC (a difference of 36%). In addition, the total number of non-empty bodies in Model 3 is more than twice the number in Model 1 (78 compared to 32). In comparison, Model 2 has a

similar (but somewhat lower) number of non-empty bodies. Model 3 has has the lowest average LOC per body, while all three have comparable standard deviation.

*Table 5.2. Horizontal distribution of lines-of-code (LOC) across bodies*

| | Model 1 | Model 2 | Model 3 |
|---|---|---|---|
| **Total non-empty bodies** | 32 | 67 | 78 |
| **Total LOC** | 333 | 479 | 518 |
| **Average** | 10,41 | 7,15 | 6,64 |
| **Standard deviation** | 6,72 | 6,02 | 6,37 |

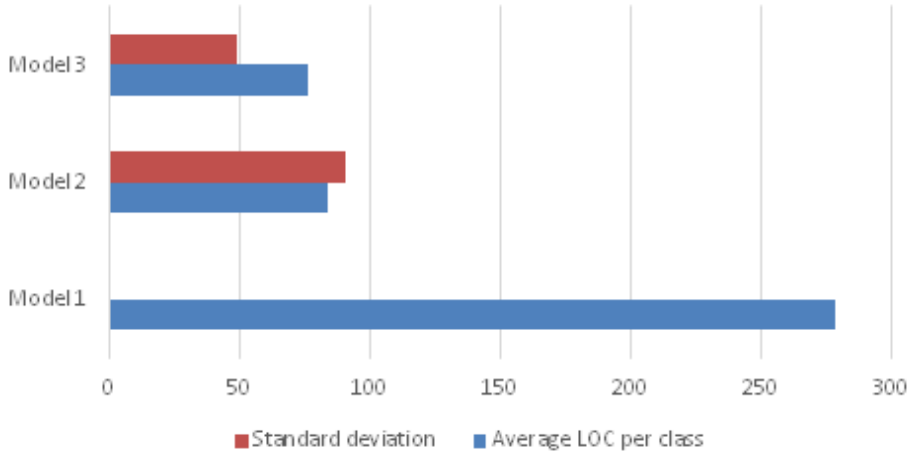



*Figure 5.2. Horizontal distribution of LOC across classes.*

*Table 5.3. Horizontal distribution of lines-of-code (LOC) across classes*

| | Model 1 | Model 2 | Model 3 |
|---|---|---|---|
| **LOC within classes** | 279 | 418 | 456 |
| **Total classes** | 1 | 5 | 6 |
| **Average** | 279 | 83,60 | 76,00 |
| **Standard Deviation** | 0 | 90,88 | 49,25 |
| **Total model LOC** | 333 | 479 | 518 |
| **Class-Total LOC ratio** | 84% | 87% | 88% |

If we observe LOC distribution across classes in table 5.3 and figure 5.2, we can see that Model 3 has lower average and standard deviation than Model 2. This is a consequence of the fact that Model 2 has almost 50% of the total LOC (237 out of total 479) within a single class. Note that Model 1 contains only a single class, which means that it makes no sense to compare it with the other two models, since it obviously has the worst LOC distribution across classes. It is obvious that Model 3 has the best distribution across classes as well.

Considering the functional equivalence and the difference in total LOC of the models, we can conclude that LOC is not a suitable complexity metric for xtUML models. This is true regardless to the results of our experiment and relative understandability of each model.

### 5.1.3 Comparing the models in terms of cyclomatic complexity

Cyclomatic complexity distribution across bodies is shown in table 5.4. For this purpose, we used the sum of decisions and calls within a single body (see section 3.1.6 for details). Model 3 has the largest total number of decisions and calls, but it also has the best distribution (the lowest average value and standard deviation) of those decision and calls across bodies. It is interesting to note that Model 2 and Model 3 have a relatively large number of bodies without any decisions or calls. Partially, this can be explained by a large number of *getter* methods which are used to abstract away selections across relations in a class model.

*Table 5.4. Horizontal distribution of cyclomatic complexity across bodies*

| | **Model 1** | **Model 2** | **Model 3** |
|---|---|---|---|
| Total non-empty bodies | 32 | 67 | 78 |
| Total decisions and calls | 161 | 221 | 239 |
| Average | 5,03 | 3,30 | 3,06 |
| Standard deviation | 4,79 | 4,56 | 4,18 |
| Bodies with zero decisions and calls | 4 | 21 | 25 |

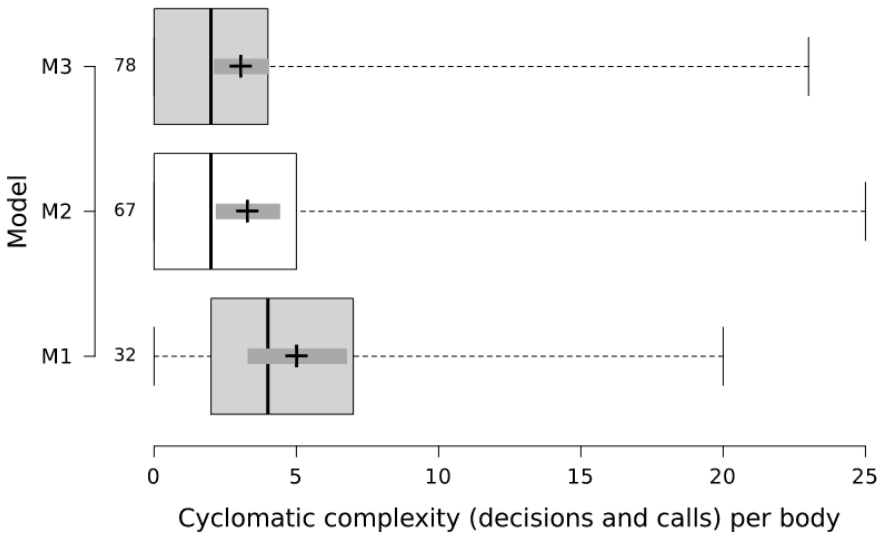



*Figure 5.3. Horizontal distribution of cyclomatic complexity (decisions and calls) across bodies.*

We can also analyze cyclomatic complexity distribution across classes, as shown in table 5.5 and figure 5.4. In this case, Model 2 has the lowest average cyclomatic complexity, but its distribution is worse than in Model 3 which has lower standard deviation. Model 1 has only one class and has all its complexity in one class so it obviously has the worst per-class distribution.

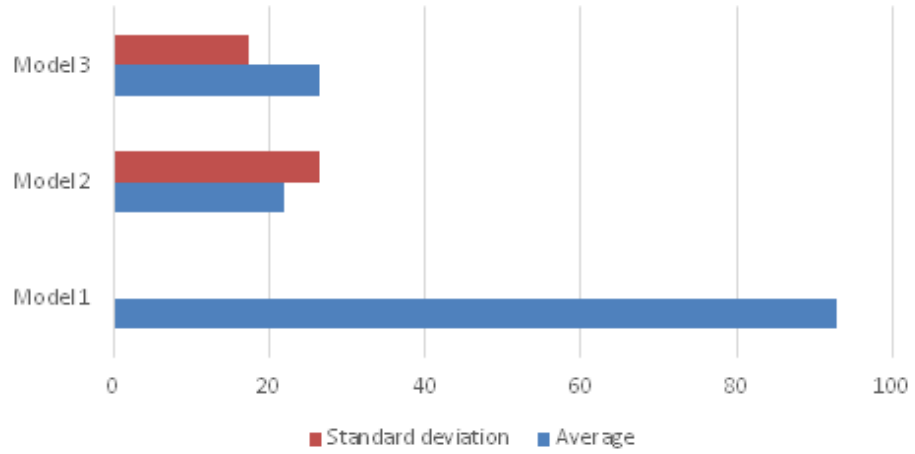



*Figure 5.4. Horizontal distribution of cyclomatic complexity (decisions and calls) across classes.*

*Table 5.5. Horizontal distribution of cyclomatic complexities across classes*

| | **Model 1** | **Model 2** | **Model 3** |
|---|---|---|---|
| **Complexity within classes (body + SM)** | 93 | 110 | 159 |
| **Total classes** | 1 | 5 | 6 |
| **Average** | 93 | 22 | 26,5 |
| **Standard deviation** | 0 | 26,34 | 17,47 |
| **Total complexity** | 136 | 161 | 208 |
| **Class-total ratio** | 68% | 68% | 76% |

In addition to determining how certain model metrics are distributed among the elements of the same type – *horizontal* distribution, we can also analyze the distribution of complexity metrics across different modelling layers – *vertical* distribution of complexity. Table 5.6 and figure 5.5 show the distribution of cyclomatic complexity across the modelling layers (components, classes, state machines, and processing code) for all three models.

Note that all three models have the same absolute component model complexity because all three versions of the component use the same interface. Model 1 and Model 2 have low or no complexities at certain layers because they are intentionally modelled without them: Model 1 uses a single class as a wrapper for its functionality, while both Model 1 and Model 2 do not use the state machines. The metric that best reflects the difference between vertical complexity distribution among the models is the relative cyclomatic complexity of *Graphical models* that expresses how much of the total cyclomatic complexity is visualized. It includes cyclomatic complexity visually expressed by component, class, and state machine models. As expected, Model 3 has the largest value for this metric which indicates the best cyclomatic complexity vertical distribution.

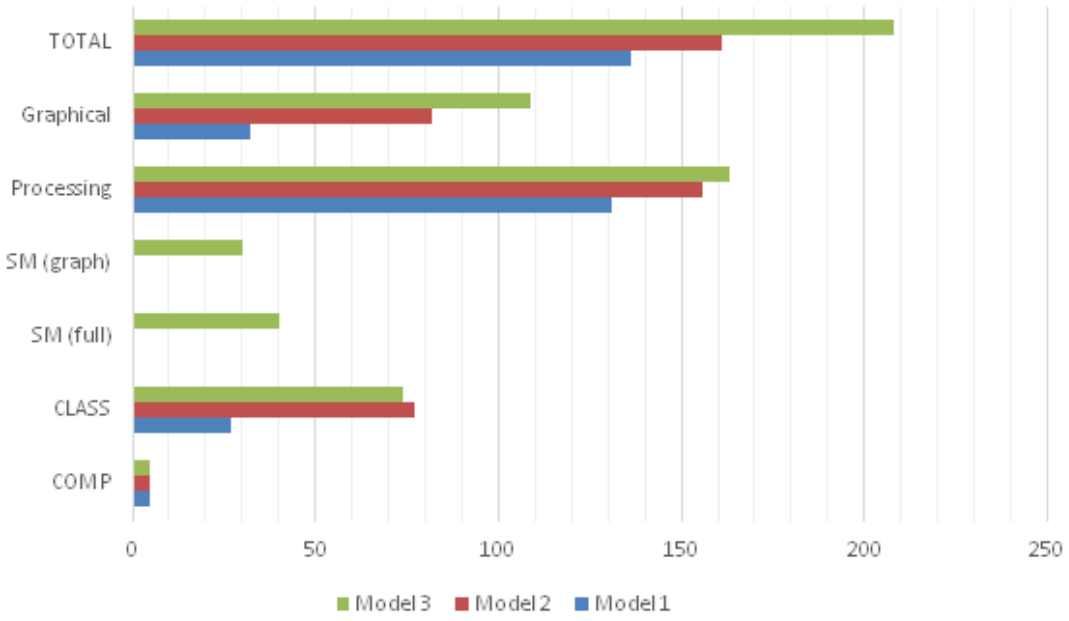



*Figure 5.5. Vertical distribution of cyclomatic complexity.*

### 5.1.4 Comparing model entropies

If we observe an xtUML model as an information source, an entropy of a model can be calculated using classical Shannon's entropy [45]. When applied to xtUML, entropy uses model element types (meta-classes) as symbols. The probability of each symbol is then calculated using the number of elements of a given type. Depending on the distribution we wish to analyse, the population of elements of certain type may be additionally split according to application class or body the element belongs. In such cases, the number of symbols is multiplied and, consequently, entropy increases. In this section we will compare model entropies and their different distributions. For details about model entropy as a measure of complexity please refer to chapter 3.2.

Tables 5.7, 5.8 and 5.9 show the vertical distribution of probabilities and entropies. Figure 5.6 provides information about number of model elements and their vertical distribution across different layers. we can see that the first model has the lowest number of elements while the third has the highest. Since, in vertical distribution, each element represents a symbol, the same is true for their entropies. However, relative difference between entropies of different models is lower than the difference between probabilities. The main reason for this is the logarithmic nature of entropy. Another interesting fact is that, in all three models, a great majority of elements belongs to the BODY layer. This is expected because bodies contain processing model which specifies by far the most information. If we compare the share of BODY elements in the three models we can see that the first model has the highest share of elements belonging to the BODY layer, while the third model has the lowest. Also, the first and the second model have zero elements belonging to the *State machine* layer because

*Table 5.6. Vertical distribution of cyclomatic complexity.*

| | Model 1 | | Model 2 | | Model 3 | |
|---|---|---|---|---|---|---|
| | absolute | relative | absolute | relative | absolute | relative |
| **Component model** | 5 | 3,68% | 5 | 3,11% | 5 | 2,40% |
| **Class model** | 27 | 19,85% | 77 | 47,83% | 74 | 35,58% |
| **State machine model (complete)** | 0 | 0% | 0 | 0,00% | 40 | 19,23% |
| **State machine model (graphical)** | 0 | 0% | 0 | 0% | 30 | 14,42% |
| **Processing model** | 131 | 96,32% | 156 | 96,89% | 163 | 78,37% |
| **Graphical models** | 32 | 23,53% | 82 | 50,93% | 109 | 52,40% |
| **TOTAL** | 136 | 100% | 161 | 100% | 208 | 100% |

*Table 5.7. Vertical distribution of entropy and model element probabilities for the first model*

| **Model layer** | **probability** | | **entropy** | |
|---|---|---|---|---|
| | absolute | relative | absolute | relative |
| COMP | 34 | 0,86% | 0,0864 | 2,01% |
| CLASS | 111 | 2,82% | 0,2214 | 5,14% |
| SM | 0 | 0,00% | 0,0000 | 0,00% |
| BODY | 3551 | 90,29% | 3,5376 | 82,11% |
| OTHER | 237 | 6,03% | 0,4631 | 10,75% |
| **TOTAL** | 3933 | 100% | 4,3086 | 100% |

they do not use state machine models at all. Although unexpected, the first model has some elements in CLASS layer because we used a single class as a container for all attributes and operations in the model. The second model has a relatively large share of elements in the class model, mostly to compensate the lack of state machine layer. All this indicates that the third model has the best distribution of entropy while the first model has the worst.

Tables 5.10, 5.11, 5.12 show the distribution of probability and entropy across application classes for all three models. The first model (table 5.10) has only one class, which includes 82,35% (3239 out of 3933) elements in that model and has entropy of 3,4124. The second model (table 5.11) uses 5 different classes that make 81,02% (3356 out of total 4142) elements in the model and has the overall entropy of 5,0598. Note that almost 50% of model elements belong to the class *Number* which indicates unequal distribution among the classes. The third model (table 5.12) uses 6 different classes that include 81,16% (3747 out of total 4617) elements in the model and has entropy of 5,7090. Similarly as in the second model, the class *Number* contains the largest number of elements (25,51%), but the distribution across classes is much better.

*Table 5.8. Vertical distribution of entropy and model element probabilities for the second model*

| **Model layer** | **probability** | | **entropy** | |
|---|---|---|---|---|
| | absolute | relative | absolute | relative |
| COMP | 35 | 0,85% | 0,0856 | 1,76% |
| CLASS | 305 | 7,36% | 0,5270 | 10,84% |
| SM | 0 | 0,00% | 0,0000 | 0,00% |
| BODY | 3534 | 85,32% | 3,7690 | 77,52% |
| OTHER | 268 | 6,47% | 0,4806 | 9,88% |
| **TOTAL** | 4142 | 100% | 4,8622 | 100% |

*Table 5.9. Vertical distribution of entropy and model element probabilities for the third model*

| **Model layer** | **Probability** | | **Entropy** | |
|---|---|---|---|---|
| | absolute | relative | absolute | relative |
| COMP | 34 | 0,74% | 0,0753 | 1,50% |
| CLASS | 313 | 6,78% | 0,5047 | 10,04% |
| SM | 119 | 2,58% | 0,2373 | 4,72% |
| BODY | 3876 | 83,95% | 3,7611 | 74,79% |
| OTHER | 275 | 5,96% | 0,4507 | 8,96% |
| **TOTAL** | 4617 | 100% | 5,0290 | 100,00% |

*Table 5.10. Horizontal distribution of entropy and model element probabilities per classes (first model)*

| Class name | Probability | | Entropy | |
|---|---|---|---|---|
| | absolute | relative | absolute | relative |
| Calculator | 3239 | 82,35% | 3,4124 | 100% |
| TOTAL (classes) | 3239 | 82,35% | 3,4124 | 100% |
| TOTAL (model) | 3933 | 100% | N/A | N/A |

*Table 5.11. Horizontal distribution of entropy and model element probabilities per classes (second model)*

| **Class name** | **Probability** | | **Entropy** | |
|---|---|---|---|---|
| | absolute | relative | absolute | relative |
| Digit | 185 | 4,47% | 0,3917 | 7,74% |
| Error | 32 | 0,77% | 0,0868 | 1,71% |
| Number | 1967 | 47,49% | 2,5610 | 50,61% |
| Operation | 652 | 15,74% | 1,0975 | 21,69% |
| Screen | 520 | 12,55% | 0,9228 | 18,24% |
| **TOTAL (classes)** | 3356 | 81,02% | 5,0598 | 100% |
| **TOTAL (model)** | 4142 | 100% | N/A | N/A |

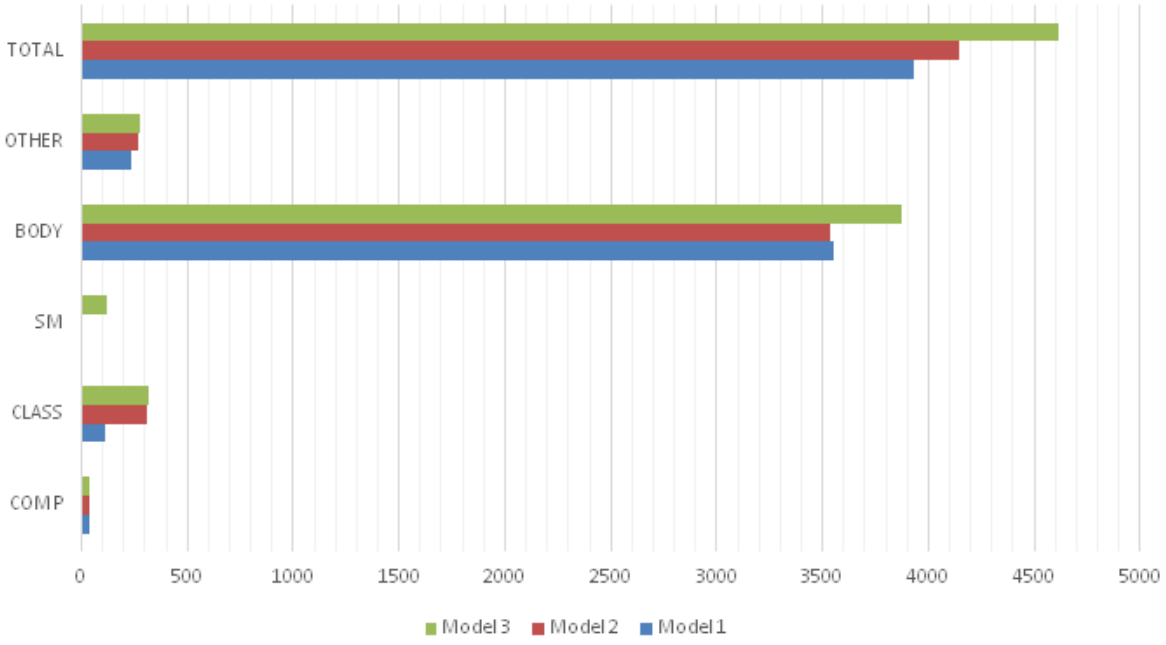



*Figure 5.6. Vertical distribution of model elements (used to calculate their probabilities).*

*Table 5.12. Horizontal distribution of entropy and model element probabilities per classes (third model)*

| **Class name** | **Probability** | | **Entropy** | |
|---|---|---|---|---|
| | absolute | relative | absolute | relative |
| CalculatedNumber | 798 | 17,28% | 1,1401 | 19,97% |
| Digit | 185 | 4,01% | 0,3577 | 6,27% |
| EnteredNumber | 260 | 5,63% | 0,5087 | 8,91% |
| Number | 1178 | 25,51% | 1,6309 | 28,57% |
| Operation | 881 | 19,08% | 1,3305 | 23,31% |
| Screen | 445 | 9,64% | 0,7411 | 12,98% |
| TOTAL (classes) | 3747 | 81,16% | 5,7090 | 100% |
| TOTAL (model) | 4617 | 100% | N/A | N/A |

Entropies presented in the tables do not include all model elements but only those that can be assigned to a class which is around 81% of elements for all three models. As already indicated in section 3.2.3, the total entropy in horizontal distribution is larger then the entropy in vertical distribution, although it includes only 81% of total model elements (only those that can be associated to a class). The main reason for this are the additional symbols introduced by splitting element population by application classes.

Tables 5.13, 5.14 and 5.15 show the distribution of probabilities and entropies across the bodies for all three models. Since there is much more bodies than classes it was not practical to present the values for each body. Instead, we are presenting the average and the standard deviation value together with some other interesting statistical information. In addition, figure 5.7 shows distribution of model elements across bodies for all three models.

*Table 5.13. Horizontal distribution of entropy and model element probabilities per bodies (first model)*

| | Meta-model instances | Probability | Entropy |
|---|---|---|---|
| **Average** | 69,6275 | 1,77% | 0,1461 |
| **Std.dev.** | 76,3814 | 1,94% | 0,1480 |
| **Total (bodies)** | 3551 | 90,29% | 7,4488 |
| **Total (model)** | 3933 | 100% | N/A |
| **Body number** | 51 | | |

*Table 5.14. Horizontal distribution of entropy and model element probabilities per bodies (second model)*

| | Meta-model instances | Probability | Entropy |
|---|---|---|---|
| **Average** | 41,0930 | 0,99% | 0,0928 |
| **Std.dev.** | 46,8900 | 1,13% | 0,0933 |
| **Total (bodies)** | 3534 | 85,32% | 7,9849 |
| **Total (model)** | 4142 | 100% | N/A |
| **Body number** | 86 | | |

Although having the largest number of elements in the bodies (TOTAL(bodies) row), the third model has the lowest average number of elements per body and the lowest median value which indicates the best distribution. The largest body has a total of 291 elements and can be found in first model.

Note that the total entropy of all bodies is significantly larger than the total entropy when distributed across classes (see tables 5.10, 5.11 and 5.12). This is an expected result since, by differentiating model elements by bodies, we introduced more distinguished symbols and, consequently, increased overall entropy. For example, instead of stating that a given *select* statement belongs to a class, we are now interested in which body that statement is defined. Since there is an order of magnitude more bodies than classes, we have much more distinguished symbols which results in larger entropy.

*Table 5.15. Horizontal distribution of entropy and model element probabilities per bodies (third model)*

| | Meta-model instances | Probability | Entropy |
|---|---|---|---|
| **Average** | 39,1515 | 0,85% | 0,0807 |
| **Std.dev.** | 50,3535 | 1,09% | 0,0900 |
| **Total (bodies)** | 3876 | 83,95% | 7,9863 |
| **Total (model)** | 4617 | 100% | N/A |
| **Body number** | 99 | | |

Using model element types (meta-classes) defined in the meta-model as symbols and

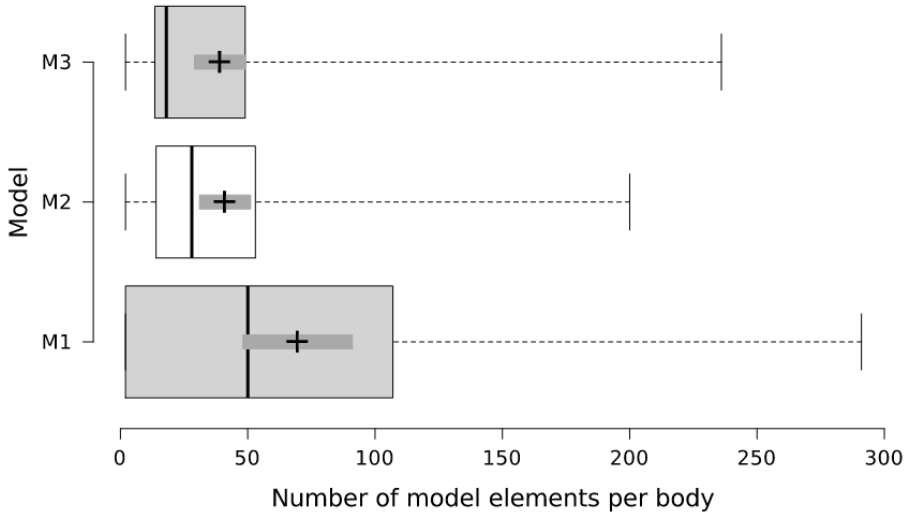



*Figure 5.7. Horizontal distribution of model elements across bodies (including empty ones).*

their cardinality (the number of instances) for calculation of application model entropy is a novel approach presented in this paper. Similarly as with the other metrics, the third model exhibits the best distribution of entropy complexity metric while the first model exhibits the worst. This is true for all distributions, vertically across different model layers as well as horizontally, across model classes and bodies.

### 5.1.5 Comparing the data type complexity of models

As already described in section 3.3.2, data type complexity measures complexity of an xtUML class definition as the number of class members and relations it is involved in. The complexity of data structures is only taken into account if it is used as a type for a member of a class. In this section we will compare the three models with regard to the total data type complexity and its distribution across classes. Since data type complexity primarily affects classes, other layers and distributions will not be considered.

*Table 5.16. Data type complexity for second model*

| **Class** | **Data type complexity** | |
|---|---|---|
| | absolute | relative |
| Digit | 8 | 25% |
| Error | 2 | 6,25% |
| Number | 11 | 34,375% |
| Operation | 5 | 15,625% |
| Screen | 6 | 18,750% |
| **TOTAL** | 32 | 100% |
| **Average** | 6,400 | 20% |
| **Std.dev.** | 3,362 | 10,505% |

*Table 5.17. Data type complexity for the third model*

| Class | Data type complexity | |
|---|---|---|
| | absolute | relative |
| CalculatedNumber | 2 | 6,061% |
| Digit | 8 | 24,242% |
| EnteredNumber | 1 | 3,03% |
| Number | 11 | 33,333% |
| Operation | 6 | 18,182% |
| Screen | 5 | 15,152% |
| **TOTAL** | 33 | 100% |
| **Average** | 5,5 | 16,667% |
| **Std.dev.** | 3,7283 | 11,298% |

Tables 5.16 and 5.17 show the number of primitive values and relations across classes for the second and the third model, respectively. Since those models have very similar class models, they do not differ much regarding the number or distribution of data type complexity across classes. Figure 5.9 shows the class model used in the second model. Otherwise very similar to the second one, the third model does not use a separate class for errors and introduces two subclasses of class *Number*: *EnteredNumber* and *CalculatedNumber*. Figure 5.8 summarizes average and standard deviation values for all models.

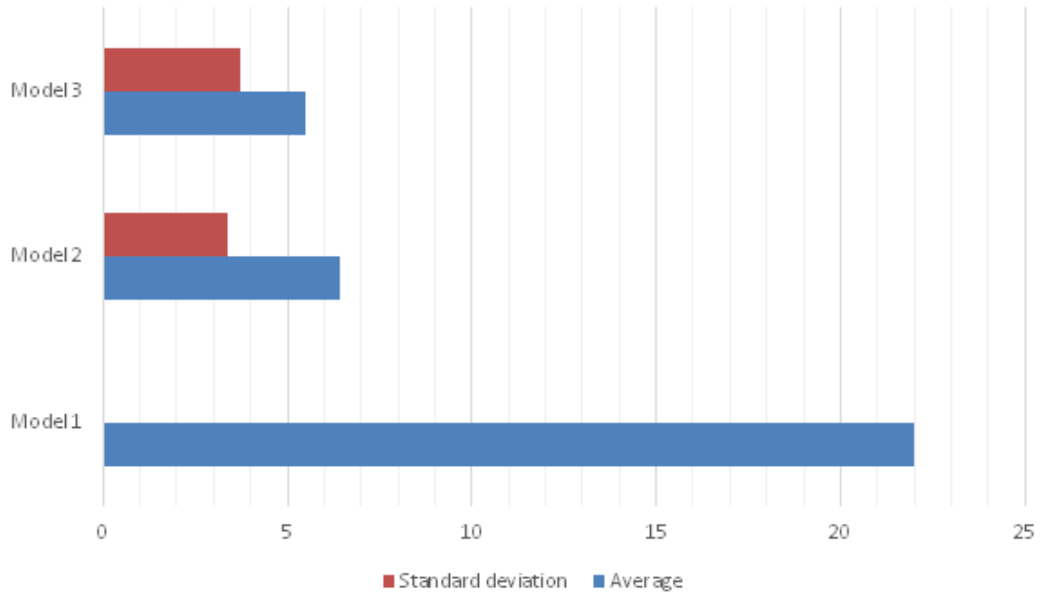



*Figure 5.8. Horizontal distribution of data type complexity across classes.*

The first model, however, uses only a single class and has the total data type complexity of 22 which includes primitive and structured member fields but no relations. That model, unlike the second and the third one, does not use class model relations but rather utilizes data structures to express relations between data items.

On the figure 5.10, we can see that *Calculator* class has members of type *ShowableNum-*

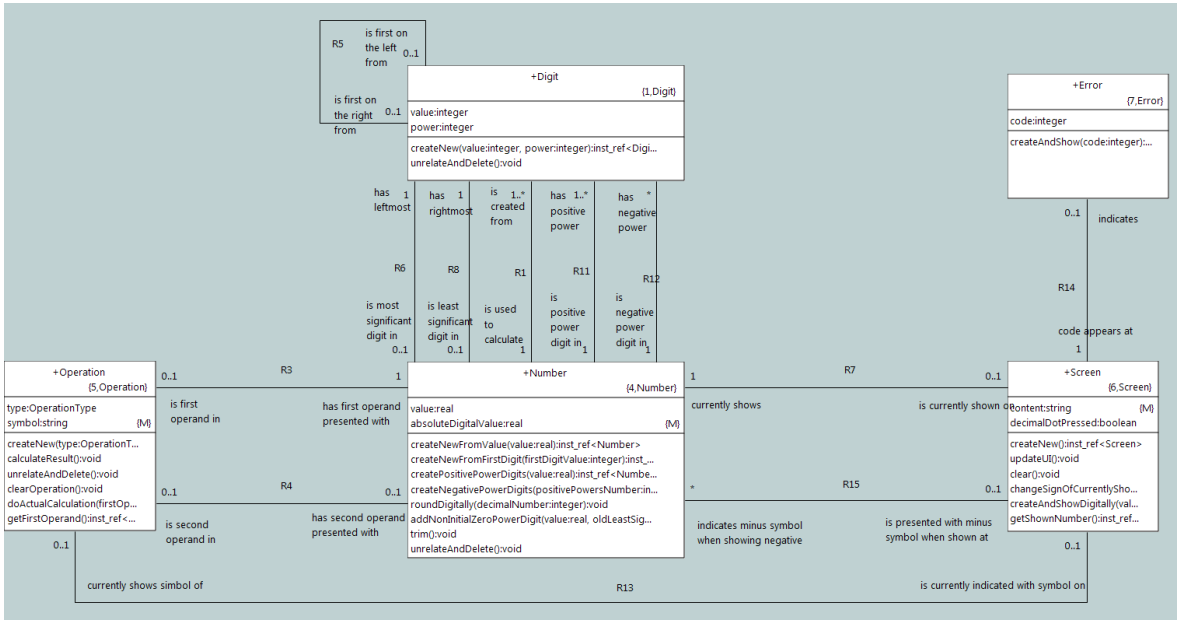



*Figure 5.9. A class model used in the second model.*

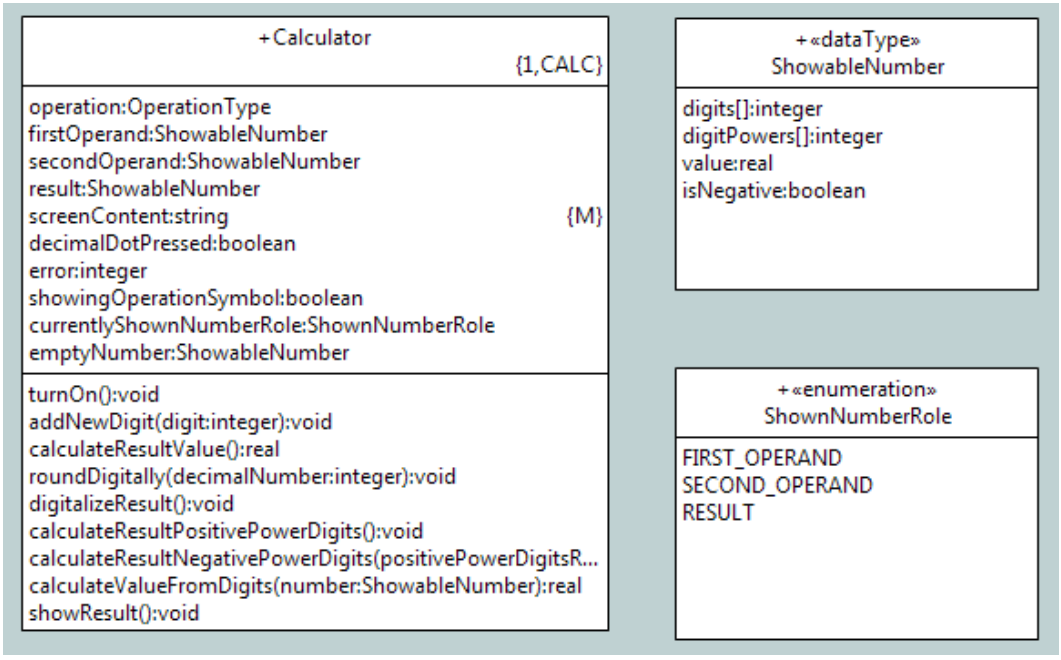



*Figure 5.10. A single class, data structure and an enumeration used for data modelling in the first model.*

*ber* to store operands and the result of an operation. In total, the first model has the lowest data type complexity, but, since it is concentrated in a single class, it has also the worst distribution.

### 5.1.6 Comparing the data flow complexity of models

Data flow complexity of an xtUML model measures the complexity of data processing performed at runtime. We are using the number of variable (re)definitions found in the processing model to quantify runtime data processing. By definition, data flow complexity is measured only in processing model, but other xtUML layers partially visualize data flow complexity from processing layer. We will now compare the three models with regard to the total data flow complexity and its distributions.

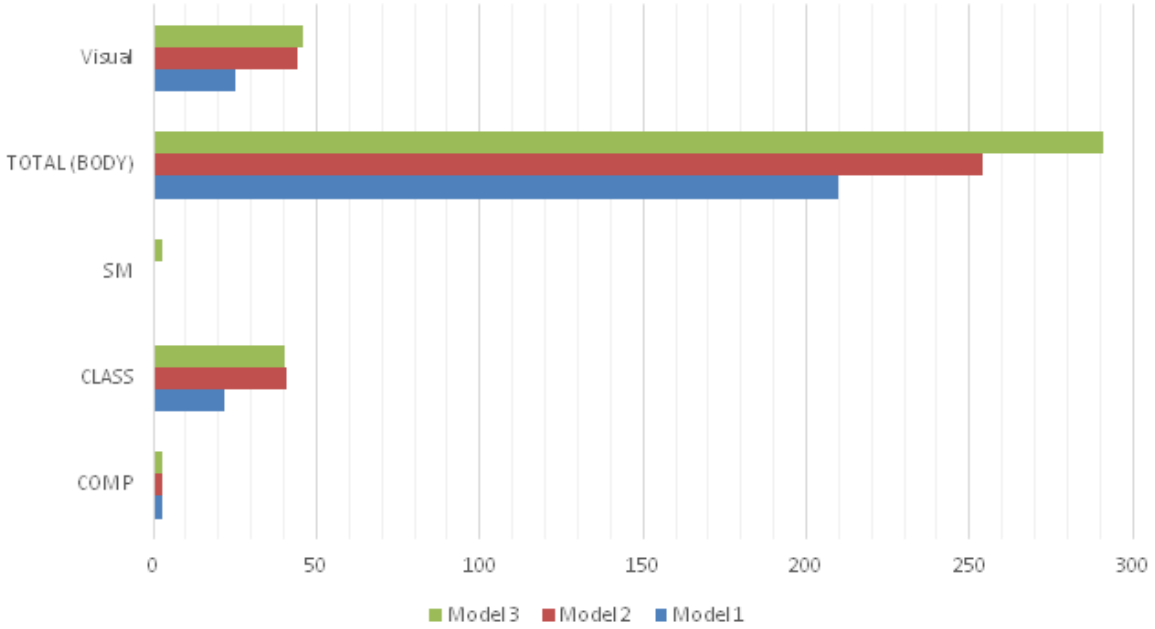



*Figure 5.11. Vertical distribution of data flow complexity for all three models*

Table 5.18 and figure 5.11 show the vertical distribution of the data flow complexity across different xtUML layers for all three models. Notice that the *BODY* layer contains complete data flow complexity of the model. The first model has the lowest absolute data flow complexity (210), while the third one has the highest (294). However, if we observe how much of data flow complexity is visualized across graphical models (*Visual* row in the table), we will see that the the second model has the highest share of data flow complexity visualized with component, class or state machine models (17,32%). This is somewhat surprising, especially taking into account that second model does not use state machine models at all.

The first model, as expected, has the lowest part of data flow complexity visualized. The last row in the table (*BODY/LOC*) displays the *density* of definitions across the lines of code. Again, the first model has the highest density of definitions per lines of code while

*Table 5.18. Vertical distribution of data flow complexity for all three models*

| | **Model 1** | | **Model 2** | | **Model 3** | |
|---|---|---|---|---|---|---|
| | absolute | relative | absolute | relative | absolute | relative |
| **COMP** | 3 | 1,43% | 3 | 1,18% | 3 | 1,03% |
| **CLASS** | 22 | 10,48% | 41 | 16,14% | 40 | 13,75% |
| **SM** | 0 | 0,00% | 0 | 0,00% | 3 | 1,03% |
| **BODY** | 210 | 100% | 254 | 100% | 291 | 100% |
| **Visual** | 25 | **11,90%** | 44 | **17,32%** | 46 | **15,81%** |
| **LOC** | 333 | | 479 | | 518 | |
| **BODY/LOC** | **0,6306** | | **0,5303** | | **0,5618** | |

the second has the lowest. Although the difference between the second and the third model is not substantial, the distribution of data flow complexity indicates the second model as the one with the best distribution. This was not the case with other metrics.

*Table 5.19. Horizontal distribution of data flow complexity across classes for the second model.*

| **Class** | **Model 2** | |
|---|---|---|
| | absolute | relative |
| Digit | 13 | 5,12% |
| Error | 4 | 1,57% |
| Number | 138 | 54,33% |
| Operation | 43 | 16,93% |
| Screen | 35 | 13,78% |
| **TOTAL (classes)** | 233 | 91,73% |
| **TOTAL (model)** | 254 | 100% |
| **Average** | **46,600** | **18,35%** |
| **Std.dev.** | **53,491** | **21,06%** |

*Table 5.20. Horizontal distribution of data flow complexity across classes for the third model.*

| **Class** | **Model 3** | |
|---|---|---|
| | absolute | relative |
| CalculatedNumber | 57 | 19,39% |
| Digit | 13 | 4,42% |
| EnteredNumber | 20 | 6,8% |
| Number | 84 | 28,57% |
| Operation | 61 | 20,75% |
| Screen | 30 | 10,2% |
| **TOTAL (classes)** | 265 | 90,14% |
| **TOTAL (model)** | 294 | 100% |
| **Average** | **44,167** | **15,02%** |
| **Std.dev.** | **27,535** | **9,37%** |

Tables 5.19 and 5.20 show the horizontal data flow distribution across classes for the second and the third model. When calculating the distribution of data flow complexity across classes, we took into account the data flow complexity from all the bodies within a class, but not the complexity visualized in the class model, because it is already taken into account in bodies. We can see that the third model has lower average and standard deviation which implies better distribution. The first model has the worst distribution of data flow complexity since it uses a single class which holds 188 (or 89,524%) out of total 210 definitions. Figure 5.12 summarizes horizontal distribution of data flow complexity across classes.

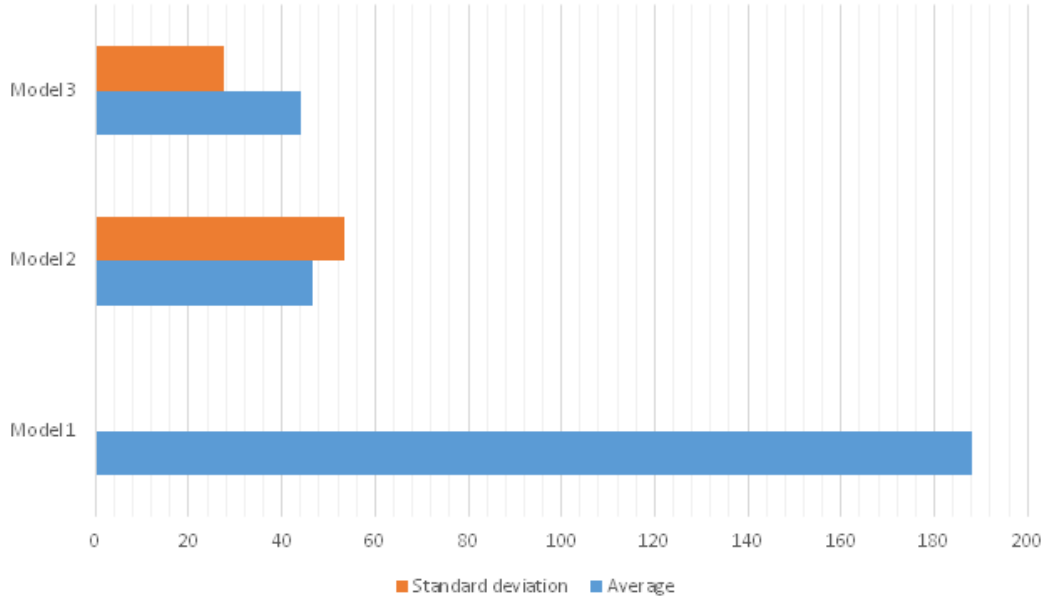



*Figure 5.12. Summary of horizontal distribution of data flow complexity across classes.*

Table 5.21 and figure 5.13 show the horizontal distribution of data flow complexity across bodies for all three models. As expected, the first model has the highest average number of definitions per body (6,3636) and the highest standard deviation value (4,1745) which indicates the worst distribution. The second and the third model have very similar distributions and no firm conclusion can be made which one is better.

*Table 5.21. Horizontal distribution of data flow complexity across bodies for all three models*

| | **Model 1** | **Model 2** | **Model 3** |
|---|---|---|---|
| **Total non-empty bodies** | 33 | 68 | 81 |
| **Total data flow complexity** | 210 | 254 | 291 |
| **Average** | 6,3636 | 3,7353 | 3,5926 |
| **Std. dev.** | 4,1745 | 3,4408 | 3,9962 |
| **Bodies with zero data flow complexity** | 1 | 2 | 4 |

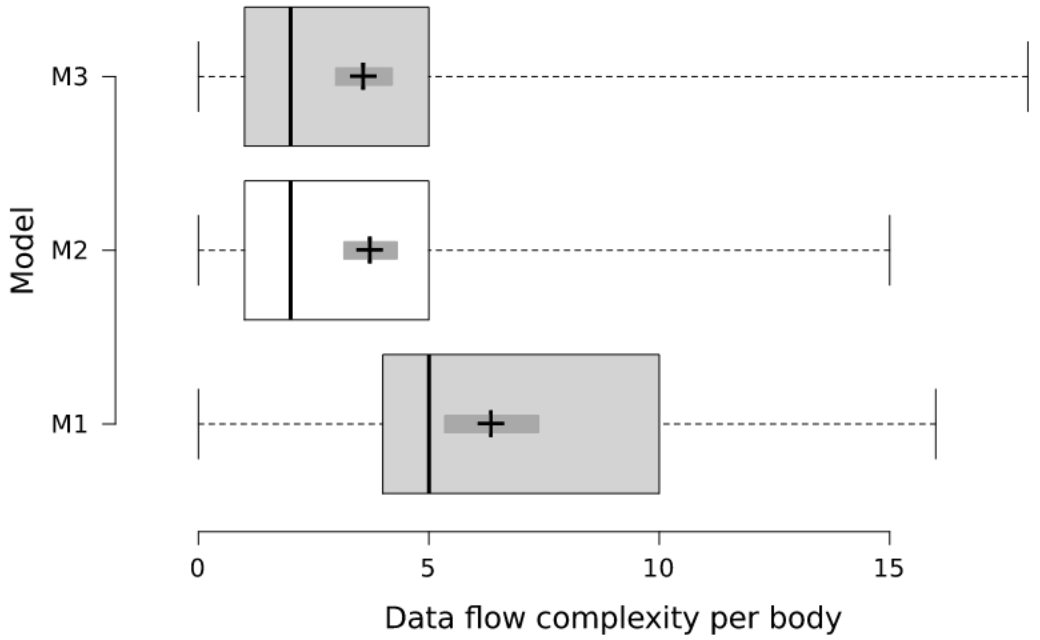



*Figure 5.13. Horizontal distribution of data flow complexity across bodies.*

To conclude, in all three distributions that we observed, the first model was the worst. However, it is not clear which of the models has the best distribution. The second model visualizes more of the data flow complexity than the third one (see table 5.18), but the third model seems to be better when it comes to distribution across classes (see tables 5.19 and 5.20). Both models have very similar horizontal distribution across bodies so no firm conclusion can be made which one is better.

### 5.1.7 Conclusion on model comparison

The three functionally equivalent models are intentionally modelled in different ways in order to achieve different complexity distributions. In some of those models we intentionally limited the usage of certain xtUML layers and directly influenced vertical complexity distributions. Regarding the horizontal distribution, we strictly followed the *rule of 30* for the number of LOC within a single body. The size of models was not artificially altered and is result of distribution restrictions, functional requirements of a common test suite, and the best practices of the design approach used in given model.

We first compared the *languages* (as defined by Laitinen [5]) used by three models and concluded that there is no significant difference between them. In this way we eliminated quality and consistency of naming conventions as a factor that may influence model understandability. As a consequence, since the implementation language of the models and functional size are already eliminated as factors, we ensured that the only factor that varies in the three models is the distribution of complexity.

This was followed by comparison of the lines of code (LOC) used to implement pro-

cessing within the models. Although functionally equivalent, there is a significant difference between the three models, which confirms LOC as poor xtUML model complexity metric. For horizontal distribution of processing code across bodies, we strictly followed the *rule of 30* [56] for the number of LOC in action bodies in all three models. Somewhat surprising, despite the difference in LOC, more abstract models had better *per-body* horizontal complexity distribution. It seems that new levels of abstraction influence, not only vertical complexity distribution (as it is expected), but also horizontal distribution to the extent that it compensates for the greater number of LOC. For example, in xtUML models that use class relations, it is trivially easy to abstract relation navigation to getter operations, at least in cases when we navigate over single-hop chains. The same getter abstraction can actually be implemented without classes and relations, but not as easy. In addition, an xtUML state machine introduces two additional types of bodies (states and transition bodies) which also improves the horizontal, per-body distribution.

Cyclomatic, entropy and data type complexity metrics follow the same interesting complexity distribution pattern: the first model has the lowest absolute complexity, but also the worst distribution, while the third has highest absolute complexity, but the best distribution. The data flow complexity metric is similar, with an exception that no firm conclusion can be made whether the second or the third model has the best distribution. Such pattern directly confronts the absolute (size-based) and the relative (distribution-based) metrics. The results of our experiment will compare those metric types and their influence on xtUML model understandability.

## 5.2 Study subjects

### 5.2.1 Preparation

The subjects of the experiment were third-year, undergraduate computer science students enrolled in the Systems Analysis and Design course at the University of Split, Croatia, during the spring semester of 2015. This was their first exposure to MDE, but it is important to note that they already had courses in Software Engineering and Object-Oriented programming, in which they gained experience with the general ideas and techniques behind software modelling. In addition, before participating in the experiments, the students were exposed to the following treatment:

1. *Theoretical lecture*: the students were given a two-hour theoretical course explaining the basic ideas, principles, and assumptions behind the MDE and xtUML in particular. Materials used in this lecture (in Croatian) can be found in *course_materials\teorija* directory on repository [55]. In the end, student knowledge was tested with a short test comprised out of 20 multiple-choice, multiple-answer questions.

2. *Practical lab assignments*: in the following three weeks, the students have participated in three two-hour lab assignments which were designed to reinforce the theoretical concepts, as well as to provide the students with practical experience of creating and understanding models. They also gained practical experience working with an open-source xtUML tool – Bridgepoint[1]. Materials used in labs (in Croatian) can be found in *course_materials\lab* directory on repository [55].

In addition, during the lab assignments the students were introduced to the main requirements and the common test suite (domain knowledge), which were used for all three applications.

Since we used three structurally different applications, we have divided the students into three groups, where each student group was assigned to one application. When creating these groups, we had to assess student abilities, in order to create groups with equal student abilities. For this reason, each student was tested with a:

1. *Theoretical test*: After the theoretical lecture, each student was given a short test with 20 multiple-choice questions (available in Croatian at *course_materials\teorija* directory on repository [55]).

2. *Domain test*: After completing all the labs, each student was given a short domain-knowledge test with 11 multiple-choice questions (available in Croatian at *course_materials\domena* directory on repository [55]).

After ordering students according to their success on the tests, the students were randomly distributed into three groups in a way that the total ability of each group is approximately the same. To verify that there is no significant difference between the mean ability within the groups, we performed the ANOVA variance analysis for each of the ability indicators:

- Theory Correct Answers (TCA) – The number of total correct answers achieved in the theoretical xtUML exam.
- Theory Points (TP) – The number of total correct answers minus the number of incorrect answers achieved in the theoretical xtUML exam.
- Domain Correct Answers (DCA) – The number of total correct answers achieved in the domain exam.
- Domain Points (TP) – The number of total correct answers minus the number of incorrect answers achieved in the domain xtUML exam.

Table 5.22 shows the number of students in each group and their average ability as measured by the tests. None of the ability indicators showed significant difference between the groups at $p = 0.05$ significance level.

[1] https://xtuml.org/download/

*Table 5.22. Distribution of student ability across groups with results of ANOVA analysis*

| | **Theory correct answers (TCA)** | | **Theory points (TP)** | | **Domain correct answers (DCA)** | | **Domain points (DP)** | | **Average mark (AM)** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | avg | std | avg | std | avg | std | avg | std | avg | std |
| **G1** | 21,71 | 4,341 | 14,24 | 5,562 | 10,53 | 2,611 | 8,00 | 4,749 | 3,24 | 1,254 |
| **G2** | 21,10 | 4,833 | 14,65 | 7,147 | 12,11 | 1,487 | 10,79 | 2,275 | 3,46 | 0,826 |
| **G3** | 21,05 | 3,845 | 14,40 | 7,344 | 11,50 | 2,066 | 9,36 | 3,522 | 2,85 | 0,708 |
| **df bg** | 2 | | 2 | | 2 | | 2 | | 2 | |
| **df wg** | 54 | | 54 | | 49 | | 49 | | 45 | |
| **F** | 0,126 | | 0,018 | | 2,700 | | 2,743 | | 1,729 | |
| **p** | 0,882 | | 0,983 | | 0,077 | | 0,074 | | 0,189 | |
| **SGN (0,05)** | NO | | NO | | NO | | NO | | NO | |

### 5.2.2 Data collection

Data collection was done through online questionnaires, one per group. Questions in all three questionnaires were the same and in the same order, while the answers differed depending on the model explored by the group. Most of the questions were multiple-choice questions with multiple correct answers. Each question indicated whether there is a single or multiple correct answers. Although having different answers, questions in all three questionnaires had exactly the same number of correct answers, in exactly the same order (for example *a*, *d* and *e* answers were correct answers for the 6th question in all 3 questionnaires). This was done in order to minimize the differences between the questionnaires.

In addition, the students were presented one question at a time, and were instructed to keep the questionnaire on the question whose answer they were trying to answer, while exploring the code base. We have collected the following data:

- Correct answers (*CA*): The total number of correct answers. If a question has multiple correct answers, each correct answer is taken into account.
- Incorrect answers (*IA*): The total number of incorrect answers. If a question is a multiple-choice question, each incorrect answer is taken into account.
- Total number of points (*P*): The total number of correct answers reduced by the the total number of incorrect answers: $P = CA - IA$
- Total Time (*TT*): The total time for completing the whole questionnaire.
- Correct answers per minute (*CAPM*): The ratio between the total number of correct answers and the total time, expressed in minutes.

- Points per minute (*PPM*): The ratio between the total number of points and the total time, expressed in minutes.

# 6 Experiment results

The goal of our experiment was to measure the understandability of xtUML models and to check whether there exists a relationship between the distribution of complexity of an xtUML model and its understandability. For this reason, we have set up an experiment that measures the understandability of three different models by testing students divided into three groups. Table 6.1 summarizes the results of the experiment.

*Table 6.1. Summary of experiment results with ANOVA single factor analysis*

| | **CA** | | **P** | | **IA** | | **TT** | | **CAPM** | | **PPM** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | avg | std | avg | std | avg | std | avg | std | avg | std | avg | std |
| **G1** | 20,10 | 4,15 | 13,10 | 7,08 | 7,00 | 3,15 | 36,77 | 6,32 | 0,57 | 0,18 | 0,38 | 0,22 |
| **G2** | 21,80 | 4,99 | 17,10 | 6,34 | 4,70 | 2,18 | 36,07 | 3,64 | 0,62 | 0,17 | 0,49 | 0,21 |
| **G3** | 19,36 | 4,22 | 11,92 | 7,45 | 7,44 | 3,57 | 27,20 | 8,56 | 0,78 | 0,28 | 0,46 | 0,35 |
| **F (2,63)** | 1,718 | | 3,215 | | 4,88 | | 14,869 | | 5,687 | | 0,947 | |
| **p** | 0,188 | | 0,046 | | 0,011 | | 0,00001 | | 0,005 | | 0,393 | |
| **SGN (0.05)** | NO | | YES | | YES | | YES | | YES | | NO | |

Table 6.1 presents the average value with standard deviation, for each group and each result. To check for any significant differences ($p \leq 0.05$) between the groups, we performed the ANOVA variance analysis, with the results also presented in table 6.1. Since ANOVA analysis does not exactly determine between which two sets of data the difference exists, we also performed a series of T-tests between each pair of result samples.

The results show that there is no statistically significant difference in *Correct Answers (CA)* between any of the groups. Although statistically insignificant, the average value is the highest in the second group, while the first and the third group, have almost the same average and standard deviation value.

Regarding *Incorrect Answers (IA)*, the second group has significantly lower average number of incorrect answers when compared to other two groups. The difference between the first and the third group is not significant.

As expected from CA and IA results, the second group had significantly higher average value for *Points* (P). The difference is significant at $p = 0.05$ level when compared to the third group, and only at $p = 0.1$ level when compared with the first group. The difference between results of the first and the third group are not significant.

*Table 6.2. Cohen's* d *factor indicating the effect size of observed differences (S = small, M = medium and L = large effect size)*

| | CA | | P | | IA | | TT | | CAPM | | PPM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **G1G2** | -0,38 | S | -0,61 | M | **0,87** | **L** | 0,14 | S | -0,29 | S | -0,52 | M |
| **G2G3** | 0,55 | M | 0,76 | M | **-0,92** | **L** | **1,33** | **L** | -0,69 | M | 0,10 | S |
| **G1G3** | 0,18 | S | 0,17 | S | -0,13 | S | **1,28** | **L** | **-0,90** | **L** | -0,27 | S |

When we consider the average total time we can see that students in the third group had a surprisingly shorter (25%) average total time (27 minutes when compared to 36 minutes in other two groups). These results are significant even on $p \leq 0.01$ level. The difference between the first and the second group is not significant.

The last two results are calculated and take into account both dimensions of the experiment results, the time and the absolute success. Similarly as for the total time, the third group had the highest average value for correct answers per minute (CAPM). This result is significantly different ($p \leq 0.05$) from the results of the first two groups. The CAPM result for the second group is not significantly greater that those of the first group.

The points per minute (PPM) results do not significantly differ. However, the second and the third group have a somewhat greater average value than the first group.

Table 6.2 shows the Cohen's $d$ value[57] calculated on each group pair for all collected results. It represents the effect size (the order of magnitude) of observed differences between the groups. We can see that the large effect size is observed for total time (TT) between the third group and other two groups, as well as for incorrect answers (IA) between second group and other two groups. For correct answers per minute (CAPM), there is a large effect size between the third and the first group and a medium effect size between the third and the second group.

To summarize the experiment results, the second group had the best absolute results of the experiment ($P$, $IA$), while the third group had the best time ($TT$) and time-relative results ($CAPM$). The first group did not have the best results (not even insignificantly) in any of the categories that we observed. Since $CAPM$ results take into account both dimensions of the model understandability (the absolute success and the time), we can conclude that students from the third group had the best overall success in the experiment.

## 6.1 The relation between experiment results and complexity distribution

If we compare the experiment results with the complexity distribution across models, we can conclude that the complexity metric distribution indeed has significant effect to model understandability.

Group 1, which worked with Model 1 (the model with the worst complexity distribution)

also had the worst results in the experiment. The difference in complexity distributions between Models 2 and 3 is not so obvious, however, for most of the metrics, Model 3 has better distribution. This was reflected in the results of the second and the third group: while the second group was the best in absolute results, the third group was the best in time-relative results which consider both dimensions of our experiment. Because of these results we reject the null hypothesis and accept the alternative one: "Distribution of complexity significantly affects the understandability of xtUML models".

The results of the experiment also indicate that size-based metrics are not well suited as understandability predictors, at least for xtUML models. The model 1 has the lowest absolute value for all the metrics we analysed, but it still exposed the worst understandability in the experiment. As opposed to that, despite the highest absolute values for all the metrics, model 3 has the best understandability. The results indicate that, when faced with multiple design alternatives, we should favour complexity distribution over size.

While the experiment results indicate that there exists a dependency between model understandability and complexity distribution across the model, the experiment itself was not designed to reveal more details about that dependency. In order to do this, a much larger set of xtUML models is required. Traditionally, experimenters use expert opinion to order software applications of different requirements and sizes on an ordinal scale. Such experiment setup is problematic for testing our hypothesis because the functional size and complexity distribution effects are confounded. In our experiment we used only three models but, since semantically equivalent, we were able to eliminate all other factors and focus on the effect of complexity distribution. Because of the low number of observed models, such experiment design prevents us to come up with any prediction model, but it enables us to reliably detect the existence of the dependency.

## 6.2 Threats to validity

In this section we discuss different threats to validity: *internal validity*, *external validity*, and *construct validity* [58].

### 6.2.1 Internal validity

The main threat to internal validity of the experiment is related to the *maturation* effect [59] [60]. The three groups have participated in their experiments in dedicated time slots: the first group from 8:00 to 9:30 in the morning, the second one from 9:30 and 11:00, and the third one from 11:00 till 12:30 (just before lunch). Unfortunately, such experiment setup is generally not ideal, because of potential *hunger/fatigue* effect [61] that the students of the last group may have experienced. This probably resulted in lower accuracy and shorter total time of participants in the group. This systematic error could have been eliminated

by equally distributing students from different groups in different time slots. This error is, to some extent, compensated with the fact that experiment results took time as well as the incorrect answers into account.

The second internal validity threat is related to the *selection* effect [62]. Although we tried to distribute student ability across different groups equally (see table 5.22), it was not possible to perfectly align student's availability with experiment time slots.

The third internal validity threat is related to *experimental mortality* [59]. Table 5.22 shows a satisfactory distribution of ability across groups, but does not take into account students that were absent from theory or domain classes (and tests) but still participated in the final experiment. It was necessary that these students participate in the experiment because the experiment was conducted in the scope of a university class, so it would be unfair to deny the learning experience to students that were not able to attend all lectures. The absence distribution is shown in table 6.3. When compared with other groups, we can see that group 3 had a significantly larger share of students that did not participate in the domain exercise. To check for any negative effects, we performed an additional ANOVA and t-test analysis, this time excluding the students that did not attend the domain exercise. The results of the analysis show that this did not have any effect on the experiment results.

*Table 6.3. Distribution of absence across the groups*

| Absent from\Group | **G1** | **G2** | **G3** | **TOTAL** |
|---|---|---|---|---|
| **Theory** | 4 | 0 | 5 | 9 |
| **Domain** | 2 | 1 | 11 | 14 |
| **Both** | 1 | 0 | 3 | 4 |
| **Total students** | 21 | 20 | 25 | 66 |

It is important to emphasize that all three internal validity threats had a negative effect on the third group. Despite this, the third group presented the best overall experiment results.

### 6.2.2 External validity

There are several concerns regarding external validity. First, the experiment subjects were third year computer science students, so the results may not be the same if the subjects were trained professionals with experience in xtUML modelling. During their previous education, students were trained in the structured programming approach (mostly used in Model 1) and class modeling approach (used in Model 2 and Model 3). However, students had little previous exposure to state machines, which were used only in the third model.

The second concern is the representativeness of models used as experiment objects. All three models are relatively simple and do not represent typical real world examples. The main reason for such model selection was the limited time required to understand the application requirements and the limited time of the experiment. The simple calculator application

used in the experiment was suitable as the experiment object because it was relatively easy to add as much complexity as necessary. Since this was a relatively small application, we consider that the full abstraction potential of the xtUML methodology was not taken advantage of.

### 6.2.3 Construct validity

The results of the experiment show that complexity distribution affects the xtUML model understandability. However, we should be careful with the possible implications of that conclusion. In terms of vertical and horizontal complexity distribution, the three models can be ordered: Model 1, Model 2, Model 3, where Model 1 has the worst and Model 3 the best horizontal and vertical complexity distribution. However, since those models differ in horizontal as well as in vertical complexity distribution, we cannot attribute the increased understandability only to better vertical complexity distribution. This means that in our case, the horizontal and vertical complexity distribution are confounded and no definite conclusion can be made if we observe them separately. In other words, we do not know if Model 3 has the best understandability because of the new abstraction layers or simply because the largest number of bodies (subroutines) we used to modularize the processing code into. Both of those approaches are forms of abstractions, but with new layers we are introducing new types of abstractions (such as classes and state machines) while with code modularization we are using subroutines as the only abstraction. We should also keep in mind the possibility that horizontal and vertical complexity distributions are not fully independent. As stated in section 5.1.7, it seems that new levels of abstraction make it easier to make abstractions on subroutine level, i.e. to detect new subroutines. A separate study and an experiment with different set of models is required to investigate isolated effects of vertical and horizontal complexity distribution to xtUML model understandability.

# 7 Conclusion

In this thesis we have investigated the influence of complexity distribution on the understandability of xtUML models. The distribution of the complexity was investigated both in vertical and horizontal directions.

We first adapted three applicable complexity metrics to models that constitute an xtUML model, that is component, class, state machine, and processing model. Three applicable complexity metrics that were adapted are: Cyclomatic complexity, entropy, data type and the data flow complexity . We also specified how to integrate one complexity aspect of the layers into a single metric for total xtUML model. This was done for each of the selected metrics. We have also presented two different ways of measuring complexity distribution: *horizontally*, among elements of the same abstraction level, and *vertically*, among different abstraction levels.

The metrics and their distributions are then used to compare the three semantically equivalent xtUML models that are used as experiment objects. The semantic equivalence of the models is verified by the set of 30 modelled test cases developed specifically for that purpose. The models are intentionally modelled differently, in order to demonstrate different ways to distribute application complexity. The first model used the structured programming paradigm which relies on functional decomposition and data structures. It made no use of object-oriented concepts such as classes, nor it used state machines. The second model heavily relied on classes, but did not use state machines. The third model used all available xtUML models including classes and state-machines. Comparison of model metrics and distributions revealed that most of the metrics follow the same pattern: the first model has the lowest absolute complexity, but also the worst distribution, while the third model has the highest absolute complexity, but the best distribution. Such pattern directly confronted absolute (size-based) and relative (distribution-based) metrics.

In order to verify our hypothesis that complexity distribution has an influence on the understandability of xtUML models, we performed an experiment with student participants in which we evaluated the understandability of the three models. For this purpose we used online questionnaires which took into account not only the absolute success, but also the time spent on the questions. Results indicate that a more uniform complexity distribution positively influences model understandability. When compared with other two groups, students working on the third model needed 25% less time to complete the questionnaire, while

keeping comparable absolute success. Time-relative success confirmed the third group as the most successful. Such results confirmed our hypothesis that complexity distribution significantly influences understandability of xtUML models. It also indicated size-based metrics as poor predictors of xtUML model understandability.

However, since models with better understandability have more uniform horizontal and vertical complexity distributions, the effect cannot be attributed to either of them. We speculate that horizontal and vertical distributions are not independent and that new layers of abstraction introduce and ease detection of new subroutines which reflects on horizontal distribution of complexity. This interesting topic will be in focus of our future work in which we plan additional research and experiments. The goal is to isolate, investigate and compare the effect of each of those distributions on model understandability.

The main scientific contributions of this doctoral dissertation are:

1. **Definition of software metrics in order to quantify the complexity of an xtUML software model**: complexity metrics have been applied to four vertical levels (or sub-models) of an xtUML model. xtUML model complexity metrics are based on cyclomatic, entropy, data type and data flow complexities. These metrics quantify various complexity aspects within the four sub-models.

2. **Integration of complexities of the four sub-models into a single measure of complexity of entire xtUML model and the two ways of measuring the distribution of the complexity across the model**: By measuring complexity distribution, we quantified uniformity of complexity distribution across xtUML sub-models (vertically) and across the elements of the same level (horizontally).

The additional contribution of this dissertation is the formalization of the metrics calculus on xtUML meta-model and methodology for empirical evaluation of understandability of xtUML models.

# Bibliography


[1] T. J. McCabe, "A complexity measure," 1976.

[2] B. Henderson-Sellers and D. Tegarden, "The theoretical extension of two versions of cyclomatic complexity to multiple entrylexit modules," 1994.

[3] S. Shlaer, "The shlaer-mellor method," *Project Technology white paper*, 1996.

[4] OneFact. (2016) Bridgepoint xtuml tool. [Online]. Available: https://www.xtuml.org/download/

[5] K. Laitinen, "Estimating understandability of software documents," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 4, pp. 81–92, 1996.

[6] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architectures*, 2002.

[7] M. G. company. (2013) Bridgepoint from mentor graphics provides agilent gc instrumentation division an efficient methodology for embedded software development. [Online]. Available: https://www.mentor.com/products/sm/news/mentor-bridgepoint-agilent

[8] OMG. (2013) Omg alf standard. [Online]. Available: http://www.omg.org/spec/ALF/

[9] B. Selic, G. Gullekson, and P. T. Ward, *Real-time object-oriented modeling*, 1994.

[10] I. Mattias Mohlin. (2013) Modeling real-time applications in rsarte. [Online]. Available: https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W0c4a14ff363e_436c_9962_2254bb5cbc60/page/RSARTE%20Concepts

[11] OMG. (2016) Semantics of a foundational subset for executable uml models (fuml). [Online]. Available: http://www.omg.org/spec/FUML/

[12] N. Ukić, P. L. Pályi, M. Zemljić, D. Asztalos, and I. Markota, "Evaluation of bridgepoint model-driven development tool in distributed environment," in *Workshop on Information and Communication Technologies conjoint with 19th International Conference on Software, Telecommunications and Computer Networks, SoftCOM 2011*, 2011.

[13] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, *Guide to the software engineering body of knowledge-SWEBOK*. IEEE Press, 2001.

[14] A. J. Albrecht, "Measuring application development productivity," in *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*, 1979, pp. 83–92.

[15] L. Lavazza and G. Robiolo, "Introducing the evaluation of complexity in functional size measurement: a uml-based approach," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010.

[16] O. M. G. (OMG). (2014) Automated function points. [Online]. Available: http://www.omg.org/spec/AFP/1.0/

[17] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. IEEE, 2002, pp. 271–278.

[18] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 215–223.

[19] S. Sarkar, G. Rama, N. Siddaramappa, A. Kak, and S. Ramachandran, "Measuring quality of software modularization," Mar. 27 2012, uS Patent 8,146,058. [Online]. Available: https://www.google.com/patents/US8146058

[20] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 367–377.

[21] R. Brooks, "Towards a theory of the cognitive processes in computer programming," *International Journal of Man-Machine Studies*, vol. 9, no. 6, pp. 737–751, 1977.

[22] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.

[23] B. Shneiderman and R. Mayer, "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *International Journal of Computer & Information Sciences*, vol. 8, no. 3, pp. 219–238, 1979.

[24] N. Wilde and M. C. Scully, "Software reconnaissance: mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.

[25] K. Chen and V. Rajlich, "Case study of feature location using dependence graph." in *IWPC*. Citeseer, 2000, p. 241.

[26] M. Petrenko, V. Rajlich, and R. Vanciu, "Partial domain comprehension in software evolution and maintenance," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 13–22.

[27] T. Hofmann, "Probabilistic latent semantic indexing," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 1999, pp. 50–57.

[28] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, p. 391, 1990.

[29] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. C. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *Software Engineering, IEEE Transactions on*, vol. 33, no. 6, pp. 420–432, 2007.

[30] K. D. Welker, P. W. Oman, and G. G. Atkinson, "Development and application of an automated source code maintainability index," *Journal of Software Maintenance: Research and Practice*, vol. 9, no. 3, pp. 127–159, 1997.

[31] Y. Zhou and B. Xu, "Predicting the maintainability of open source software using design metrics," *Wuhan University Journal of Natural Sciences*, vol. 13, no. 1, pp. 14–20, 2008.

[32] M. Nazir, R. A. Khan, and K. Mustafa, "A metrics based model for understandability quantification," *arXiv preprint arXiv:1004.4463*, 2010.

[33] C. Van Koten and A. Gray, "An application of bayesian network for predicting object-oriented software maintainability," *Information and Software Technology*, vol. 48, no. 1, pp. 59–67, 2006.

[34] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of systems and software*, 1993.

[35] K. Shibata, K. Rinsaka, T. Dohi, and H. Okamura, "Quantifying software maintainability based on a fault-detection/correction model," in *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*. IEEE, 2007, pp. 35–42.

[36] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," in *Reliability and maintainability symposium, 2002. Proceedings. Annual*. IEEE, 2002, pp. 235–241.

[37] R. Gunning, "The fog index after twenty years," *Journal of Business Communication*, vol. 6, no. 2, pp. 3–13, 1969.

[38] B. H. Sellers, "Modularization and mccabe cyclomatic complexity," 1992.

[39] M. Shepperd, "A critique of cyclomatic complexity as a software metric," 1988.

[40] G. J. Myers, "An extension to the cyclomatic measure of program complexity," *ACM Sigplan Notices*, 1977.

[41] S. S. M. S. S. A. I., "Cyclomatic complexity: The nesting problem," in *Preceedings of 2013 Eighth International Conference on Digital Information Management (ICDIM)*, 2013.

[42] S. Abd-El-Hafiz, "Entropies as measures of software information," in *Software Maintenance, 2001. Proceedings. IEEE International Conference on*.

[43] W. Harrison, "An entropy-based measure of software complexity," 1992.

[44] K. Kim, Y. Shin, and C. Wu, "Complexity measures for object-oriented program based on the entropy," in *Software Engineering Conference, 1995. Proceedings., 1995 Asia Pacific*, 1995.

[45] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, 2001.

[46] S. Henry and D. Kafura, "Software structure metrics based on information flow," 1981.

[47] K. D. Cooper, T. J. Harvey, and K. Kennedy, "Iterative dataflow analysis, revisited," *Proceedings of the PLDI'02*, 2002.

[48] M. S. Hecht, *Flow analysis of computer programs*, 1977.

[49] E. I. Oviedo, "Control flow, data flow, and program complexity," in *Proceedings of IEEE COMPSAC*, 1980.

[50] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," 1985.

[51] J. A. Cruz-Lemus, A. Maes, M. Genero, G. Poels, and M. Piattini, "The impact of structural complexity on the understandability of uml statechart diagrams," *Information Sciences*, 2010.

[52] OneFact. (2016) Bridgepoint tool. [Online]. Available: https://xtuml.org/download/

[53] G. Genova. (2009) What is a metamodel: the omg's metamodeling infrastructure. [Online]. Available: http://www.ie.inf.uc3m.es/ggenova/Warsaw/Part3.pdf

[54] BridgePoint, "Bridgepoint uml suite help - rsl reference," 2016.

[55] N. Ukic. (2016) Nenad ukic phd online resources. [Online]. Available: https://bitbucket.org/nukic/phd

[56] M. Lippert and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*, 2006.

[57] J. Cohen, "Statistical power analysis for the behavioral sciences (revised ed.)," 1977.

[58] T. D. Cook and D. T. Campbell, "The design and conduct of quasi-experiments and true experiments in field settings," *Handbook of industrial and organizational psychology*, vol. 223, p. 336, 1976.

[59] D. T. Campbell, "Factors relevant to the validity of experiments in social settings." *Psychological bulletin*, vol. 54, no. 4, p. 297, 1957.

[60] J. R. Fraenkel, N. E. Wallen, and H. H. Hyun, *How to design and evaluate research in education*. McGraw-Hill New York, 1993, vol. 7.

[61] S. Danziger, J. Levav, and L. Avnaim-Pesso, "Extraneous factors in judicial decisions," *Proceedings of the National Academy of Sciences*, vol. 108, no. 17, pp. 6889–6892, 2011.

[62] R. A. Berk, "An introduction to sample selection bias in sociological data," *American Sociological Review*, pp. 386–398, 1983.

# Curriculum Vitae

**Nenad Ukić**

Nenad Ukić was born on June 7, 1981 in Čakovec, Croatia. After finishing elementary school in Trogir, he attended Marko Marulić secondary school in Split where he graduated in 1999. During the same year he enrolled in study program at University of Split, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture (FESB). He graduated in October 2005 and defended his diploma thesis *Quality of Service management in Internet telephony* under supervision of prof. dr. sc. Nikola Rožić.

After the graduation he started working at research department of Ericsson Nikola Tesla company in Split. In the next few years, he was involved in development of several software prototypes including mobile ad-hoc (MANET) networks, sensors and machine-to-machine communication.

In 2010, as part of Model Driven Workflow (MDW) research and development project, he started working with xtUML executable software models. He was involved in development of several smaller prototypes using xtUML technology, including the Diameter protocol stack. As project continued, Nenad was involved in development, testing and maintenance of C++ and Java code generator which was later used for production of an EATF (Emergency Access Transfer Function) node in an IMS network. Recently, as the focus of the project moved towards standardization, Nenad is involved in migration of xtUML models to fUML-based executable models.

In 2008, he enrolled in PhD program at Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture. His main research interests are executable software modelling, software quality and understandability.

# Životopis

**Nenad Ukić**

Nenad Ukić rođen je 7. lipnja 1981. u Čakovcu, Hrvatska. Nakon završetka osnovne škole u Trogiru, pohađao je matematički smjer gimnazije Marko Marulić u Splitu, gdje je i maturirao 1999. Iste godine upisao je studij elektortehnike pri Fakultetu elektrotehnike, strojarstva i brodogradnje (FESB) na Sveučilištu u Splitu. Diplomirao je u listopadu 2005. godine i obranio diplomski rad pod naslovom *Upravljanje kvalitetom usluge u Internet telefoniji* pod mentorstvom prof. dr. sc. Nikola Rožića.

Nakon diplome počeo je raditi u istraživačkom odjelu tvrtke Ericsson Nikola Tesla u Splitu. U sljedećih nekoliko godina bio je uključen u razvoj nekoliko softverskih prototipova, uključujući mobilne senzorske ad-hoc (Manet) mreže, te automatsku strojnu (M2M) razmjenu podataka.

U 2010. godini, u sklopu *Model Driven Workflow* (MDW) razvojno-istraživačkog projekta, Nenad je počeo raditi s izvršnim xtUML softverskim modelima. Bio je uključen u razvoj nekoliko manjih prototipa koristeći xtUML tehnologiju, uključujući stog Diameter protokola. U nastavku projekta, Nenad je bio uključen u razvoj, testiranje i održavanje C ++ i Java generatora koda koji je poslije korišten u razvoju EATF (*eng.* Emergency Access Transfer Function) čvora u telekom mreži. Kako je fokus projekta stavljen na standardizaciji izvršnog softverskog modeliranja, u posljednje vrijeme Nenad radi na migraciji xtUML modela na izvršne modele bazirane na fUML-u.

Godine 2008. Nenad je upisao doktorski studij na Fakultetu elektrotehnike, strojarstva Strojarstva i brodogradnje te je objavio nekoliko znanstvenih radova. Njegovi glavni istraživački interesi su izvršno softversko modeliranje te kvaliteta i razumljivost softvera.

U N I V E R S I T Y O F S P L I T
FACULTY OF ELECTRICAL ENGINEERING, MECHANICAL ENGINEERING
AND NAVAL ARCHITECTURE

S V E U Č I L I Š T E U S P L I T U
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

**Nenad Ukić**

# XTUML MODEL COMPLEXITY METRICS AND THE INFLUENCE OF THEIR DISTRIBUTION ON MODEL UNDERSTANDABILITY

# MJERE SLOŽENOSTI XTUML MODELA I UTJECAJ NJIHOVE DISTRIBUCIJE NA RAZUMLJIVOST

DOCTORAL THESIS
DOKTORSKA DISERTACIJA

Split, 2016.

View publication stats