ROLE OF OCL AND ITS SUPPORTING TOOLS IN REQUIREMENT SPECIFICATION

A thesis submitted in partial fulfillment of the requirements

for the award of degree of

Master of Engineering

in

Software Engineering

Submitted By

SUNIL BABU DHARMANA

(Roll No. 800831013)

Under the supervision of

Ms. SHIVANI GOEL

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

THAPAR UNIVERSITY

PATIALA – 147004

June 2010

Certificate

I hereby certify that the work which is being presented in thesis entitled **"Role of OCL and its Supporting Tools in Requirement Specification"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering Submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under supervision of Ms. Shivani Goel and refers other researcher's works which are duly listed in reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

Sunil Babu D.

(Sunil Babu Dharmana)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

S Shivani Goel.

(Ms. Shivani Goel)
Assistant Professor
Computer Science and Engineering Department
Thapar University
Patiala.

Countersigned By

12/07/10

(Dr. Rajesh Bhatia)
Head
Computer Science and Engineering Dept,
Thapar University,
Patiala.

20/7

(Dr. R.K. Sharma)
Dean (Academic Affairs)
Thapar University
Patiala.

Acknowledgment

No volume of words is enough to express my gratitude towards my guide, Ms. Shivani Goel, Assistant Professor in Computer Science and Engineering Department, Thapar University, who has been very concerned and has aided for all the material essential for the preparation of this thesis report. She has helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research oriented venture.

I am also thankful to Dr. Rajesh Bhatia, Head of Department, CSED, and Dr. Inderveer Channa, P.G. Coordinator, for the motivation and inspiration that triggered me for the thesis work.

I would also like to thank the staff members and my colleagues who were always there in the need of the hour and provided with all the help and facilities, which I required, for the completion of my thesis. I am also grateful to my friends and fellow students, who gave me their support in and outside the laboratory.

Most importantly, I would like to thank my Parents and the Almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

Sunil Babu D

(Sunil Babu Dharmana)

800831013

Abstract

During the formal specification phase, the engineer rigorously defines a system using a modeling language. Modeling languages are fixed grammars which allow users to model complex structures out of predefined types. This process of formal specification is similar to the process of converting a word problem into algebraic notation. The Object Constraint Language (OCL) is a notational language for analysis and design of software systems, which is used in conjunction with the Unified Modeling Language (UML) to specify the semantics of the building blocks precisely. If the focus of the development activities is shifted from implementation code to more abstract models then software developers need a formalism to provide a complete, unambiguous and consistent model at a very detailed level. So, OCL is currently the only language that can bring this level of detail through specification to UML models. OCL can also be used by other languages, notations, methods and software tools in order to specify restrictions and other expressions of their models. Likewise, OCL is used by the Object Management Group (OMG) in the definition of other fast spreading industrial standards such as Meta Object Facility (MOF) or XML Metadata Interchange (XMI). The language is very powerful because it can be used together with class and other UML diagrams at different model layers by the specification of OCL constraints at the meta model layer.

In this thesis, work is an attempt to synchronize the language specification and its understanding, straight related to the language improvement in CASE tools, by proposing solutions for incomplete or non deterministic OCL specifications. Also present here is a tool based approach to validating UML models and OCL constraints by implementing a Case Study. The ideas presented in a Case Study have been implemented in the OCLE2.0.4, describe existing principles and stages for generating code from OCL expressions pointing out the drawbacks that cause inefficiencies of the OCL Specifications and its resulting code generated by OCLE2.0.4. The proposed improvement of the code transformation is based on extended Abstract Syntax Trees (AST) with context specific attributes. Finally the feasibility of results presented in this

work was shown with the realization of the USE2.4.0 tool. The USE tool has been used to validate the well formedness rules in the UML standard and OCL specifications of a case study. The results provide input for improving future versions.

Key Words: UML, OCL, USE2.4.0, AST, OCLE2.0.4, CASE, MOF, OMG, XMI, XML.

Table of Contents

List of Figures

List of Tables

Abbreviations

UML	-	Unified Modeling Language
OMG	-	Object Management Group
MOF	-	Meta Object Facility
XMI	-	XML Meta Data Interchange
OCL	-	Object Constraint Language
OMT	-	Object Modeling Technology
IBM	-	International Business Machines Corporation
OCLE	-	Object Constraint Language Environment
CASE	-	Computer Aided Software Engineering
DTD	-	Data Type Definition
USE	-	UML based Specification Environment
MDA	-	Model Driven Architecture
PIM	-	Platform Independent Model
PSM	-	Platform Specific Model
AST	-	Abstract Syntax Tree
SDLC	-	Software Development Life Cycle
ICO	-	Interactive Cooperative Objects
CST	-	Concrete Syntax Tree
MDE	-	Model Driven Engineering
AATC	-	Advanced Automatic Train Control
ASSL	-	A Snapshot Sequence Language

CHAPTER 1
INTRODUCTION

1.1 Overview

Analysis and design are important tasks in the process of constructing software systems. A number of different methods and languages support systematic approaches to these tasks. The Unified Modeling Language (UML) has recently gained much attention in this area. The language was accepted as a standard [22] by the Object Management Group (OMG) in 1997. The most recent version adopted as an official standard by the OMG is UML 2.0. The UML provides nine diagram types such as use case diagrams, class diagrams, state diagrams and sequence diagrams for modeling different aspects of a system. A further important part of UML is the Object Constraint Language (OCL) [7] which is a textual language that allows to specify additional constraints on models in a declarative way similar to predicate logic. While the UML offers an appealing and expressive notation, still what the precise meaning of a model is, is an unsolved problem [32]. A number of problems related to under specified constructs, ambiguities and contradictions have already been identified in the past[13]. For overcoming this incompleteness in Specifications OCL tools were emerged[8], Although the UML definition is much more rigorous than most of its predecessors like Object Modeling Technology (OMT)[4], the syntax and semantics of UML currently do not have a formal foundation. In our view, it is important to have a precise semantics of UML models and OCL constraints. Precise foundations are needed for analysis, validation, verification, and transformation (such as refinement and code generation) of models. They are also pre-requisite or providing tools with a well defined and predictable behavior.

1.2 The Role of Formal Methods

Formal methods are intended to systematize and introduce rigor into all the phases of software development. This helps us to avoid overlooking critical issues, provides a standard means to record various assumptions and decisions, and forms a basis for consistency among many related activities. By providing precise and unambiguous

description mechanisms, formal methods facilitate the understanding required to coalesce the various phases of software development into a successful endeavor.

The programming language used for software development furnishes precise syntax and semantics for the implementation phase, and this has been true since people began writing programs. But precision in all but this one phase of software development must derive from other sources. The term formal methods pertains to a broad collection of formalisms and abstractions intended to support a comparable level of precision for other phases of software development. While this includes issues currently under active development, several methodologies have reached a level of maturity that can be of benefit to practitioners.

1.2.1 Why Use Formal Methods?

- Improve quality of software system
- Maintainability
- Ease of construction
- Higher confidence in software product
- Detect design flaws
- Determine correctness
- Reduces burden on testing phases to detect all critical errors
- Tool support

1.2.2 Formal Specification Language OCL Evaluation

OCL was developed by Jos Warmer as a business modeling language within IBM and derived from Steve Cook's and John Daniels's Syntropy. OCL was IBM's first contribution to UML 1.1. The recent success has also led to increased popularity of OCL, especially from the 1.1 version of UML up onwards, when this language is used in the UML standard. The OMG is currently working on the new version OCL2.0 [16], where an important new aspect is a well defined meta model, which makes it easier to develop OCL tools. OCL2.0 is currently used in the definition of other OMG standards. In the specification of MOF (Meta Object Facility)[30], for example, OCL2.0 is used to define the more complex concepts of the MOF model. It is likewise used in the specification of XMI (XML Meta data Interchange) to specify the manner in [33,21] which a model is transformed into a document.

1.2.3 Why OCL?

A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints [18], so called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use. OCL has been developed to fill this gap. It is a formal language that remains easy to read and write. It has been developed as a business modeling language within the IBM Insurance division. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model. The evaluation of an OCL expression is instantaneous. This means that the states of objects in a model cannot change during evaluation.

1.2.4 Where to Use OCL?

OCL can be used for a number of different purposes:

- As a query language
- To specify invariants on classes and types in the class model
- To specify type invariant for Stereotypes
- To describe pre and post conditions on Operations and Methods
- To describe Guards
- To specify target (sets) for messages and actions
- To specify constraints on operations
- To specify derivation rules for attributes for any expression over a UML model.

1.2.5 Main Characteristics of OCL

- It is a pure expression language. It guarantees that an OCL expression will have no side effects [11]. In other words, the state of the system will never change as a result of an OCL expression.
- OCL2.0 assists in formalizing the language and provides a precise, easy way of expressing constraints in the structure of the models [30].

- OCL2.0 is not a programming language; therefore, it is not possible to write program logic or flow control in OCL2.0.
- OCL2.0 is a typed language and hence every expression has a type. For an expression to be correct, all the types must be in agreement.
- Each OCL expression is attached to a model [22] and describes specific characteristics of the model.
- Describe pre conditions and post conditions on Operations and Methods.
- Its act as a navigation language.
- OCL enables a better documentation. Visual models define some constraints, like association multiplicity, but in OCL we can specify richer ones. Diagrams are incapable of conveying important elements such as uniqueness constraints, formulae, limits and business rules [11].

An OCL expression is declarative in the sense that an expression says what constraint has to be maintained, not how this is accomplished. Therefore, specification of constraints is done on a conceptual level, where implementation aspects are irrelevant.

1.2.6 OCL Special Concepts

Navigation: An association between two classes provides a path that can be used for navigation [7, 17]. A navigation expression may start with an object and then navigate to a connected object by referencing the latter by the role name attached to its association end. The result is a single object or a collection of objects depending on the specified multiplicity for the association end. Navigation is not limited to a single association. Expressions can be chained to navigate along any number of associations.

Iterate: OCL provides a general iteration construct by means of an iterate expression. The source of an iterate expression always is a collection value. For each element of the collection, an expression is evaluated whose result may be accumulated in a result variable. Many other collection operations can be defined in terms of an iterate expression.

Undefined values: An expression may result in an undefined value. In general, an expression evaluates to an undefined value if one of its sub expressions is undefined. Exceptions to this rule are given for logical operations.

Flattening: OCL tries to avoid complex collections by an "automatic flattening" mechanism. Whenever a nested collection occurs, it is reduced to a simple collection. Nested collections often result from navigation.

Special Types: The types OclType, OclAny, OclExpression, and OclState are special in [28] OCL. OclType introduces a meta level by describing as instances the set of all types. OclAny is the top of the type hierarchy for all types except for the collection types. OclExpression seems to be required only for defining expressions which combine other expressions. OclState is used for referring to state names in a state machine.

Shorthand Notations: There are several features on the syntax level for writing constraints in a more convenient and abbreviated way. A let-construct can be used to bind a complex expression to a variable which may then repeatedly occur in a constraint. Further shorthand notations exist for collect expressions and certain kinds of navigation expressions.

1.3 OCLE2.0.4 brief description

OCLE2.0.4 is a UML CASE Tool [17] offering full OCL support both at the UML meta model and model level. The first objective was to implement the support needed for checking the well formedness of UML models. OCLE2.0.4 offers a very strong support for compiling and debugging OCL specifications [36]. We can use UML models saved in XMI 1.0 or 1.1, regardless of the tools and parsers used in producing and transferring the models.

1.3.1 Why OCLE2.0.4?

- OCLE helps users in realizing both static and dynamic checking at the user model level. Dynamic support is offered by means of the generated Java source code. In this context it is worth mentioning the quality of the code generated for the model architecture and for OCL specifications.

- Semantic checking of XML (Extensible Mark-up Language) documents described using DTDs is one of the straightforward applications of UML model checking.
- OCLE enables the reverse engineering of DTD (Data Type Definition)[30,36] files.
- The graphical interface was conceived and implemented with the aim of supporting the use of OCLE in a natural and intuitive manner.

1.4 Overview of USE Features

The USE (UML based Specification Environment) application has been developed in [6] Java by Mark Richters at the University of Bremen. The formal foundation of this tool, as well as a survey on some OCL tools is presented in [7]. Model validation through exploring properties of models is a significant task within model based software development. The USE system supports developers in analyzing the model structure (classes, associations, attributes, and invariants) and the model behavior (operations and pre and post conditions) by generating typical snapshots (system states) and by executing typical operation sequences (scenarios) [19]. Developers can formally check constraints (invariants and pre and post conditions) against their expectations and can, to a certain extent, derive formal model properties.

A USE specification contains a textual description of a model using features found in UML class diagrams (classes, associations). Expressions written in the Object Constraint Language (OCL) are used to specify additional integrity constraints on the model. A model can be animated to validate the specification against non-formal requirements. System states (snapshots of a running system) can be created and manipulated during an animation. For each snapshot the OCL constraints are automatically checked. Information about a system state is given by graphical views. OCL expressions can be entered and evaluated to query detailed information about a system state.

1.5 Research Goals and Contributions

The goal of this work is to define a precise syntax and semantics for the Object Constraint Language in its Supporting tools and essential parts of the UML core that provide the context for OCL constraints in a problem requirement specification. The practical benefits of this formalization are demonstrated by a tool based approach for

code generation and validating OCL constraints in UML meta models. The claim here is that a formalization of OCL and a well defined subset of UML provides a proper foundation for precise modeling of software systems. This work makes the following significant contributions:

- A precise definition of OCL avoiding ambiguities, under specifications, and contradictions is given by comparing results of constraints in OCLE2.0.4 and USE2.4.0 [3].
- Several lightweight extensions are proposed to improve the orthogonality of the language.
- The integration of UML and OCL is improved by making the relationships and dependencies between both languages explicitly using OCLE 2.0.4.
- A solid foundation for tools supporting analysis, simulation, transformation, code generation and validation of UML models with OCL constraints is developed by using case study. Results of this work is verified and validated in a tool USE 2.4.0.

1.6 Thesis Structure

The thesis is organized as follows:

Chapter 2: Gives guidelines for formal specifications and its use in different stages of Software Development Life Cycle (SDLC), background information on OCL Meta Models and its Meta Object Facilities. It covers aspects mainly related to the generic three layer Meta Data Architecture, its supporting syntactic, semantic meanings of expressions containing undefined values, code generation process and its currently supported tools.

Chapter 3: Problem description and approach which is followed to find the solution by using above mentioned formal language OCL and its supporting tools (OCLE2.0.4, USE2.4.0).

Chapter 4: Provide solution for above mentioned problem in a way to write specification first in language like English then implementing it in OCL2.0 and generate Java source code by OCLE2.0.4.

Chapter 5: Reports on a tool (OCLE2.0.4, USE2.4.0) based approach to validating UML models and OCL constraints of above solution.

Chapter 6: Concludes the thesis with a short summary and an outline of the contributions and future work.

CHAPTER 2

LITERATURE REVIEW

2.1 Guidelines for developing formal specifications

Following are guidelines given by [2]

- Clearly define the role of formal specifications in your software development process.
- Define the steps for developing formal specifications.
- Carefully divide specifications among broad classes such as functionality, operations, behavior, and interface.
- Select appropriate parts of your application for formal specification.
- Choose appropriate formal specification notations and tools for each class of requirements.
- Maintain high level abstraction by avoiding design decisions and implementation details.
- Avoid over and under specification as well as nondeterministic and partial specification.
- Build modular specifications containing good and natural structure.
- Always try to find many alternate representations and then choose the one best suited for the problem.
- Build specifications for reusability using libraries of useful metaphors and patterns.
- Be true to the spirit of the notation.
- Review and test the specifications thoroughly, document the test cases.
- Document the specifications well and provide explanations.
- Effectively use available tools.
- State and prove (or argue about or demonstrate) all the specification's necessary properties.

2.1.1 Using formal specifications in different stages of the SDLC

Formal Specifications: Advocate the use of mathematical notations and methods throughout the SDLC and encompass formal specifications in [2], formal program derivation through refinements, and program verification. Rigorous mathematics,

aided by prototyping and proofs, leads to early requirements problem detection. Tools can sometimes derive more practical benefits from formal specifications for code generation, refinement and test generation.

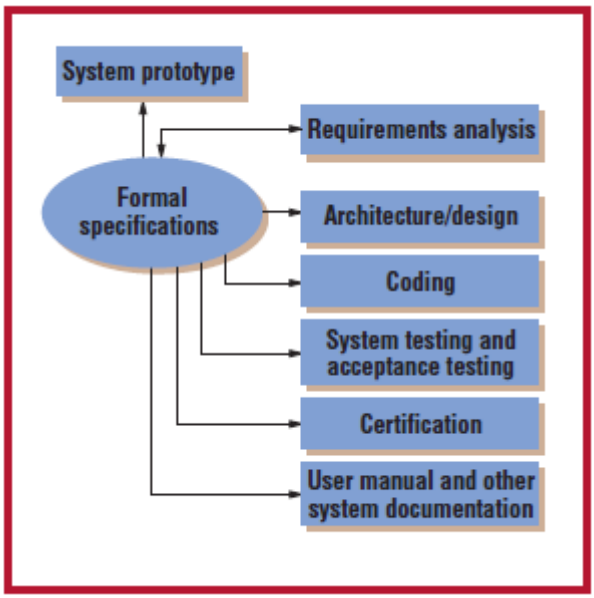


Figure2.1: Formal Specification in SDLC [2]

2.1.2 The ICO Formal Description Technique

The Interactive Cooperative Objects (ICOs) formalism is a formal description technique dedicated to the specification of interactive systems [37]. It uses concepts borrowed from the object oriented approach (dynamic instantiation, classification, encapsulation, inheritance, client/server relationship) to describe the structural or static aspects of systems, and uses high level Petri nets [1] to describe their dynamic aspects at the end of this prototyping phase. Indeed, development of safety critical applications require a more structured and global development process as the one promoted by the waterfall model. The right hand side of Figure2.2 shows the basics of the waterfall development process.

This Figure2.2 represents also how the prototyping iterative process and the waterfall one are related. The approach we promote provides several valuable inputs for this classical development process:

- A set of validated requirements are elicited and tested by the users during the prototyping phase and thus reducing time spent in the requirement phase.

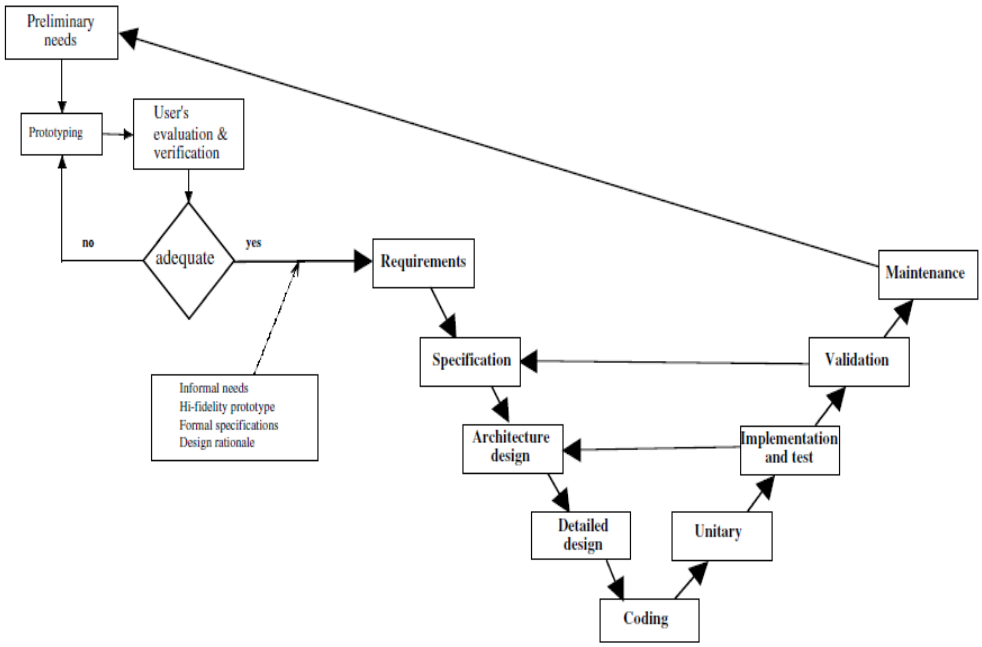


Figure 2.2: Integrating formal prototyping with software development[37].

- A set of formal specification of the interactive part of the application. These specifications will be used as inputs in the specification phase and will thus contribute to reduce development time.
- A complete user interface (both its presentation and its behavior) that will have to be re-implemented in the development phases. Indeed, the user interface produced by the execution of the ICO specification cannot be used for the final system towards interpretation and thus cannot reach the level of performance required for safety critical applications.

2.2 The Generic three layer Meta Data Architecture

Each modeling language is defined in another language, its meta modeling language. For example, the Unified Modeling Language is defined using the Meta Object Facility (MOF) in [30], the standardized meta-meta language of the OMG. The MOF is used to describe the UML meta model that can be used to model UML models. Generally speaking, each model requires a meta model that is used to describe the model. The model can be instantiated by model instances (for example a UML class diagram could be instantiated by a UML object diagram). One may also say, that the model is an instance of its meta-model.

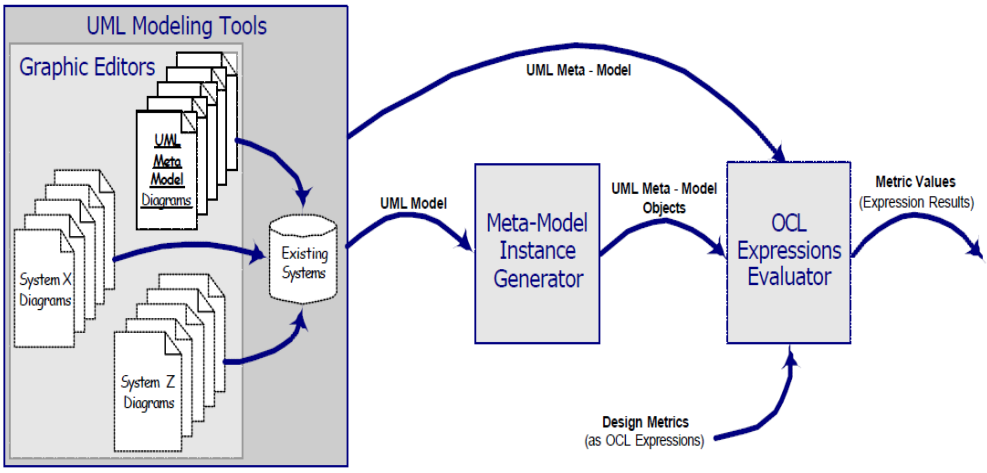


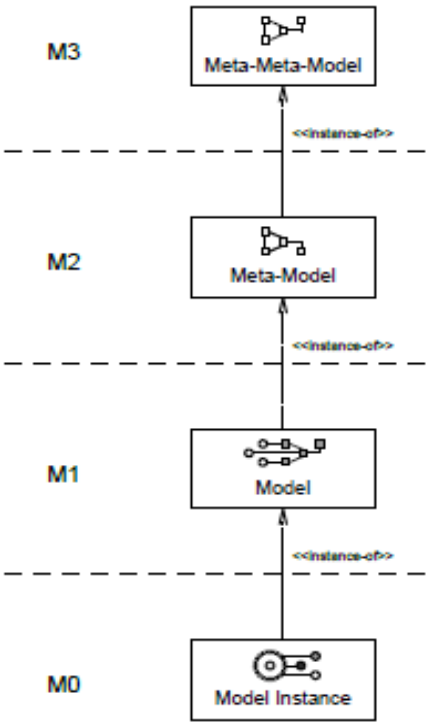
Figure 2.3: Meta Model Level Architecture [1]

This model to meta model instance of relationship can be considered as relative, because every model is an instance of a meta model. The meta model is again an instance of a meta-meta model (the meta model's meta model). Each model can be enriched with OCL constraints that are defined on the model and can then be verified for instances of the model.

2.2.1 Generalization of Three Layer Metadata Architecture

The OMG introduced the MOF Four Layer Metadata Architecture[9], which is used to arrange and structure the meta-model, the model, and its model instances into a layered hierarchy (Generally, four layers exist, the Meta- Meta Model Layer (M3), the Meta-Model Layer (M2), the Model Layer (M1), and the Model Instance Layer (M0). OCL constraints can be defined on both meta models and models to verify models (Model Level Constraints) at instances level (Instance Level Constraints).

Thus, the four layer metadata architecture can be generalized to a Generic Three Layer Metadata Architecture in the scope of an OCL [9] definition (see Figure 2.4(b)). On the M_{n+1} Layer is the meta model that is used to define the model that shall be constrained. On the M_n Layer is the model that is an instance of the meta-model and can be enriched by the specification of OCL constraints. Finally, on the M_{n-1} Layer is the model instance on that the OCL constraints shall be verified.



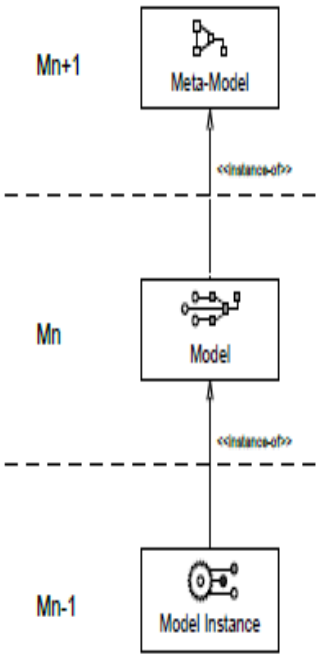


Figure 2.4(a): The MOF Four Layer Metadata Three Layer Architecture [9].

Figure 2.4(b): The Generic Architecture [9].

2.3 OCL Constraints and System States

OCL is a language allowing the specification of formal constraints in context of a UML model. Constraints are conditions on all states and transitions between states of a system implementing a given model. A set of constraints therefore restricts the set of possible system states in [5]. An invariant is a statement about all existing objects of a class. Only single system states need to be considered for determining whether an invariant is fulfilled or not.

Additionally, pre and post conditions enable behavioral specifications of operations in terms of conditions on a previous state and a post-state after executing the operation. The syntax style of OCL is similar to object oriented programming languages [28]. Most expressions can be read left to right where the left part usually represents in object oriented terminology the receiver of a message.

2.4 Ambiguous or not enough detailed aspects in the OCL specification

OCL is meant to complement model description in order to support a more complete and rigorous model specification. Working with undefined [13] values is really important when using models. At different stages of software development, it happens that not all decisions are taken or not all the information is known. However, even in these situations, for modelers, it is crucial to work with models in [15], and, by consequence, to evaluate OCL expressions that may contain undefined values. Of course, in such situations it is expected to obtain results as accurate as possible.

2.4.1 Evaluation of operations on collections that contain undefined values

The fact that the collection contains undefined values should not imply that all collection operations invoked on that collection result in undefined.

- The evaluation of the collection size should always return a positive integer value:

  Bag {1 , 9 , undefined , undefined }−>s i z e = 4

  Sequence {1 , 9 , undefined , undefined }−>s i z e = 4

- The evaluation of the exist() operation should iterate the entire collection and not terminate the iteration when an undefined value is encountered:

  Collection { undefined , 1 , 9 , undefined}−>e x i s t (e | e > 7) = true

  Collection {1 , 9 , undefined}−>e x i s t (e | e = 7) = undefined

- The evaluation of the any() operation on a collection that contains at least one element that satisfies the condition of the any() operation should result in one of these elements:

  Collection {1 , 9 , undefined , undefined } −>any (e | e>1) = 9

  Collection {1 , 9 , undefined } −>any (e | e>10) = undefined

- The evaluation of a forAll() operation on a collection that contains one element that does not satisfy the forAll() condition should result in false:

  Collection {1 , 9 , undefined } −>forAll (e | e>2) = false

  Collection {1 , 9 , undefined } −>forAll (e | e>=1) = undefined

The use of the undefined value is helpful in specifying operation behavior. Therefore, a new kind of Expression, UndefinedLiteralExpression, needs to be defined in the OCL grammar.

2.5 Code Generation Overview

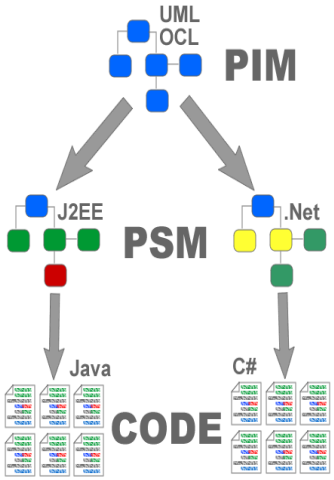


Figure 2.5: Process of Code Generation [17]

UML is a visual language having graphical means for modeling software. However, all modeling needs cannot be easily met by a graphical notation alone, thus UML models usually tend to be incomplete and inconsistent in [5]. Such models cannot be used for full fledged code generation and correctly updated when reverse engineering is performed because they are not precise enough.

The preciseness of models gets even more important under MDA (Model Driven Architecture). The purpose of MDA is to enable development of software on the higher abstraction level by using models and reusing them when migrating projects to different platforms and technologies. MDA proposes several abstraction layers, where Platform Independent Models (PIM) are transformed to Platform Specific Models (PSM) and finally to code. Traceability is one of the main requirements of successful software engineering and it should be possible to reverse code back to PSM and PIM models. Namely, OCL enables creation of precise models that can be transformed to rich code (not only skeletons of classes or interfaces) and reversed back without loss of information.

2.6 Analysis of OCL tools

Tools have been selected as being of great interest, on account of their incorporating new or distinctive characteristics. We have tried to take into consideration the whole range of possibilities offered by the tools and thus enable software developers to choose an OCL support tool [18, 23] which fulfils their previsions and meets their needs. Most of the tools analyzed are still in the development stage. The criteria that we have applied to evaluate the tools have been classified in five groups:

1. Syntactic analysis and type checking.

2. The next group corresponds to the communication capacity of each tool with UML models. The possibilities are:

- Model-independent tool, the tool can be used with no UML model loaded. The application can work without model information.
- Connection with UML model, i.e. the tool can have information about the model. Means of introducing the UML model: this may be through the standards XMI[30], XML, its own file or through "frontend" applications.

3. The third group is related to the facilities that the tools offer the user:

- Guided support given by the tool to construct constraints [27].
- Code generation from the OCL specifications.
- Definition of possible target programming language(s).
- Means of introducing OCL expressions. Here we find three options: first, the expressions are imported from a UML model; second, they are imported from an independent file; third, they can be manually introduced.

4. The fourth group, the capacity of dynamic validation that the tools bring to the UML model, includes:

- Dynamic validation of the invariant. Constraints are evaluated with respect to a UML model. The constraint of an invariant must be true for all possible instances of a class. The term "dynamic" here means that specific objects have been created either as a result of the execution of the model, or have been simulated manually by the analyst.

- Consistency checking. Most of the tools work with the so called constraint compatibility, i.e. checking that the names of the classes, attributes and operations used in [31] the constraints correspond to those of the UML model.
- Dynamic validation [19] of the preconditions and post conditions.

In this case the validation is made on pre- and post conditions of an operation. The operation is executed (or simulated manually) and evaluated. The pre and post conditions are checked.

2.7 OCLE2.0.4 Features

Compiler features

- Recursive declarations[36]
- Function redefinition
- Ambiguity detection in case of multiple inheritance
- Possibility to use let functions irrespective of the declaration file or of their position in the file where they are declared

Project structure

In order to offer an optimal management and easier understanding of large OCL specifications [17], the user can (but is not required to) define a project structure comprising

- The UML model
- One or more OCL specifications expressed at the meta model level
- One or more OCL specifications expressed at the user model level
- Different files including the UML diagrams attached to the project

UML Models and OCL Specifications can be managed even outside OCLE projects.

Debugging features

- Explicit error messages
- Detailed identification of erroneous sub-expression
- Possibility to compute the type of sub-expressions
- Possibility to acquire debugging information by means of metamodel and model browsers and the property sheet
- Support for modifying element properties by means of the property sheet

- Neutral printing functions built in the OCL type system

Evaluation features

- Dynamic and static linking
- Single evaluation for a (model Element, OCL constraint) pair explicitly specified by the user
- Batch evaluation of the whole model: all compiled constraints are evaluated on all the corresponding instances identified in the active user model
- Partial batch evaluation of the model: the constraints explicitly specified by the user are evaluated for the model elements chosen by the user
- Context persistence
- The errors and warnings identified in the batch evaluation process are reported in an intuitive manner
- The user has the possibility to navigate the model and to do a detailed evaluation of sub expressions of the invariants in order to find the causes

Browsers and Property sheet

- Project browser used to define and modify the project structure
- Property sheet enabling the user to see and modify the properties of model elements and to navigate the model
- Powerful meta model and model browsers
- Filtering of model elements based on their type
- Presentation of model elements at different levels of detail
- Advanced search functions: search options by name and type

Text editor

- Multi document architecture
- Implements all the usual standard facilities of a text editor: undo-redo, copy, paste, search & replace
- Syntax highlighting and auto-indent (for OCL specifications)
- Easy configuration control including persistence of user settings
- Can be used to edit other text sources apart from OCL text files (XML documents, java source code)

Code generation

Java source code generation for model structure and OCL specifications.

All generated code can be used in the application due to glue code (also includes the OCL type system implementation).

- Proper management of associations, even qualified. The generated accessor methods are designed to ensure the consistency of the runtime configuration, irrespective of the object that triggers the update operation
- Data types and Enumerations are also converted to functional source code
- Automatic verification of operation pre/post conditions

Static semantic validation of XML documents

- Semantic validation of XML documents described by means of DTDs [33].
- The type of class attributes can be changed from String to other types, supporting a finer checking.

2.8 USE2.4.0 Features

The USE tool (excluding the generator extension) is developed [8] as part of the PhD thesis of Mark Richters. The main task of USE is to validate and verify specifications consisting of UML class diagrams together with OCL invariants [23] and pre- and post conditions that are kept in a USE specification file (suffix .use). By validation, we mean that the developer can give test cases by means of object diagrams and manipulations of them and check whether the USE responses meet the intuition. By verification we mean that the test cases are formally checked with respect to invariants and pre and post conditions [12]. There are special USE commands for creating and manipulating object diagrams that can be accumulated in command files (with suffix .cmd).

2.8.1 Architecture of USE

A high level overview of the USE architecture is given in Figure 2.6. We distinguish between a Description Layer at the top, and an Interaction Layer below. The description layer is responsible for processing a model specification. The main component is a Parser for reading Specifications in USE syntax and generating an abstract syntax representation of a model. A USE specification defines the structural

building blocks of a model like classes and associations. Furthermore, OCL expressions may be used to define constraints and operations. The output of the parser is an abstract representation of a specification containing a Model and OCL expressions. The representation of the model is done with a subset of the Core package of the UML metamodel.

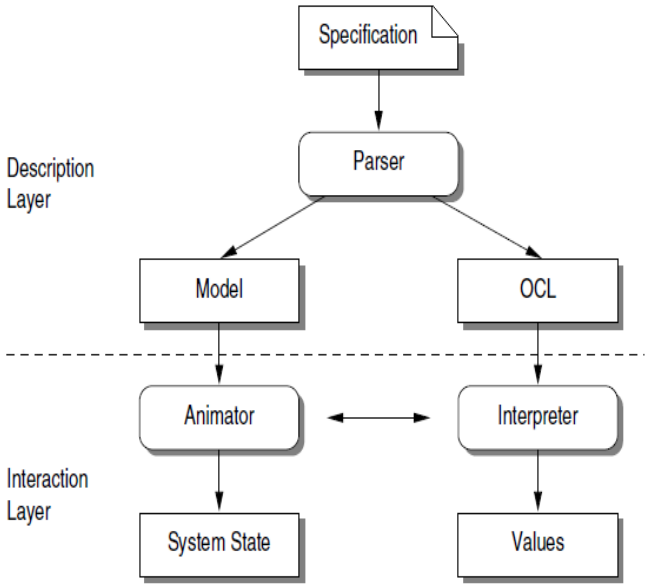


Figure 2.6: Overview of the USE architecture [25]

The chosen subset corresponds to the Basic Modeling Language presented and excludes all model elements which are not required during the analysis and early design phase of the software development process. The abstract representation of OCL expressions closely follows the OCL meta model.

The Interaction Layer provides access to the dynamic behavior and static properties of a model. The main task of the Animator component is the instantiation and manipulation of System States. A system state is a snapshot of the specified [19] system at a particular point in time. The system state contains a set of objects and a set of association links connecting objects [14,26]. As a system evolves, a sequence of system states is produced. Each system state must be well formed, that is, it must conform to the models structural description, and it must fulfill all OCL constraints. Furthermore, a transition from one system state to the next must conform to the

dynamic behavior specification given in form of pre and post conditions. The Interpreter component is responsible for evaluating OCL expressions. An expression may be part of a constraint restricting the set of possible system states. In order to validate a system state, the animator component delegates the task of evaluating all constraints to the interpreter. The interpreter is also used for querying a system state. A user may query a system state by issuing expressions that help inspecting the set of currently existing objects and their properties.

The Model and OCL branches in Figure 2.6 are tightly related to each other. For example, a model depends on OCL since operations of classes defined in a model may use OCL expressions in their bodies. A dependency in the other direction exists, because the context of OCL constraints is given by model elements. However, it is in general possible to define models which do not use OCL at all, and vice versa, there may be OCL expressions which just require an empty user model.

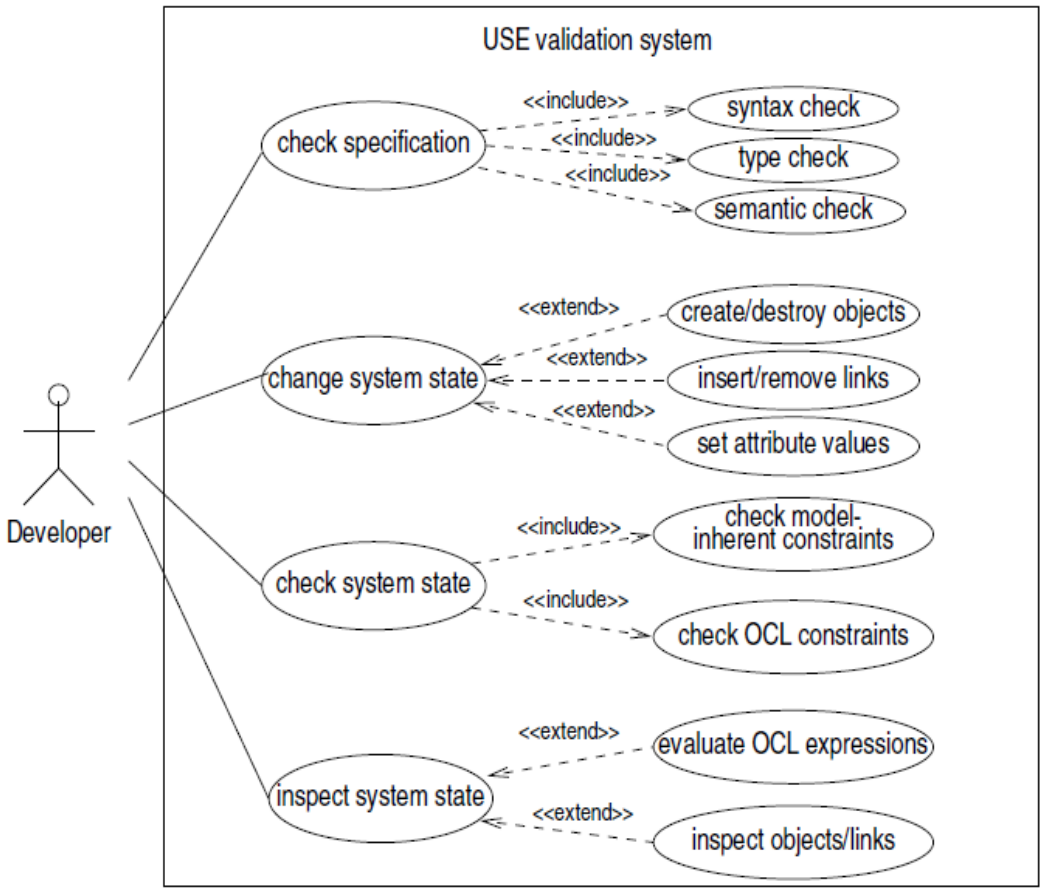


Figure 2.7: Use case diagram showing basic functionality of USE [6]

Task	USE2.4.0	OCLE2.0.4
Supporting Graphical and textual formalism	Yes	Yes
Supporting Abstraction levels	1 level	2 levels
Static evaluation for Invariants & pre/post conditions	Both	Invariants only
Code Generation	No	Yes
Browsing between different views	No	Yes
Facilities of text editor	Few facilities	Missing auto completion
Comply with standards	Proprietary model format	Mainly Yes
Exchange of models	No	UML in XMI
Snapshot Generation	Yes	No

Table 2.1: Evaluation results of OCLE2.0.4 and USE2.4.0

For example, the realization of various general purpose algorithms with OCL like sorting or determining the transitive closure of a relation is an interesting task on its own and can be done without the need for any particular model. Animator and interpreter closely work together. The animator asks the interpreter for evaluating OCL expressions. On the other hand, the Interpreter needs information about the current system state, for example, when evaluating an expression which refers to the attribute value of an object.

2.9 Shortcomings of OCL2.0

OCL2.0 has the following shortcomings with respect to these requirements. For each concern,

(i). The use of operations in constraints appears to be problematic in two respects.

a. First, this adds to the operational flavor of OCL2.0. An operation may go into an infinite loop or be undefined. In these cases, the expression containing the operation is said to be undefined. This adds unnecessary complexity to the language and raises concerns in [34] about its precision such as:

What does it mean for a model to satisfy an undefined constraint?

How do we know that a constraint is undefined?

b. Secondly, consider an operation applied to a collection of objects that are instances of some class. A subset of these objects may also be instances of a subclass of that class [13], and the subclass may redefine the operation. It is not clear what operation gets applied to each object in the collection. Further more, even if we assume that we apply the redefinition of the operation to objects which are instances of the subclass, then this implies that the meaning of a constraint may change as the model evolves and subclasses are added. This is quite undesirable, since the expression of the constraints of a system may have to be revised as the model evolves.

c. OCL's type system is unnecessarily complicated. Conceptually, a class is a set of objects, and a subclass is a subset of these objects. So if a class inherits from two classes, then the two classes must be non disjoint sets. In OCL2.0, it is possible to have a class which inherits from two seemingly disjoint classes.

(ii). OCL2.0 expressions are at times unnecessarily verbose due to the following:

a. OCL2.0 uses two different symbols to navigate through sets and scalars (-> and . respectively). This lack of uniformity adds unnecessary complexity and makes OCL expressions less succinct.

b. Classes are not treated simply as collections of objects. As a result, we cannot use set operators to manipulate them directly, which increases the use of quantifiers.

(iii). OCL2.0 expressions are often unnecessarily hard to read:

Sub expressions are often not textually placed where they normally would be in English, because quantifiers (for all), and collection operators are stacked", i.e. followed through via navigation. This also makes parenthesis matching difficult. Logical operators (like not) are not stacked, and this causes the separation of certain logical and set operators, making some expressions hard to read.

2.9.1 OCL Applications

Applications of OCL can be found in several standards published by the OMG. The following list gives some examples showing applications of OCL within meta modeling frameworks for defining languages.

• The probably largest published application of OCL is within the UML standard itself. OCL is used to specify well formedness rules for the UML abstract syntax. There are more than 150 invariants defined [24] on the meta model. Chapter 4 shows how these constraints can be automatically validated by a tool developed as part of this work to check UML models for conformance with the OMG standard.

• The Meta Object Facility (MOF) takes the meta modeling approach one step further and provides a meta-meta model for describing meta models in various domains [30]. For example, the UML can be considered an instance of the MOF. Because defining a meta-meta model is conceptually similar to defining a meta model, OCL is applied in a similar way within the MOF for specifying well formedness rules.

• The XML Metadata Interchange (XMI) provides a mechanism for interchange of metadata between UML based modeling tools and MOF based metadata repositories in distributed heterogeneous environments [33]. OCL is used in the XMI proposal to define the XMI stream production rules. The production rules specify how a model can be transformed into an XML document conforming to the XMI proposal.

CHAPTER 3

PROBLEM STATEMENT

As seen in previous chapters there are many inefficiencies in OCL specifications while using it with UML. UML models usually tend to be incomplete and inconsistence, such models cannot be used for full fledged code generation and correctly updated when reverse engineering is performed because they are not precise enough. So, Identifying and resolving design problems in the early design phase can help ensure software quality and save costs. There are currently few tools for analyzing designs expressed using the Unified Modeling Language (UML) with OCL constraints. Tools such as OCLE2.0.4 and USE2.4.0 support analysis of static structural properties and behavioral properties of UML models and OCL invariants. In this thesis, mechanisms are provided for checking instance models against invariant properties expressed using the Object Constraint Language (OCL) and Generic steps and principles are implemented for transformation of OCL code to the code of another language using Visitor patterns. The approach includes a technique for generating a class model of behavior from operation specifications expressed in a restricted form of OCL behavioral properties are expressed as invariants defined in the class model of behavior. Static analysis tools such as USE2.4.0 and OCLE2.0.4 can be used to check object models describing series of snapshots. Most of the analysis can be automated.

3.1 Proposed Approach

- By evaluating OCL expressions in OCLE2.0.4 and USE2.4.0 ,which includes:
  - Evaluating expressions containing undefined values and managing evaluation exceptions.
  - Evaluating undeterministic operations - any, asSequence, asOrderedSet.
  - Accessing features with the same name, from ascendants.
- Implementing a Case Study in OCLE2.0.4 and doing Verification and Validation of a Case Study in Validating tool USE2.4.0. In this way we can identify limitations of tools and its supporting levels.

3.1.1 Case Study

Objective is the specification of a simplified version of the Train System and of an algorithm that can control the speed and acceleration of trains in this system. A track of the system is partitioned into track segments. Segments may be bounded by gates. Station computers, which are part of the Advanced Automatic Train Control (AATC) system, control the trains in their immediate area by giving speed and acceleration commands to the trains. In each control zone, up to 20 trains can be handled. We abstract from communication links between computers and trains, the on board train control system, managing entry and exit of trains into the system, and hand offs between stations. We assume that the interlocking system does not close a gate when it is to late for an approaching train to stop, not even as a signal to following trains.

3.1.2 Specifying Constraints in General Language

- A train should not enter a closed gate.
- A train should never get so close to a train in front that if the train in front stopped suddenly the (following) train would hit it.
- A train should stay below the maximum speed that track segment can handle.
- The invariant fitting states that the end of a segment is equal to the begin of the next segment if there is a next segment.
- The length of a segment has to be the difference of segment end and begin.
- Connected segments belong to the same track.
- The origin and the destination of a train have to be connected by a sequence of segments.
- A train can not be commanded to travel faster than 80 mph or slower than 0 mph.
- A station computer is able to calculate the worst case stopping distance only of trains that are in the region the computer is responsible.
- The segments that bound the region a station computer is responsible for are connected.
- The constraint civilSpeedSafety demands that a “train should stay below the maximum speed that segment of track can handle”.

- A train should not enter a closed gate. The invariant with the name closed-GateSafety says that if a next closed gate exists, the distance to it is greater than the worst case stopping distance of the train, i.e., the train can stop in time.

- A train should never get so close to a train in front that if the train in front stopped suddenly the (following) train would hit it". The invariant crashSafety says (analogously to the preceding invariant) that if a train in front exists, the distance to it is greater than the worst case stopping distance of the (following) train.

  These constraints are specified in OCL in next chapter.

CHAPTER 4

DESIGN AND IMPLEMENTATION

4.1 Identifying Inefficiencies of OCL Specifications

4.1.1 Expressions Containing Undefined Values and Managing Exceptions

In the OCL specification, undefined [13] is used both for mentioning that some information is missing and in case of some runtime exceptions such as division by 0 or accessing the elements of an empty collection. As we will discuss in the following, this can cause unpleasant situations. Suppose that, by navigation, we obtain a collection of Integers, representing some person's ages. If we are interested in knowing whether there is at least a person aged more than 80, using the following OCL expression:

Bag{89,15,23,undefined}->exists(a|a>80), in accordance with the OCL specification, we will obtain undefined, even if such a person exists. This situation is unacceptable, therefore, proposing a more detailed evaluation strategy for expressions containing undefined values. In accordance with proposal obtain true in each case when the collection iterated by the exists() operation contains at least one element complying with the rule specified in the body of exists. Evaluating expressions containing undefined values is included in the bench mark test addressed in [15]. Analyzing the results obtained when evaluating the following three OCL expressions, we have noticed that the obtained results depend on the tool used.

We consider that the results obtained by using OCLE are more accurate than those obtained with the USE tool.

The differences in evaluation are due to the followings reasons:

• In OCLE, Sequence{1,8,7}->at(i) > Sequence{2,3}->at(i)}

Result: Raises the exception \sequence index out of range" when i=3, because Sequence{2,3} has only two elements.

• In USE evaluate Sequence{1,8,7}->at(i) > Sequence{2,3}->at(i)}

Result: false.

• The expression 7 > oclUndefined(Integer) is evaluated to undefined in OCLE and to false in USE.

The results obtained using OCLE2.0.4 are the same with the results obtained in case of a dynamic evaluation, at run time, when the expressions specified in Java would be translated from the above OCL expressions. In the context of the new Software Engineering paradigms (MDA, MDE), the full conformance of static and dynamic evaluations is essential. We suggest introducing a new class, Exception, in the OCL meta model, in order to model the exceptions that can be raised when evaluating OCL operations. In this manner, the results obtained will be more accurate and the framework required for obtaining the same results at both static and dynamic evaluation will be ensured.

In cases when is expected to obtain undefined values, we recommend taking a more secure approach, namely using the standard oclIsUndefined() operation. This because, in our opinion, it is not secure to say that the value of the expression oclUndefined(T) = oclUndefined(T), is true. In this case, we agree with the standard. The above mentioned value is undefined.

4.1.2 Evaluating Undeterministic Operations - any, asSequence, asOrderedSet

From a tool maker perspective, the OCL standard [16] gives an incomplete specification for the any(iterator | <boolean expression>) operation defined on Collections. Therefore, the results obtained when evaluating OCL expressions that include any depend on the tool.

The results will be the same, irrespective of the number of times the evaluation is performed. As both outputs comply with the standard, we consider that, in this case, the algorithm for evaluating the any operation must be included in the OCL specification. Moreover, modelers must use this operation carefully, taking into account the different results that can be obtained by evaluation.

In the standard, the collection operations asSequence and asOrderedSet are specified as undeterministic operations. Therefore, the results obtained when evaluating these operations depend on the used tool. It is enough clear that the evaluation strategies implemented in these tools are different.

- In OCLE, both asSequence and asOrderedSet operations do not change the order in which the set elements are listed.
- In USE, asSequence is equivalent to sortedBy(i | i). That is why in USE we obtain:

  Set{9,1,7,oclUndefined(Integer)}->sortedBy(i | i) =
  Sequence{oclUndefined(Integer),1,7,9}.

The results obtained with USE show that, in the 2.4.0 version of this tool, OrderedSet was not yet implemented, and that oclUndefined(Integer) < 1.

However, when we evaluated the expression oclUndefined(Integer) < 1 separately, the result obtained was false. The solution adopted by OCLE gives the opportunity to keep the order of terms when including new terms in the collection.

4.1.3 Accessing Features with the Same Name, from Ascendants

Accessing features defined in parents is an usual operation in object oriented applications. Suppose that we are in the C context and that we need to redefine the operation o1():Integer, initially defined in A. In order to do this, normally there are at least two possibilities:

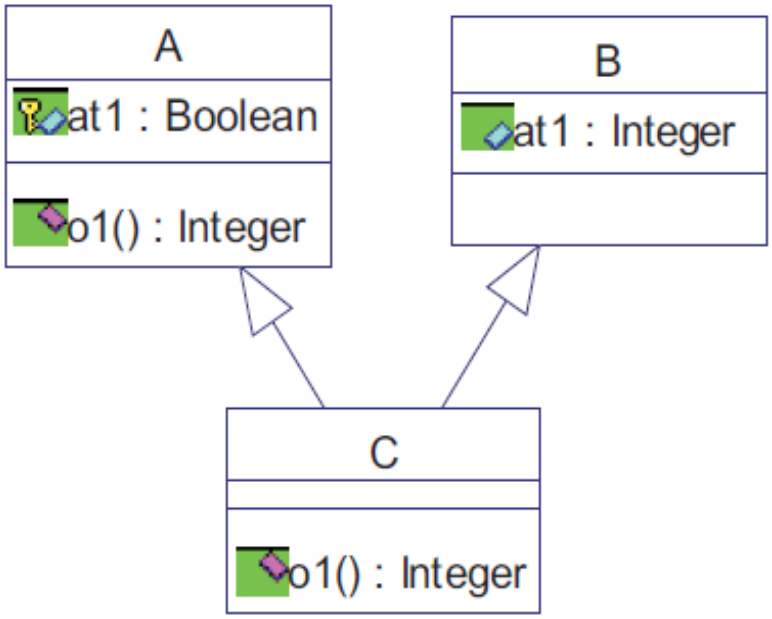


Figure 4.1: Accessing features with the same name, from ascendants

(1) using an upcast, as in:
context C::o1():Integer
body: self.oclAsType(A).o1() + 1 or

(2) using an explicit notation, like in C++:
context C::o1():Integer
body: self.A::o1() + 1

context C

inv: self.oclAsType(A).at1 implies self.oclAsType(B).at1 > 0
The OCL standard specification mentions only the downcast, but not the upcast. We consider that this is a mistake, at least for two reasons:
(1) upcasts (to an ascendent) are always safe and
(2) sometimes it is really necessary to be able to access features defined in ascendents, as we have discussed before.

4.2 Implementation of a Case Study

4.2.1 Inputs to the control algorithm

The control algorithm has access to all relevant attributes of all trains, including position, speed, acceleration, length, and currently commanded speed and acceleration. Information of the track is also available, i.e., location and grade of segments, maximum allowable speed, and location and state (open, closed) of gates.

4.2.2 Worst case stopping profile

Speed and acceleration of a train has to be selected so that
(1) The train does not hit a train in front of it or
(2) Enter a closed gate, not even in the case of very poor stopping conditions. Speed and acceleration must be consistent with worst-case safety bounds. We do not have any secondary objectives. Some secondary objectives of the case study (e.g. regarding the passengers comfort) possibly will be accomplished but we neither put any effort in it nor do we test such properties.

4.2.3 Design of a Train System

- Specifying existing train system in natural language like English in chapter 3.
- Modeled by a UML class diagram with additional OCL constraints in OCLE2.0.4.
- Implementing OCL Specification to all side effect free operations of the classes and safety requirements, that will calculate the results. Syntactic and Semantic checking of the structure by OCLE2.0.4.
- Due to lack of completeness and consistency of UML models, OCLE2.0.4 generate inefficient source code in Java according to the specification of a system. So, proposing and implementing concepts and methods for efficient source code.
- Finally validating a Case Study by using validating tool USE2.4.0.

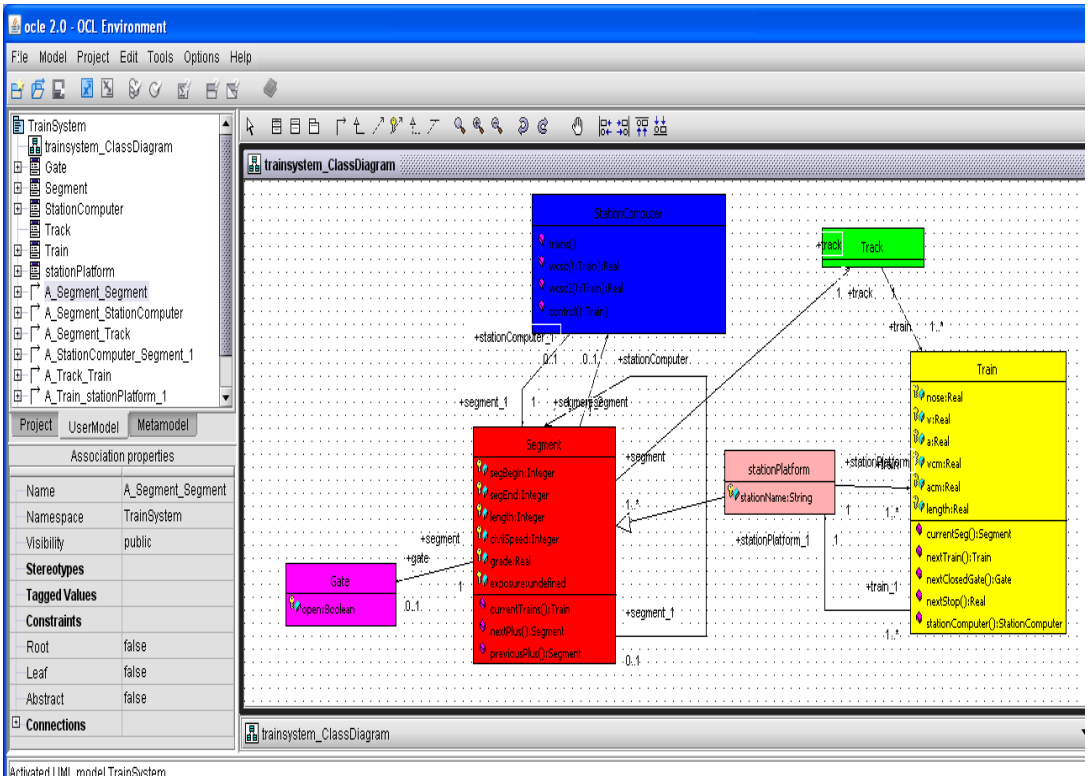


Figure 4.2: Class Diagram of Train System in OCLE2.0.4.

The ASSL procedure control(t: Train) implements the operation of class Station Computer that calculates the new commanded speed and acceleration for a train t. These values have to respect the constraints of the train system including civilSpeed Safety, closedGateSafety and crashSafety. The algorithm inspects the section of the track that begins with the nose of the train and is twice as long as the (more pessimistic but less fluctuating) worst case stopping distance wcsd2(t).

4.2.4 Implementing OCL Specification

- The invariant fitting states that the end of a segment is equal to the begin of the next segment if there is a next segment.

```
context Segment
    inv fitting:
    self.next.isDefined implies
    self.next.segBegin=self.segEnd
```

- The length of a segment has to be the difference of segment end and begin:

```
context Segment
    inv correctLength:
    self.segEnd-self.segBegin = self.length
```

- Connected segments belong to the same track:

```
context Segment
    inv track:
    self.next.isDefined implies self.track = self.next.track
```

- The origin and the destination of a train have to be connected by a sequence of segments:

```
context Train
    inv line:
    self.orig.nextPlus()->includes(self.dest)
```

- A train can not be commanded to travel faster than 80 mph or slower than 0 mph :

```
context Train
    inv vcm:
    self.vcm >= 0 and self.vcm <= 80
```

- The commanded acceleration is either between 0 and 3 or between -2 and -0.45 mps^2:

```
context Train
    inv acm:
```

(self.acm >= 0 and self.acm <= 3) or

(self.acm >= -2 and self.acm <= -0.45)

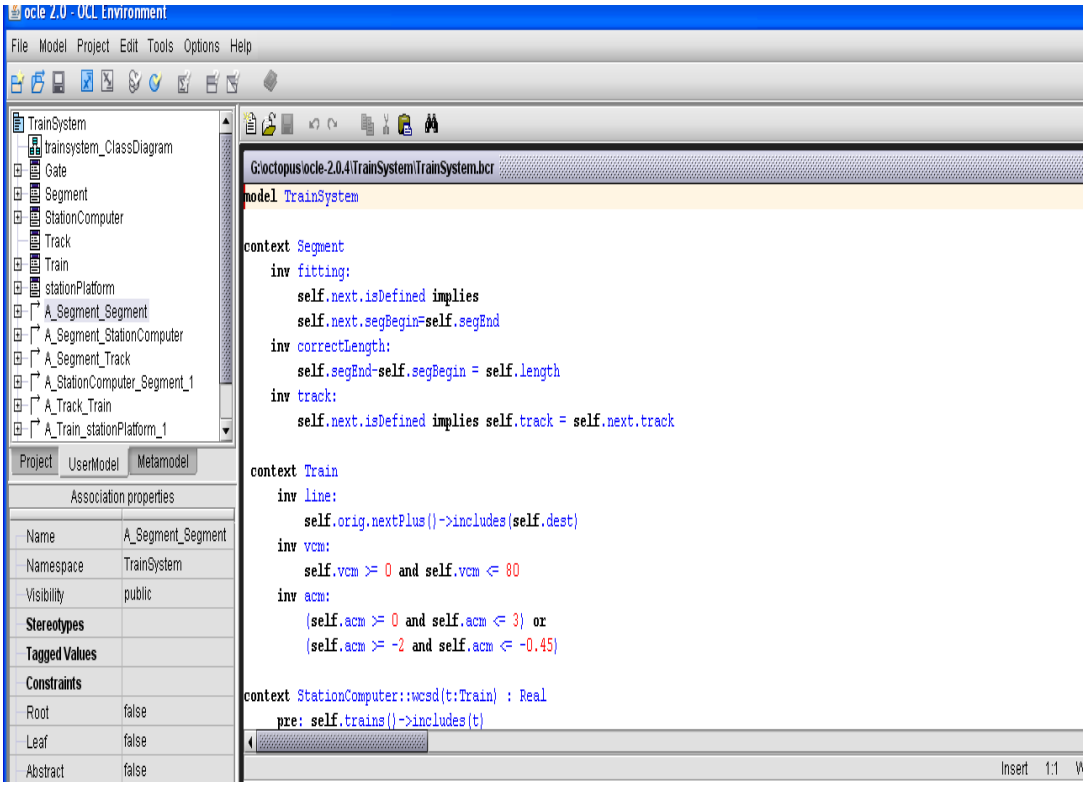


Figure 4.3: OCL Constraints and operations of a Train System in OCLE2.0.4

- A station computer is able to calculate the worst case stopping distance only of trains that are in the region the computer is responsible for:

context StationComputer::wcsd(t:Train) : Real

pre: self.trains()->includes(t)

The same holds for the controlling of trains:

context StationComputer::control(t:Train)

pre: self.trains()->includes(t)

- A station computer is responsible for at most 20 trains:

context StationComputer::trains() : Set(Train)

post: result->size() <= 20

• The segments that bound the region a station computer is responsible for are connected.

context StationComputer

inv bounderies: self.sb.nextPlus()->includes(self.se)

The last three invariants form the central part of the specification. They formalize the main constraints of the train system. The control() operation has to be designed in a manner that assures that these invariants do not fail.

• The constraint civilSpeedSafety demands that a“train should stay below the maximum speed that segment of track can handle.

context StationComputer

inv civilSpeedSafety:

self.trains()->forAll(t | t.v <= t.currentSeg().civilSpeed)

• A train should not enter a closed gate. The invariant with the name closedGate-Safety says that if a next closed gate exists, the distance to it is greater than the worst case stopping distance of the train, i.e., the train can stop in time.

context StationComputer

inv closedGateSafety:

self.trains()->forAll(t | t.nextClosedGate().isDefined implies t.nose+self.wcsd(t) < t.nextClosedGate().segment.segEnd)

• A train should never get so close to a train in front that if the train in front stopped suddenly the (following) train would hit it .The invariant crashSafety says (analogo usly to the preceding invariant) that if a train in front exists, the distance to it is greater than the worst case stopping distance of the (following) train.

context StationComputer

inv crashSafety:

self.trains()->forAll(t |t.nextTrain().isDefined implies

t.nose+self.wcsd(t) <t.nextTrain().nose-t.nextTrain().length).

4.2.5 Code Generation from OCL Specification

Generic steps and principles for transformation of OCL code to the code of another language using Visitor patterns:

Start from a rough algorithm and go down over abstraction levels to applying Visitor patterns and templates. In past, Many implementations of Visitor patterns exist, but most of them are doing only instant code generation when each node gets traversed. Such implementations are simpler and easier to develop and maintain, however, the resulting target language code in these implementations is usually redundant and sometimes even incorrect. Observe the segment of code generated by OCLE2.0.4 for Specification in section 4.2.4.

```
/*
 * @(#)Segment.java
 *
 * Generated by <a href="http://lci.cs.ubbcluj.ro/ocle/>OCLE 2.0</a>
 * using <a href="http://jakarta.apache.org/velocity/">
 * Velocity Template Engine 1.3rc1</a>
 */

/**
 *
 * @author unascribed
 */
public class Segment {

    public Train currentTrains() {
        return null;
    }

    public Segment nextPlus() {
        return null;
    }

    public Segment previousPlus() {
        return null;
    }

    public final Gate getGate() {
        return gate;    }

    public final void setGate(Gate arg) {
        gate = arg;    }

    public final StationComputer getStationComputer() {
        return stationComputer;    }

    public final void setStationComputer(StationComputer arg) {
        stationComputer = arg;    }

    public final Track getTrack() {
        return track;    }
```

4.3 Generic Principles for code Improvement

In OCLE2.4.0 the standard Visitor pattern [10] and the standard way of traversing AST trees for generating code from them were presented [36]. It leads to inefficiency and redundancy problems when code gets generated instantly for every traversed node. In this section the conceptual idea of adding attributes to AST tree nodes is presented, so that the generated code would be of better quality [20]. Attributes will be used for analysis of node context to enable early and lazy generation of code thus making it more efficient. So, Visitor patterns can be used for traversing various models [11]. The traversal algorithm relies on meta model and can be easily extended when meta model gets augmented. Visitor Pattern extended Steps in parsing OCL code:

- Construction of CST based on OCL grammar
- CST tree is transformed to the Abstract Syntax Tree(AST) for evaluating expressions[29] or generating code

4.3.1 What is Visitor Design pattern ?

A solution to the problem of adding operations on the elements of an object structure without changing the classes of the elements on which it operates.

4.3.2 Assigning Attributes to Nodes of OCL AST Trees

AST tree nodes (*x*, *y* and *z*), attributes and their relations are depicted in Figure 4.4, where solid arrows represent attribute inter dependencies, and dashed arrows parent child relations between nodes of an AST tree. The attribute concept should preserve its semantics, it is a variable assigned to a node. The concept of synthesized and inherited attributes returned by functions I and S should also be preserved. Inherited and synthesized attributes should be related as follows

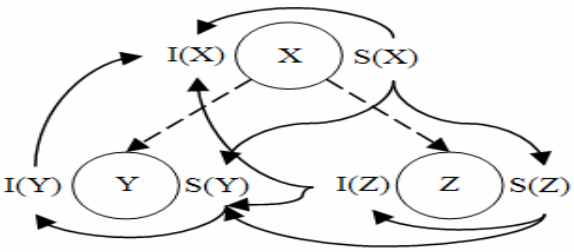


Figure 4.4: Attribute Inter dependencies in AST tree

4.3.3 Defining Attributes Inter dependency rules in AST tree

x, y, z are set of OCL meta model elements

A(m) - all attributes for node m

I(m) and S(m) - sets of inherited and synthesized attributes for node m

$A(m) = I(m) \cup S(m)$.

Attribute evaluation rules (f and g are functions here):

- $S(m) = f(I(m), S(m_1), S(m_2), ..., S(m_n))$

  where n = C(m), C –number of sibling AST tree elements;
- $I(m_j) = g(I(m), S(m_1), S(m_2), ..., S(m_{j-1}))$, for j in 1..n.

4.3.4 Applying rules for the OCL AST tree

$A(x) = I(x) \cup S(x)$

$A(z) = I(z) \cup S(z)$ $S(z) = f_3(I(z), S(y))$

$A(y) = I(y) \cup S(y)$ $S(x) = f_1(I(x), S(y), S(z))$

$S(y) = f_2(I(y))$ $I(z) = g_2(I(x), S(y))$

$I(y) = g_1(I(x))$

4.3.5 Attribute evaluation rules

- Synthesized attributes of node *z* depend on synthesized attributes from left siblings and its own inherited attributes. It means that attributes of node *z* indirectly depend on inherited attributes of node *x*.
- Synthesized attributes of node *y* depend on its own inherited attributes. These attributes can also indirectly depend on inherited attributes of parent node *x*.
- Synthesized attributes of node *x* depend on synthesized attributes of node *y* and *z* and on its own inherited attributes.
- Inherited attributes of node x do not depend on attributes of its child nodes.
- Inherited attributes of node *z* depend on inherited attributes of its parent node *x* and synthesized attributes of its left siblings.

- Inherited attributes of node *y* depend only on inherited attributes of its parent node *x* since there are no left siblings for node *y*.

4.3.6 Code generation using Visitor pattern extended with analysis of context

Visitor pattern should not only match the exact operation that represents the node, but it should evaluate attributes as well. Attributes defined for AST nodes convey information about elements of underlying sub tree that would enable to optimize generation of code, i.e. in many cases code might be fine tuned or not generated at all. Code transformation engines usually use templates to generate pieces of code of the target language. These templates should access all attributes of a node that code is generated for, and generate code not only for the traversed node, but for the whole sub tree.

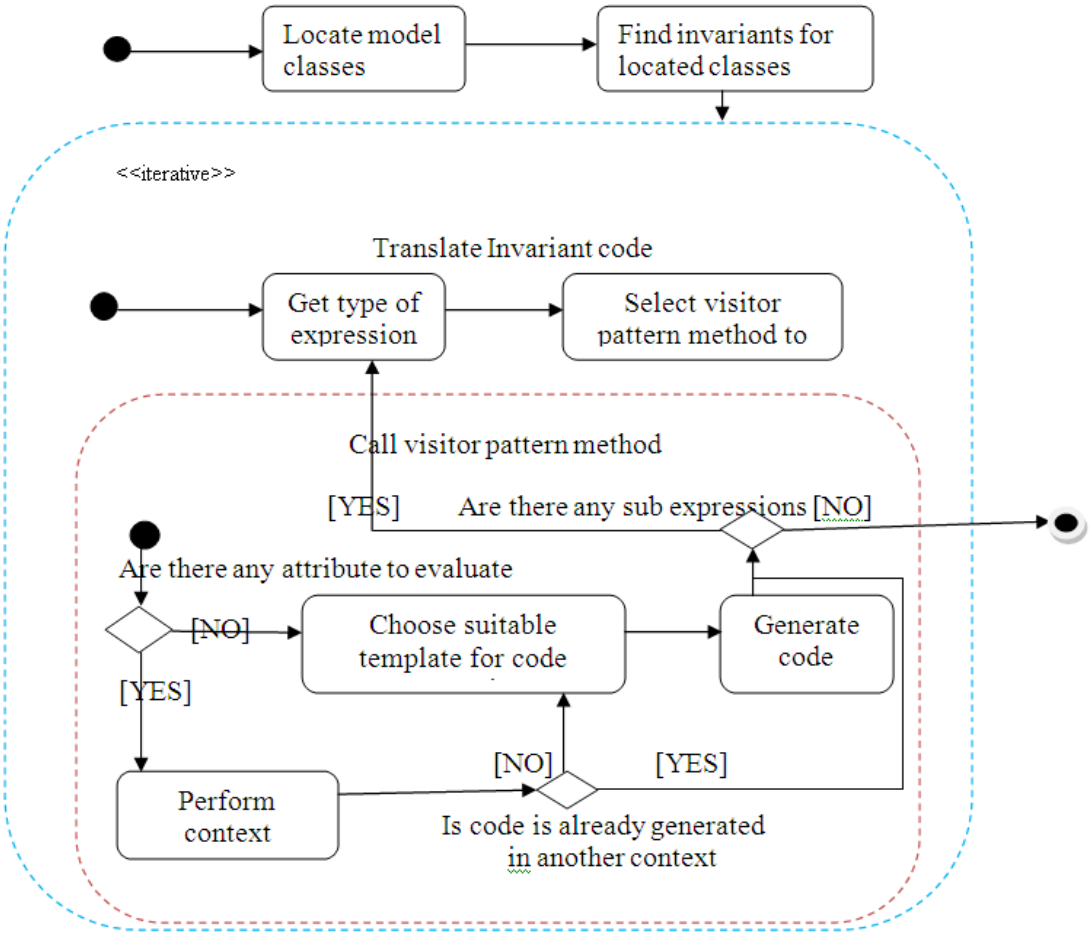


Figure 4.5: Process of Code Generation

Modified process of code generation with context analysis is presented in Figure 4.5. Here every Visitor method has routines that analyze context by acquiring attributes of

the currently traversed node. Depending on analysis results different templates may be chosen for code generation. As mentioned above, it might be that no code will be generated or optimized code will be generated for several nodes at once. Verification and Validation of an implementation is summarized in next chapter.

CHAPTER 5
VERIFICATION AND VALIDATION

5.1 Results of Expressions

Table 5.1:Results of expressions with undefined values in OCLE and USE tools

Expression	Results in OCLE2.0.4	Results in USE2.4.0
Sequence{1..3}->iterate(i; c:Sequence(Boolean)= Sequence{} |c ->including (Sequence{1,8,7}->at(i)> Sequence{2,3,8}->at(i)))	Sequence{false, true, false}	Sequence{false, true, false}.
Sequence{1..3}->iterate(i; c:Sequence(Boolean)= Sequence{} |c ->including (Sequence{1,8,7}->at(i)> Sequence{2,3}->at(i)))	Sequence index out of range	Sequence{false, true, false}.
Sequence{1..3}->iterate(i; c:Sequence(Boolean)= Sequence{} |c ->including (Sequence{1,8,7}->at(i) >Sequence{2,3,oclUndefined(Integer)} ->at(i)))	Sequence{false, true,Undefined}	Sequence{false, true, false}.

Table 5.2: Evaluating Undeterministic Operations

Expression	Results in OCLE2.0.4	Results in USE2.4.0
Set{9,1,7,oclUndefined (Integer)}->any(true)	1	oclUndefined(Integer)
Set{9,1,7,oclUndefined (Integer)}->asSequence()	Sequence{9,1,7,ocl Undefined(Integer) }	Sequence{oclUndefined (Integer),1,7,9}

5.2 Verification of a Train System

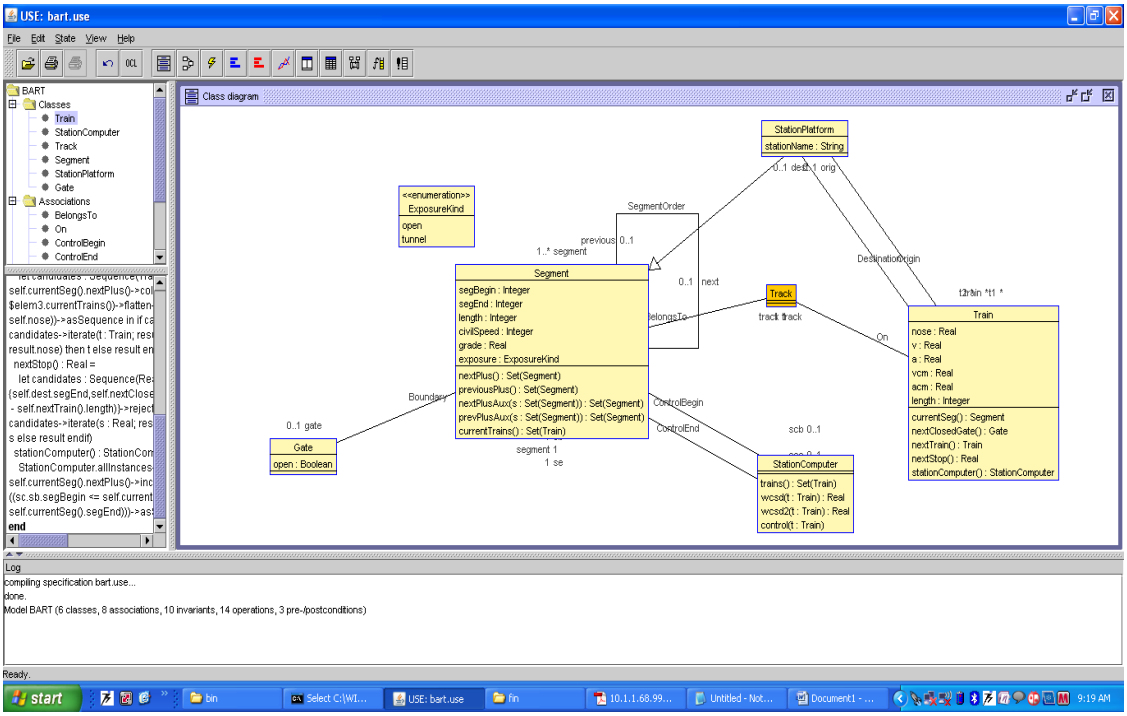


Figure 5.1: Checking Structure of a problem

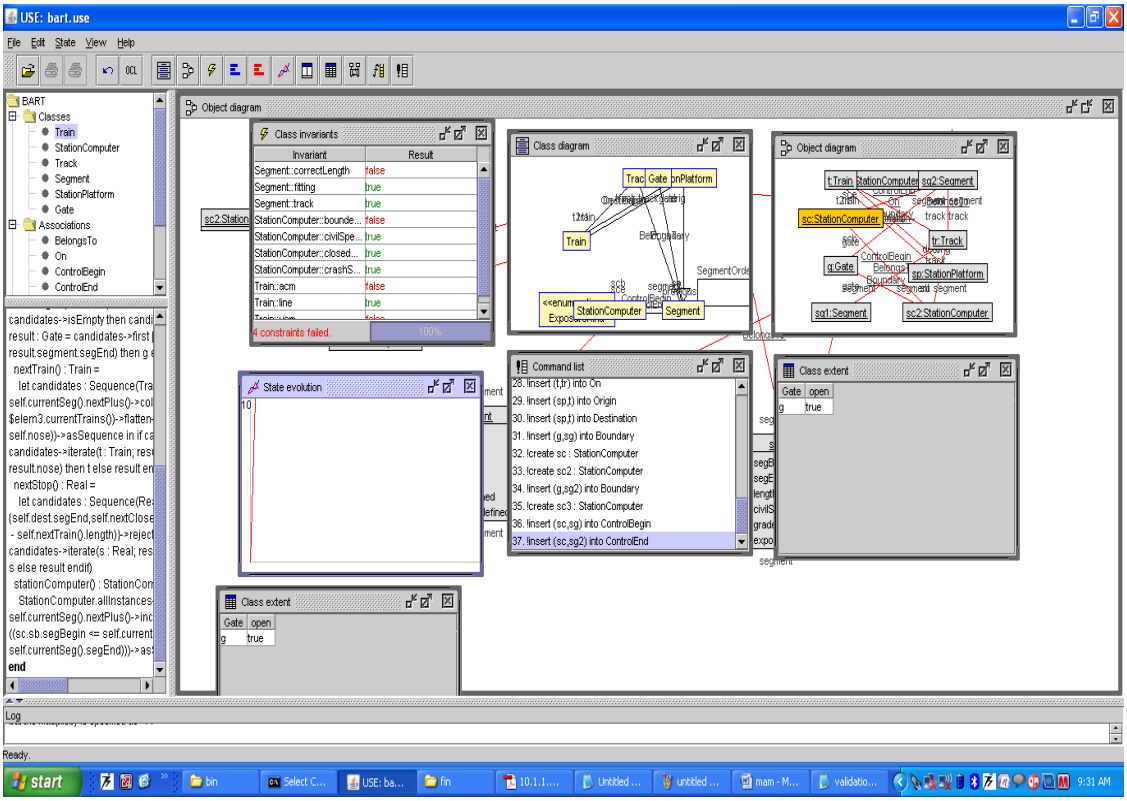


Figure 5.2: Generating different views represented by Object model

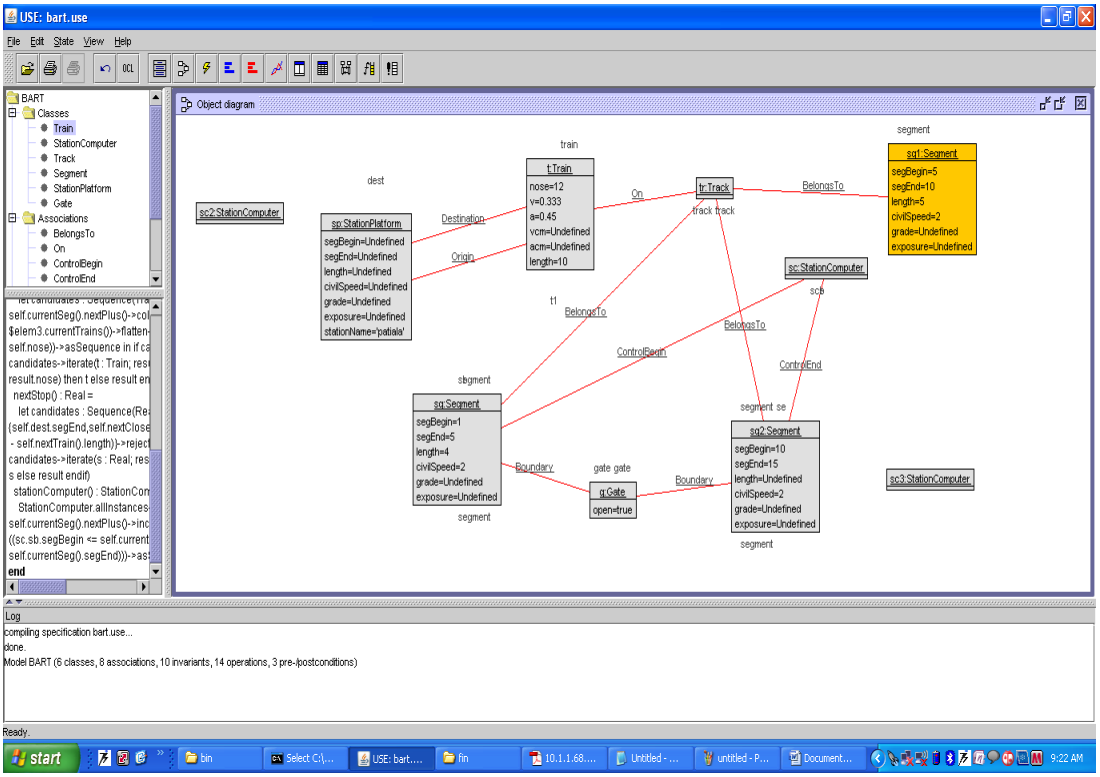


Figure 5.3: Generating State of an Objects

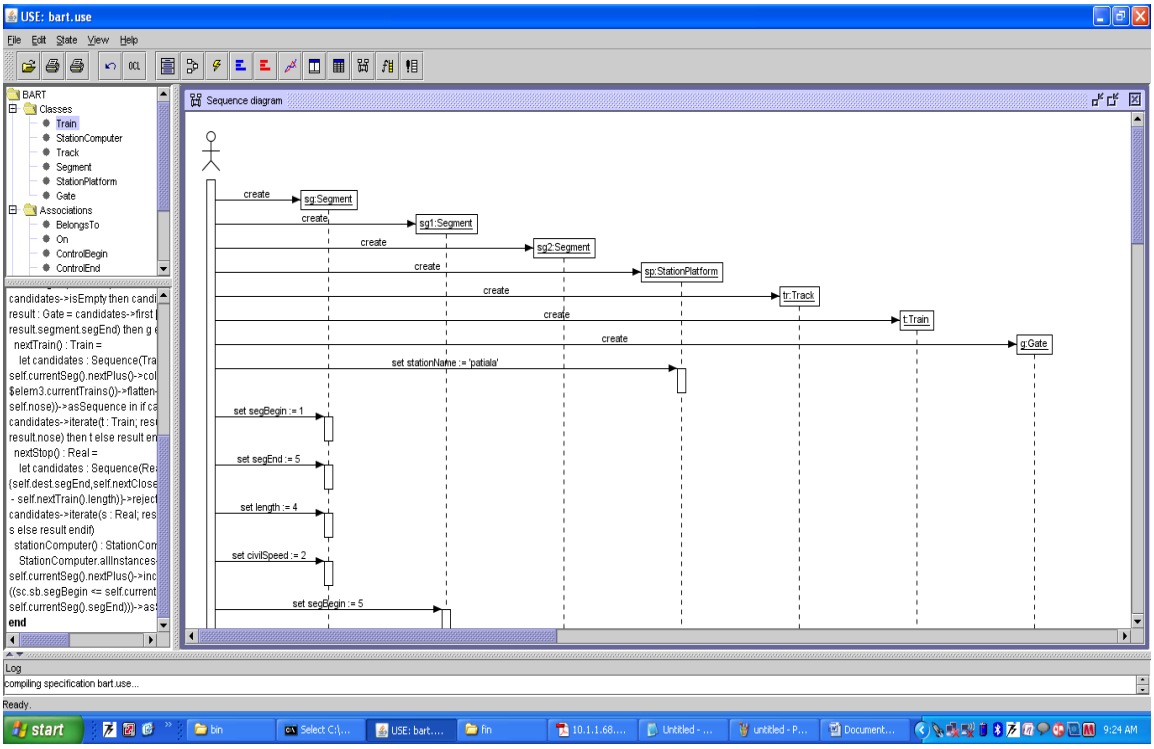


Figure 5.4: Sequence flow according to Object model

5.3 Validation of a Train System:

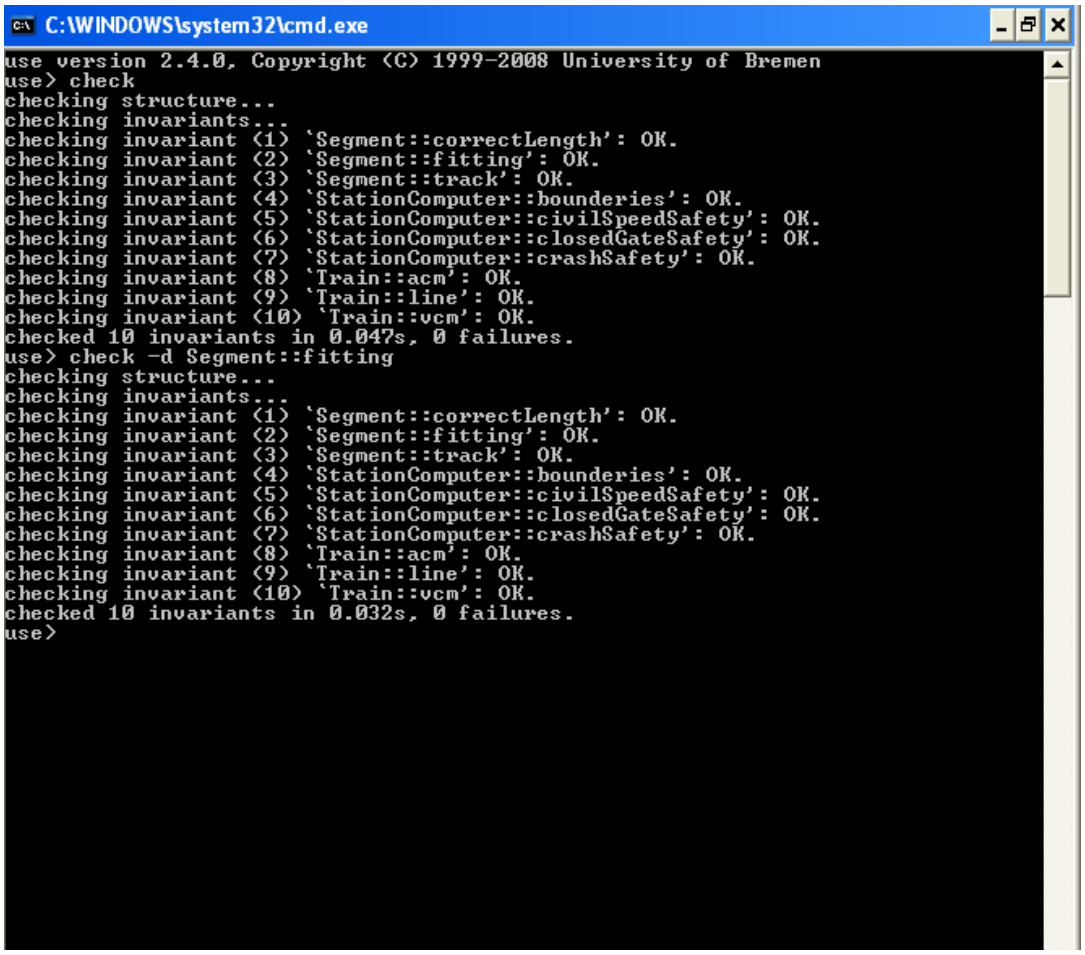


Figure 5.5: Invariants checking

use version 2.4.0, Copyright (C) 1999-2008 University of Bremen

use> !create sg=Segment

<input>:1:1: unexpected token: create

use> !create sg:Segment

use> !create sg1:Segment

use> !create sg2:Segment

use> !create sp:StationPlatform

use> !create tr:Track

use> !create t:Train

use> !create g:Gate

use> !set sp.stationName:='patiala'

use> !set sg.segBegin:=1

use> !set sg.segEnd:=5

```
use> !set sg.length=4
<input>:1:15: expecting ':=', found 'end of file or input'
use> !set sg.length:=4
use> !set sg.civilSpeed:=2
use> !set sg1.segBegin:=5
use> !set sg1.segEnd:=10
use> !set sg1.length:=5
use> !set sg1.civilSpeed:=2
use> !set sg2.civilSpeed:=2
use> !set sg2.segBegin:=10
use> !set sg2.segEnd:=15
use> !set t.nose:=12
use> !set t.v:=0.333
use> !set t.a:=0.45
use> !set t.length:=10
use> !set g.open:=true
use> !insert(sg,tr)into BelongsTo
use> !insert(sg1,tr)into BelongsTo
use> !insert(sg2,tr)into BelongsTo
use> !insert(t,tr)into on
<input>:1:18: Association `on' does not exist.
use> !insert(t,tr)into On
use> !insert(t,sp)into
<input>:1:13: expecting an identifier, found 'end of file or input'
use> !insert(t,sp)into Origin
Error: Argument #1 of insert command does not evaluate to an object of class `St
ationPlatform', found `@t : Train'.
use> !insert(sp,t)into Origin
use> !insert(sp,t)into Destination
use> !insert(sg,g)into Boundary
Error: Argument #1 of insert command does not evaluate to an object of class `Ga
te', found `@sg : Segment'.
use> !insert(g,sg)into Boundary
use> !create sc:StationComputer
```

```
use> !create sc2:StationComputer
use> !insert<g,sg2>into Boundary
<input>:1:7: expecting '(', found '<'
use> !insert(g,sg2)into Boundary
use> !create sc2:StationComputer
Error: Object `sc2' already exists.
use> !create sc3:StationComputer
use> !insert(sg,sc)into ControlBegin
Error: Argument #1 of insert command does not evaluate to an object of class `St
ationComputer', found `@sg : Segment'.
use> !insert(sc,sg)into ControlBegin
use> !insert(sc,sg2)into ControlEnd
use> check
checking structure...
Multiplicity constraint violation in association `BelongsTo':
  Object `sp' of class `StationPlatform' is connected to 0 objects of class `Tra
ck'
  but the multiplicity is specified as `1'.
Multiplicity constraint violation in association `Boundary':
  Object `g' of class `Gate' is connected to 2 objects of class `Segment'
  but the multiplicity is specified as `1'.
Multiplicity constraint violation in association `ControlBegin':
  Object `sc3' of class `StationComputer' is connected to 0 objects of class `Se
gment'
  but the multiplicity is specified as `1'.
Multiplicity constraint violation in association `ControlBegin':
  Object `sc2' of class `StationComputer' is connected to 0 objects of class `Se
gment'
  but the multiplicity is specified as `1'.
Multiplicity constraint violation in association `ControlEnd':
  Object `sc3' of class `StationComputer' is connected to 0 objects of class `Se
gment'
  but the multiplicity is specified as `1'.
Multiplicity constraint violation in association `ControlEnd':
```

```
Object `sc2' of class `StationComputer' is connected to 0 objects of class `Se
gment'
  but the multiplicity is specified as `1'.
checking invariants...
checking invariant (1) `Segment::correctLength': FAILED.
  -> false : Boolean
checking invariant (2) `Segment::fitting': OK.
checking invariant (3) `Segment::track': OK.
checking invariant (4) `StationComputer::bounderies': FAILED.
  -> false : Boolean
checking invariant (5) `StationComputer::civilSpeedSafety': OK.
checking invariant (6) `StationComputer::closedGateSafety': OK.
checking invariant (7) `StationComputer::crashSafety': OK.
checking invariant (8) `Train::acm': FAILED.
  -> false : Boolean
checking invariant (9) `Train::line': OK.
checking invariant (10) `Train::vcm': FAILED.
  -> false : Boolean
checked 10 invariants in 0.047s, 4 failures.
use>
```

First specify a UML class diagram and OCL constraints and verified the structure as shown in Figures 5.1, 5.2 then on the one hand create object diagrams that represent valid system states and on the other hand create objects diagrams that represent invalid system states as shown above. Then it is checked with USE whether those system states are really valid or invalid in the specified system as above, i.e., the specification is validated. To validate the specification in this way, it is necessary to create system states as shown in Figure 5.3 that can confidently be regarded as “has to be valid” or “must not be valid”. Once the specification is considered as correct, more complex system states that are not easily understandable can be checked with respect to the specification by generating sequence of snapshots for different Object instances.

CHAPTER 6

CONCLUSIONS AND FUTURE SCOPE

6.1 Conclusions

The results presented in this thesis highlight some aspects related to the improvement of OCL specifications in problem requirement Specifications.

- A precise definition of OCL avoiding ambiguities under specifications and contradictions was given. Several lightweight extensions were improved the orthogonality of the language like in case, when it is expected to obtain undefined values recommend to take a more secure approach, namely using the standard oclIsUndefined() operation and proposed approach to accessing features with the same name, from ascendants in section 4.1.3.

- OCLE2.0.4 generates glue source code in java. So, for efficient code generation implemented some generic principles using Visitor pattern extended with analysis of context at attribute level.

- By implementing a Case Study experience proved that validating models before compiling and evaluating OCL specifications is mandatory. As WFRs are specified in OCL, the model conformance with its modeling languages rules is a required functionality for all tools supporting OCL. The model description is realized by weaving the specification made in the modeling language UML with the OCL specification made in OCLE2.0.4.

- The USE2.4.0 tool has been used to validate the well formedness rules in the UML standard and OCL specifications of a Case Study for checking Safety requirements of a train system by generating snapshots for each and every state of an objects. So, it is popular as a validation tool in industries.

6.2 Future Scope

Future work will consider the further development of expression evaluation operators in depth. It means evaluations of an expression should be done in attribute level. In this way we get better results in code generation and in evaluation of expressions in OCLE2.0.4. The results shown in Section 5 evaluating pre and post conditions in the command window could be represented in the GUI. Invoking and finishing operations could also be supported interactively. In order to increase the computational power of operations in USE2.4.0 one could consider efficient programming languages, i.e. features like conditions and loops, for the command interpretation. Moreover, tools must implement functionalities related to the support for specifications reuse at M2 and M3 levels.

REFERENCES

[1] Bastide, Rémi, and Philippe Palanque, "Petri Net Objects for the Design Validation and Prototyping of User-Driver, Interfaces", 3rd IFIP Conference on Human Computer Interaction, Cambridge, UK, Aug 1996.

[2] Girish Keshav Palshikar, "Applying Formal Specifications to Real World Software Development", IEEE, 2001.

[3] University of Bremen, "The USE tool", Available at: http://www.db.informatik.unibremen.de/projects/USE .

[4] Aline Lucia Baroni, Fernando Brito Abreu, "Formalizing Object Oriented Design Metrics upon the UML Meta Model", FCT/Universidade Novade Lisboa Departamento de Informática Monte da Caparica, Portugal,2004.

[5] "UML 2.0 ,OCL2.0 Specification", final rtf/ftf report, OMG Document formal, June 2005.

[6] Richters M, "UML-based Specification Environment", 2001available at: http://www.db.informatik.uni-bremen.de/projects/USE.

[7] Richters M, Gogolla M, "OCL – Syntax Semantics and Tools Advances in Object Modelling with the OCL", LNCS, vol.2263. Springer, Berlin, 2001.

[8] Ambrosio Toval, Victor Requena, Jose Luis Fernandez, "Emerging OCLtools", available at http://www.um.es/giisw , Springer-Verlag 2003 .

[9] Claas Wilke, Michael Thiele, "Dresden OCL Manual for installation, use and Development", available at http://dresden-ocl.sourceforge.net/ .

[10] Gamma E, Helm R, Johnson R, Vlissides J, "Design Patterns - Elements of Reusable Object-Oriented Software", Addison Wesley,1994, pp. 319-331

[11] Buttner F, Radfelder O, Lindow A, Gogolla M, "Digging into the Visitor Pattern", Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering SEKE-2004, ISBN 1-891706-14-4., 2004, pp. 135-141.

[12] Claas Wilke, "Model Based Run Time Verification Of Software Components By Integrating OCL Into Treaty", Submitted at Dresden University Software Group on September 3, 2009.

[13] Chiorean, Borte D, Corutiu M, "Proposals for a Widespread Use of OCL", In Proceedings of the MODELS-05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica (2005), Technical report LGL-REPORT-2005,available at: http://lgl.epfl.ch/members/baar/oclwsAtModels05/technicalReport.pdf .

[14] Oliver I, "Using Class Relationships for Identifying Invariants", Technical report NRC-TR-200, Nokia Research Center (2006).available at: http://research.nokia.com/files/NRC-TR-2006-006.pdf.

[15] Gogolla M, Kuhlmann M, Buttner F, "A benchmark for OCL Engine Accuracy, Determinateness, and Eciency", Proc. 11th Int.Conf. Model Driven Engineering Languages and Systems (MoDELS-2008), Springer, Berlin (2008).

[16] "OCL Specification v2.0", available at: http://www.omg.org/docs/formal/.pdf .

[17] OCLE web page, http://lci.cs.ubbcluj.ro/ocle/index.htm.

[18] Richters M, Gogolla M, "Validating UML model and OCL constraints", vol.3263. Springer, Germani, 2002.

[19] Martin, Gogolla M, Jorn Bohling, Richters M, "Validating UML and OCL models in USE by automatic snapshot generation" ,Springer-Verlag 2005.

[20] Martin R, "Acyclic visitor Pattern Languages of Program Design". Addison Wesley Publishing Co., Reading, MA, 1998, pp. 93-104.

[21] Jacobsen I, Christerson M, Jonsson P, and Overgaard G, "Object Oriented Software Engineering". Addison-Wesley, 1992.

[22] Object Management Group, "OMG Unified Modeling Language Specification", Version 1.5, available at : http://www.omg.org ,2003.

[23] Richters M, "A Precise Approach to Validating UML Models and OCL Constraints", PhD thesis, Universit at Bremen, Logos Verlag, Berlin, 2002.

[24] Richters M, Gogolla M, "A Metamodel for OCL", Proc. 2nd Int. Conf. Unified Modeling Language (UML'99), pages 156–171. Springer, Berlin, 1999.

[25] USE web page, http://www.db.informatik.uni-bremen.de/projects/USE/ .

[26] Rumbaugh J, Blaha M, Premerlani W, Eddy E, and Lorensen W. "Object Oriented Modeling and Design". Prentice Hall, Englewood Cliffs, 1991.

[27] Richters M, Gogolla M, "OCL-Syntax, Semantics and Tools", Springer, Berlin, LNCS 2263,2001.

[28] Warmer J, Kleppe W, Clark T, Ivner A, Hogstrom J, Gogolla M, Richters M, Hussmann H, Zschaler S, Johnston S, Frankel D, and Bock C, "Object Constraint Language 2.0". Technical report, Submission to the OMG, 2001.

[29] Gagnon E, "An Object-Oriented Compiler Framework", Master's Thesis, McGill University, Montreal, 1998

[30] OMG, "Meta Object Facility (MOF) Specification", Version 1.3 RTF, Inc., available at: http://www.omg.org , 2 July 1999.

[31] OMG, "Object Constraint Language Specification, OMG Unified Modeling Language Specification", Version 1.3, chapter 7, June 1999 .

[32] Warmer J, Kleppe A, "The Object Constraint Language:Precise Modeling with UML". Addison-Wesley, 1998, Pages 1, 12, 14,19, 85, 123, 135.

[33] OMG. "XML Metadata Interchange (XMI) Version 1.1", October 25, 1999, Inc.,available at: http://www.omg.org , 1999.

[34] Mandana Vaziri and Daniel Jackson, "Some Short Comings of OCL", The Object Constraint Language of UML(pdf),2000.

[35] Akehurst D, Patrascoiu O, "OCL 2.0 Implementing the Standard for Multiple Metamodels", UML 2003. ENTCS vol. 102, 2003, pp. 21-41.

[36] Dan Chiorean, Maria Bortes, Dyan Corutiu, and Radu Sparleanu, "UML/OCL tools - objectives, requirements, state of the art - the OCLE experience", In Proceedings of the NWPER-2004, pages 163–180.

[37] David Navarre, Philippe Palanque, and Rémi Bastide, “Reconciling Safety and Usability Concerns through Formal Specification based Development Process”, HCI-02 Proceedings, 2002.

LIST OF PUBLICATIONS

Published:

- Sunil Babu D, Shivani Goel , "Refining OCL for use in its Supporting tools", National Conference on Next Generation Computing, Gurgoan,20 March 2010.