The University of York Department of Computer Science

Submitted in partial fulfillment for the degree of MSc SWE.

Transforming OCL to PVS: Using Theorem Proving Support for Analysing Model Constraints

Lukman Ab. Rahim

14th September 2007

Supervisor: Dr. Richard Paige

Number of words = 29686, as counted by the MS Word word count command.
This includes all the body of the report, but not the references and appendices.

ABSTRACT

The Unified Modelling Language (UML) is a de facto standard language for describing software systems. UML models are often supplemented with Object Constraint Language (OCL) constraints, to capture detailed properties of components and systems. Sophisticated tools exist for analysing UML models, e.g., to check that well-formedness rules have been satisfied. As well, tools are becoming available to analyse and reason about OCL constraints. Previous work has been done on analysing OCL constraints by translating them to formal languages and then analysing the translated constraints with tools such as theorem provers.

This project contributes a transformation from OCL to the specification language of the Prototype Verification System (PVS). PVS can be used to analyse and reason about translated OCL constraints. A particular novelty of this project is that it carries out the transformation of OCL to PVS by using *model transformation*, as exemplified by the OMG's Model-Driven Architecture. The project implements and automates model transformations from OCL to PVS using the Epsilon Transformation Language (ETL) and tests the results using the Epsilon Comparison Language (ECL).

ACKNOWLEDGEMENT

I would like to express my highest gratitude to people who have helped me in completing this dissertation. Especially to my supervisor, Dr Richard Paige, who has helped me countless times by giving me insights and guidance. To Mohd Ibrahim Abdul Razak that willingly spending his time to proof read this dissertation.

A special thanks to Petronas University of Technology. Without them I would not be here studying my MSc.

Finally, to my lecturers for their knowledge and to all my friends that have stood by me during the good times and bad times. I am indebted to you all.

TABLE OF CONTENTS

1 INTRODUCTION

The Unified Modelling Language (UML) [68] is one of the most widely used modelling languages for developing software. The current version of UML (UML2) has moved a long way since its initial version. It incorporates more notations to support modelling of different types of software with different levels of complexity.

Although UML has been modified to better meet the current need of software developers, it has been proved that diagrammatic modelling language such as UML are insufficient to represent all the information and design decisions relevant to a system design. The Object Constraint Language (OCL) [57, 62] was created to fill in this gap by providing a formal text-based syntax to represent constraints on element of a UML model.

To ensure the correctness of software being developed, OCL constraints should also be verified along with the UML model to which they apply. Currently, active research focuses on finding suitable methods for verifying UML models and OCL constraints. Two widely used methods are model checking and theorem proving.

Model checking represents OCL constraints in temporal logics such as Linear Temporal Logic (LTL) or Computational Tree Logic (CTL) while theorem proving approaches represent OCL using formal languages such as B-method [22], Z [31, 32], Isabelle/HOL [21, 29] and others. A more thorough study on different methods proposed to verify OCL and UML will be discussed in the Literature Review chapter.

Although there exists a substantial body of previous work on verifying OCL constraints through the means of theorem proving, much of this work did not specify how the transformation from OCL to the formal languages used by theorem provers is to be carried out. What are the techniques, languages or tools used to transform OCL into the formal languages? As of now only the work by Marcano and Levy [53] rigorously specifies their method of translation, which is using a tool (OCL2B) that translates OCL to B specification. The translation is done using OCAML language.

To our knowledge, there is no translation of OCL to any formal languages using the *model transformation* method. Being able to transform OCL (or any model) to formal languages

using model transformation will make Model Driven Engineering (MDE) a rigorous approach in software development. Using the same approach to create and verify the models allow the streamlining/integrating the process of creating a model and verifying it. Thus, it minimise the development time and reduce the cost for buying development tools.

There is research such as [50] that tries to represent OCL in Prototype Verification System specification language (PVS) [26, 64] for the purpose of proving properties about OCL constraints but the work concentrates more on resolving the differences in mapping from First-Order logic to Higher-Order logic than the mapping of OCL to PVS. This project will transform OCL to PVS using model transformation, study the suitability of PVS in representing OCL constraints, identifying the process of analysing OCL through model transformation and finding the most suitable type of transformation. We also concentrate more on the mapping of OCL to PVS.

What makes this project interesting and at the same time challenging is the novelty of our method. We chose to use model transformation, a new technique that is still undergoing research. Model transformation is a technique originally proposed by Object Management Group (OMG). It is part of the Model Driven Architecture (MDA) [66] initiative. But it has not been effectively applied, yet, to transformations involving formal languages such as PVS.

Tools and model transformation languages exist but most of them are still prototypes. There are several model transformation languages but there is no research or case studies similar enough to help in selecting the most suitable transformation language. Model transformation is still in its infancy in both research and implementation, and has a long way to go. The use of model transformation as part of the process to analyse model is not the original purpose of its creation, thus improving the value this project.

Another challenge exhibits from using PVS and OCL together. Both language are formal languages but developed for different purpose by different groups. OCL was created for developers to write constraints on elements in UML model while PVS specification language was created for proving properties of a theory. Both languages have manuals and guidelines to explain their syntax and semantics. OCL abstract syntax and semantics are explained using UML models while PVS's syntax is describe in Extended Backus-Naur Form (EBNF) and its semantics in set theory. The transformation is not a straight forward process since PVS does not have a metamodel that explains its syntax. Depending on the type of transformation, metamodel for PVS may need to be created. Considering the complexity and size of PVS syntax, the creation of PVS metamodel is no easy task.

OCL and PVS also used different kinds of logic: OCL using First-Order logic while PVS uses High-Order logic. This may add to the complexity of transforming OCL to PVS.

1.1 Project Background

This project aims to provide support for analysing properties written as OCL constraints by transforming OCL to PVS and used the PVS theorem prover to reason about the result of the transformation. Figure 1.1.1 illustrates the process of reasoning OCL properties using our proposed method.

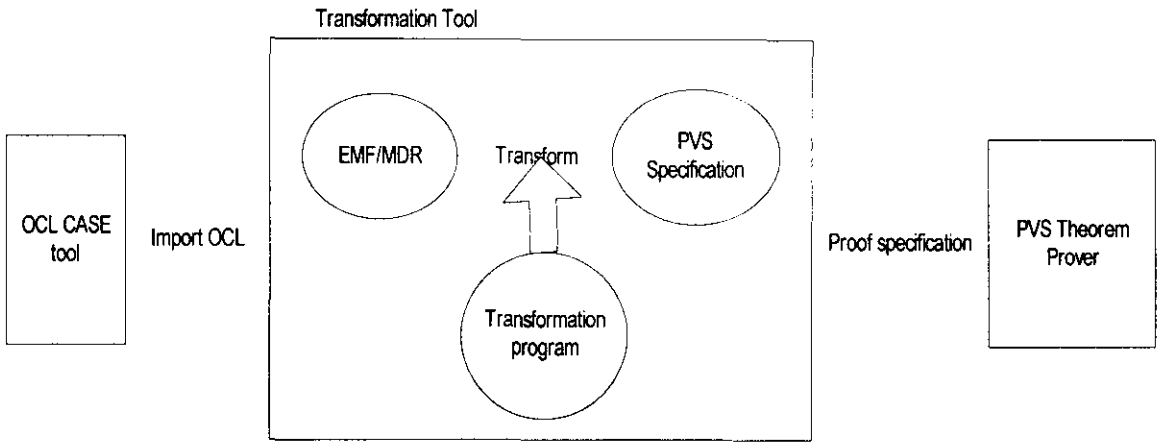


Figure 1.1.1: Process to verify OCL through transformation

Our proposed process starts by creating the OCL using OCL CASE tools such as Octopus and then importing the OCL constraint into a transformation tool. In the transformation tool the OCL is expressed in the form of Eclipse Modelling Framework (EMF) or Metadata Repository (MDR) model which from now on will be referred to as OCL model. The OCL model will be transformed into a PVS specification using transformation program written in a transformation language. The PVS specification will then be analysed by the PVS theorem prover. PVS can therefore provide validation and verification for the transformations.

From Figure 1.1.1, the rectangle at the centre is the focus of this project. This project will develop transformation program that transform EMF/MDR model of the OCL into PVS. As mention earlier model transformation is part of MDA and it plays a vital role in achieving MDA's objectives. MDA's main objective is to upgrade the status of models in software development, not just a medium to exchange information or documenting design and ideas. Models going to be first class artefact that can be used throughout the software production chain [65]. From this objective the intended used of model transformation is to help to iteratively refine models created in early phase of software development into detailed models of the system and finally to a working product.

In [66] different approaches in model transformation are explained along with the definition of important elements in the transformation. Table 1 in Appendix E is the definition of these elements. There are five approaches to model transformation:

1. **Marking**: Transform PIM, which is marked using the given marks, to PSM using the mapping provided.
2. **Metamodel Transformation**: The mapping specifies the transformation from source metamodel to target metamodel. PIM that conforms to source metamodel will be transformed to PSM that conforms to target metamodel.
3. **Model Transformation**: The mapping specifies the transformation from platform independent types to platform specific types. The PIM contains elements that are a subtype of elements in platform independent types while the PSM contains elements that are a subtype of elements in platform specific types. The mapping will transform PIM to PSM.
4. **Pattern Application**: Applying transformation patterns that is supplied with the mapping, or marking PIM with pattern names and transformed the marked PIM to PSM based on the patterns.
5. **Model Merging**: PSM is created by merging PIMs with other models.

Figures in Appendix E show the five approaches.

One of the milestones of MDA is to provide automatic transformation tools to users. Currently there are a number of transformation tools that have its own transformation languages. We can group these languages into two major groups: model-to-model transformation language and model-to-text transformation language. Model transformation languages will be discussed in more detail in Chapter 2.

1.2 Dissertation Outline

This chapter explains the objective of this project and background information on this project. The next chapter is Literature Review where it focuses on previous work in verifying OCL and UML models, formal language and theorem provers, and model transformation languages. In Chapter 3, project requirements will be identify along with the implementation plan. We will also discuss issues that need to be resolve before the start of design and implementation phase and choose our project methodology. Chapter 4 will explain the design and implementation of our transformation and in Chapter 5 result of testing the transformation will be discuss. Discussion on the transformation will be discussed in Chapter 4 and will be revisited again in Chapter 5. Chapter 5 will also evaluate the whole project. The project will be concluded in Chapter 6.

Next Chapter: Literature Review

2 LITERATURE REVIEW

There are three main topics of interest that relate or contribute (based on the chosen solution) to this project. These topics are model transformation technology, theorem proving tools and verification techniques for OCL.

Section 2.1 will discuss different methods in verifying UML models and OCL constraints. A lot of attention is given to UML models, other than OCL, because OCL is developed to complement UML models, particularly in order to show information that cannot be represented using UML notations. Thus the verification of OCL may require the work of verifying UML models, as what has been done in previous work [50, 52, 53, 54 and 55]. OCL is mainly used in class diagrams but in this section research work on verification and checking will also touch on other models in UML.

The second section of this chapter will explain about automated theorem proving. Discussion in this section will be divided into two parts, tools and specification language. Tools of interest here are PVS, HOL System (HOL-S) and Isabelle/HOL (Isabelle). PVS is the main focus in this section. HOL-S and Isabelle are two theorem provers that similar to PVS uses Higher-Order Logic (HOL). PVS also includes a specification language and in this section, PVS specification language will not be discuss in detail. However, B and Z specification language will be discuss because these two languages have been used for verifying models. Final word about this section is it will not discuss any of the tools and specification languages in detail and will not compare PVS, HOL-S and Isabelle, or Z and B. Aim of this section is to research other tools and specification languages other than PVS and discuss their characteristic.

Model transformation is one of the main contributing elements of the Model Driven Engineering (MDE) initiative [8]. Section 2.3 is dedicated to discussing the state-of-the-art in model transformation. Although currently model transformation is being done using transformation languages, and discussions in section 2.3 focus on these languages, the phrase model transformation technology is used (instead of model transformation languages) because these languages are now part of a larger framework that includes other model management operations. Examples of model transformation languages include the Atlas Transformation Language (ATL) which is now part of Atlas Modelling Management Architecture (AMMA),

Extensible Platform for Specification of Integrated Languages for Model Management (Epsilon) and Visual Automated Model Transformation (VIATRA). Query View Transformation (QVT), AMMA, Epsilon and VIATRA will be discussed in more detail. A final note about model transformation technologies discussed here is that they are chosen from Eclipse Generative Modelling Technologies (GMT) and Model to Model Transformation (M2M) projects. Technologies within GMT and M2M are preferred in the MDE community because they are well supported with tools, case studies and tutorials.

2.1 UML/OCL Verification

In software engineering, verification means ensuring the correctness of features and behaviours of the product [1]. Previous research has been done in verifying UML models and OCL constraints. The effort to verify UML/OCL is to achieve rigorous software engineering, avoid propagation of design errors to later stage in development and proving properties of UML models.

Most of the research work can be group into two groups, base on the methods that they use in verifying UML/OCL. UML/OCL is verified by mapping UML/OCL to a formal language or temporal logic language. The result of the mapping will be a representation of UML/OCL in the target language, which later can be verify/check/proof by tools.

This section will discuss about various works that maps UML/OCL to four specification language: B, Object-Z, Isabelle/HOL and Prototype Verification System (PVS). Discussion will also include research that verifies UML/OCL using model checkers (temporal logic). For completeness of our research, other works on verification of UML/OCL is also included at the end of this section.

2.1.1 Mapping UML/OCL to B

Verification of UML/OCL by mapping it to B was done by three groups of researchers: 1) Marcano and Levy [53, 54], 2) Souquieres et.al. [34, 42, 55], and 3) Butler and Snook [37]. Work by Marcano and Levy concentrates on the mapping of OCL and UML class model, Souquieres et.al. concentrates more on OCL and UML behavioural models while Butler and Snook covers both UML class model and behavioural model (UML state chart) but not OCL. All three groups also create tools to automatically translate UML/OCL to B specification.

In [53] and [54] the tools created to translate UML/OCL to B is called UML2B that uses OCAML higher order language to write mapping rules shown in Appendix B. As mention earlier the mapping covers only UML class model and OCL expressions but the work lack in discussing the mapping of OCL types to B types. In both [53] and [54] Marcano and Levy does not clearly state the mapping of OCL operations on basic and collection types to B. Only a few operations were shown by examples.

This remark is also agreed by Souquieres and Ledang in [55]. [55] continues the work in [53] and [54] by highlighting and overcoming the shortcoming of the work. The shortcomings are 1) post conditions on behavioural concepts have not been considered, 2) mismatch between OCL and B types are not considered in mapping operations on types, and 3) mapping of all types of OCL collections to a B set [55].

The shortcomings were overcome by specifying in detail the mapping of operations on OCL basic types to operators and expressions in B. Different types of OCL collections are also map differently in [55]. OCL set of T is map to a power set of T, a bag of T is map to a partial function T to a natural number, while a sequence of T is map to B sequence. Different types of collections generated from OCL navigation expressions have also been handled ([53] and [54] only consider navigation expression to return a single element). Additionally [55] also give a mapping of @pre, result, OCL If expression and OCL Let expression.

Souquieres and Ledang also contribute in the research of mapping UML Statechart to B, continuing the work of Meyer, Nguyen and Lano. Their contributions are identifying two types of events (deffered and non deffered events) and proposed a mapping of both type of events to B. Non-deferred events are events that must be handled immediately just after its occurrence and deferred events can be handled in another state [34]. Non-deferred events can be modelled using B implementation construct or B refinement constructs with B inclusion primitives [34]. Deferrable events are modelled by introducing a buffer that stores events when they are deferred and removed from buffer when events are handled. Both non-deferred and deferred events are modelled using B refinement construct because, if compared to implementation construct, refinement construct avoids auxiliary operations [34].

Other contributions of [34] are modelling asynchronous communication between state-charts and a framework that creates specification in B from both structure (class) and behaviour (states) diagrams. The result can be use to check the conformance of both diagrams.

Butler and Snook [37] take a different approach than Souquieres and Ledang [55], and Marcano and Levy [53, 54]. The research presented in [37] does not directly translate UML models to B and in [37] Butler and Snook did not even specify their intension to verify UML models. Their work is more to providing formal modelling capabilities by synchronising UML with B. The synchronisation of B and UML is achieve by first creating a UML profile that used stereotypes to close the gap between elements in UML and elements in B. Second, they create a constraint language called μB (micro-B) that is closer to B expressions than OCL. Using the UML profile and μB language allow the translation of UML models to B specification, which the later can be used for verification. Translation in [37] is also shown in Appendix B and we only took the translation of UML. μB translation is not taken thus translation to important elements such as operations, invariants and post conditions are not specify in Appendix B. Translation of μB is neglected because the language is of no interest in this project.

Both [34] and [37] translate UML state charts in B and the difference between the work of Meyer, Nguyen and Lano (taken from [34]) compared to [37] is the representation of an object state. Both work represent all state of a class as a enumerated set but [37] represent object state as a variable of type one of the element in the enumerated set, while Meyer, Nguyen and Lano (taken from [34]) represent it as a variable that define as a function from variable of type objects that owns the state to the enumerated set. Transitions are modelled as B operation in both, and Meyer, Nguyen and Lano also modelled Actions and Events in the same way [34]. There is no indication of how statechart Action and Event is modelled in [37]. However, [37] did introduce the idea of invariants in states and decision points (similar to the one in UML Activity diagram).

2.1.2 Mapping UML/OCL to Object-Z

A formal mapping of UML has been done by Kim and Carrington in [33]. The mapping is done by first representing elements in UML metamodel in Object-Z. This representation only covers elements in the class model. Second, abstract syntax and semantics of Object-Z language in UML class model is created so that Object-Z syntax will have the same representation as UML syntax. This will simplify the mapping of UML to Object-Z. A representation of Object-Z abstract syntax and semantics in Object-Z specification are created and the mapping is done between both representations of the language in Object-Z. The mapping rules are written as Object-Z specification.

Based on the work in [33], Roe et.al. [52] proposed a mapping of UML and OCL to Object-Z. The mappings of individual elements of UML and OCL are shown in Appendix C. The mapping covers main elements of class model (class, attributes, operations, associations, inheritance, and association class), and OCL expressions (invariants, pre conditions, post conditions, result keyword, @pre). Operations on OCL basic types and collections are mention but examples and explanation are only given to selected operations of collections. OCL expressions that fail to be mapped are OCL IF expression, collect, iterate, subsequence, and some operations on integers and real. There are also no explanations on the compatibility of OCL types to Object-Z types.

2.1.3 Mapping UML/OCL to Higher-Order logic

Previous research have map OCL to two Higher-Order logic (HOL) languages; PVS and Isabelle/HOL. In [50] other than OCL the mapping to PVS also include the mapping of class diagrams and state machines. State machines are mapped into graph while class diagram are mapped into enumerations that contains all the names of classes in the class diagram. Attributes, operations are also mapped to enumerations. The mapping proposed in [50] can only be apply to class diagram that have association with multiplicity 0...1 and does not have generic classes, and flat state machines.

The most important issue that need to be resolve when mapping OCL to PVS, and is explained in detail in [50], is how to represent OCL 3 valued logic in PVS that only have two values for its logic: true and false. Besides true and false, OCL other value is undefined. In OCL undefined value can be a result of using recursive function in OCL, using universal quantifier with allInstances() operation on types (e.g. integer) that creates infinite sets, and accessing undefined values in variables or arrays [50]. Some standard operations provided by OCL also generate undefined value because the operations are partial functions [50].

The solution to map undefined value in PVS in [50] depends on how the value is generated. If the value is generated from partial function operations, domain of the operation will be restricted so that the operation is now a total function. If undefined is the result of a recursive function, the recursive function in PVS must have a ranking function and a termination proof that need to be defined by the user. Using universal quantifiers with allInstances() operation that create an infinite set can be solve by using PVS FORALL expression on a type that is similar to the type in OCL. For undefined value obtain from accessing undefined attributes, [50] only assume that all attributes are defined and have values.

Mapping of OCL to Isabelle/HOL also consider the problem of mapping undefined value [51]. [51] handles undefined value by creating *lifted* type for each type. Lifted types have additional value to represent undefined [51]. Besides undefined other concerns when mapping

OCL to Isabelle/HOL are mutual recursion between object instances, dynamic types and extensible class hierarchies.

Other than [50], mapping of UML state chart to OCL have also being done in [35]. Work done in [35] aim in checking the correctness of UML statecharts compared to its semantics and syntax. To achieve this, three theories in PVS-SL is written for abstract syntax of UML statechart, well-formedness rule of UML statechart and semantics of UML statechart respectively. Formalization steps specified in [35] is supported by a tool called PrUDE (Precise UML Development Environment) [35, 36]. PrUDE provides proof strategies to proof well-formedness and validity of UML models [36].

2.1.4 Model checking UML/OCL

Model checking is a technique to check properties of a model based on properties given by users. A common way to specify properties for this technique is by using temporal logic and the most common temporal logic is Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [46]. Verification using temporal logic usually supported by a model checker like the one specified in [46], [47], [48] and [67].

NuSMV is the model checker use in [46] to verify workflow model with formal requirements. Workflow is modelled using UML activity model and the requirements must be represented in LTL. For NuSMV to be able to verify workflow model, an extension for strong fairness constrain is added to the tool in order to ensure all loops eventually exits [46].

TABU (Tool for the Active Behaviour of UML) is a tool for verifying UML activity model and UML statechart [47]. TABU will take as input a XMI file that is generated by UML modelling tools such as Rational Rose or ArgoUML. XMI will be translated automatically into a formal specification that can be latter use in Symbolic Model Verifier (SMV) tool. To complete the verification, user need to provide properties in temporal logic and TABU provide assistance for this task. SMV can process both LTL and CTL.

Rhapsody is another tool that provides verification of UML models by comparing the model with properties specify in temporal logic. VIS (Verification Interacting with Synthesis) is the tool use for verification and accepts two input, finite state machine (FSM) description of the model and the properties in CTL [48]. The UML models need to be translated into FSM, but instead of translating it directly, Rhapsody translates it to SMI (System Modelling Interface). SMI is preferred over direct translation because translation from SMI to FSM is already available [48]. Properties in CTL are generated from two sources, temporal pattern definition and Life Sequence Charts (LSC). [48] does not indicate specific type of UML model but from the example given, Rhapsody can verify class model and state chart.

SPIN is a well known model checker that checks properties of a model written in PROMELA (Process Meta Language). In [67] a translation of UML state charts to PROMELA is given where each class is translated to a PROMELA process that have only one argument, the instance number of created object. However [67] restrict their research by only testing their translation on state charts used to modelled protocols.

From our study, only one research has been found to check properties of OCL using model checking tools. In [56] a translation of OCL syntax to BOTL is given where BOTL is an object-based extension of CTL. One notable translation of BOTL is to translate OCL collections

(set, sequence and bag) to arbitrary list. Arbitrary list is chosen because it is sufficient to represent sets and bags, and solve the problem of non-determinism [56]. However the use of arbitrary list is only for flatten (non-nested) collections. Also, [56] does not specify any tools that automate the translation of OCL to BOTL or tools that model checked BOTL.

2.1.5 Other verification of UML

One way of verifying UML is to check for consistencies between UML models. Consistencies are a concern for modelling language like UML that allow the representation of a system from multiple views. It is normal to see software being design using UML to have multiple diagrams that represent the software from multiple views.

Model Consistency Checker (MCC) is a tool that checks the consistency of UML model by using UML metamodel to represent domain knowledge and user models as individual knowledge [38]. Individual knowledge is compared with domain knowledge to check consistency of user models. Translation of UML metamodel and user model to knowledge is done with Knowledge Representation System (KRS) and Description Logic (DL) [38]. User models are the input for MCC and can be provided by UML modelling tool such as Poseidon. MCC also used Racer, a DL engine, to check the consistency of models.

Another approach is consistency checking for UML models are through constraint checking. There are two types of constraints for UML, syntactic and semantic constraints [39]. Syntactic constraints are constraints that are written in formal language and can be check automatically while semantic constraints are the same constraints written in natural language. There are many levels of constraints as specified in [39]:

- Paradigmatic level: Constraints on metamodel
- Paradigmatic extension level: Constraints on domain-specific application of model using stereotypes(e.g. UML for real-time or UML for web service)
- Modelling domain level: Constraint on styles of design (e.g. design that will be implement in Java cannot have multiple inheritance)
- Target domain level: Constraint on the design element itself (e.g. attribute studentId in class Student cannot be null)
- Implementation level: Constraint when translating designs to code (e.g. constants in design does not have a value for it to be translate into C++ code)

In [39] a tool called Constraint Checker (CC) is introduced. CC is an expert system that checks constraints of UML model. UML model constraints are translated in production rules and it will be used by inference engine Sherlock for checking systems design. CC will accept a system design in XMI (XML Metadata Interchange) and translates it to knowledge base for Sherlock to diagnose.

Consistency rules in UML can be considered as constraints of paradigmatic or paradigmatic extension level. Constraint that is normally specified using OCL in a UML model is of target domain level.

Malgouyres and Motet [40] formalize UML consistency rules in CLP (Constraint Logic Programming). Consistency is check by first taking a UML model and coding it in CLP. CLP representation of the UML model with consistency rules (also in CLP) will be input to CLP solver for diagnosis. In [40] to fully automate the process of checking consistency of UML model, UML metamodel and MOF is also formalise in CLP. Formalizing UML metamodel and

MOF allow the inheritance of consistency rule used for checking them to be re-used in checking consistency of UML model [40].

Another tool that checks the consistency of UML models are presented in [41]. The tool takes XMI files that have been added with semantics for each element and compare it with Specification Language for XML Semantics (SLXS), a XML document that contains the consistency rule [41]. Adding semantics to XML element is base on XML Semantics technique (propose to check the semantic of XML document) and the semantics are added as attributes to the element [41].

Besides work to check the consistency of UML diagrams other works exists that did not fall into any of the categories we specified above. A framework called Property Verification Framework (PVF) is proposed in [43]. PVF verifies UML models from properties that are taken from UML abstract syntax, well-formedness rules and semantics of UML, same as [42]. The framework is integrated with RIVIERA, a UML modelling tools. User can select which property to be check and verification will be done by calling internal methods or using theorem provers. There is no theorem provers integrated with the framework and there is no translation module (translates UML model to theorem prover language) provided [43]. The only suggestion made for using theorem prover is through RIVIERA that can use MAUDE theorem prover [43].

Another verification for UML based on properties is proposed in [44], but unlike [42] and [43], [44] is aim at Aspect-UML. Aspect-UML is a UML profile for Aspect-Oriented programming. The work in [44] focuses on verifying interactions in Aspect-UML models and separate properties to verify as local properties of global properties. Local properties are pre and post conditions induced by woven aspects or advices, while global properties are invariants for aspects [44]. Alloy [45], a tool for analysing model, is used in verification. Alloy came with a modelling language based on first-order calculus. Verification is done by translating the model into element of Alloy modelling language and properties into Alloy predicate [44].

USE validates UML models by creating snapshot of the system that represents the system state at a particular point in time [49]. The snapshot can be created using ASSL (A Snapshot Sequence Language) that allow the user to create snapshot by specifying the properties of the system [49]. USE work with 4 types of file: [49]

1. USE model (.use): Contains information on classes, associations and invariants
2. command file (.cmd): Responsible in performing the validation test
3. invariant file (.invs): Additional properties of the snapshot
4. ASSL file (.assl): Procedures in constructing the snapshot

The validation starts when the command file loads USE model file. The command file can then load the ASSL file to create a snapshot. Snapshot with specific properties can be created by loading invariants file before loading ASSL file. When loading the ASSL file and running procedure inside ASSL file, USE will display the output which is the existence of valid state for the ASSL and invariants file. If there is a valid state, sequence of operations that leads to the valid state will be displayed.

From all the previous research, this project is most interested in works that maps UML and OCL to formal languages. OCL has been map to B, Object-Z, PVS and Isabelle/HOL. Research on mapping OCL to B and Object-Z mostly concentrates on mapping constraints on classes (invariants) and on operations (pre/post conditions). Mapping of OCL types and its operations are mention but only [55] discusses it in details.

Research mapping OCL to PVS and Isabelle/HOL on the other hand concentrate on handling the differences of First-Order logic (OCL) and HOL (PVS and Isabelle). It concerns so much on this issue that it neglects translation of other parts of OCL such as invariants, pre/post conditions, operations on basic and collection types, and @pre.

Comparing all the formal languages used in representing OCL, representation in Object-Z seems to be the clearest. What it means by clearest is the Object-Z specification generated from the translation can be easily traced back to its UML model and OCL constraints. The reason for this is because Object-Z already supports object-oriented concepts. Other formal languages need to handle these concepts during their translation thus creating complex mappings and specifications.

PVS, B, Isabelle/HOL and Z (parent language of Object-Z) will be discussed in more detail in Section 2.2.

2.2 Automated Theorem Proving

Automated theorem proving is a form of automated reasoning where theorems specified in mathematical notations are verified using software tools. There are three types of tools for theorem proving; model checkers, interactive theorem provers, and automated theorem provers. Table 2.2.1 give a brief description about the three types of tools and examples. The description is taken from [27].

Theorems are normally written in first-order logic (FOL) or higher-order logic (HOL). FOL, also known as predicate calculus, is an extension of propositional calculus with quantifiers (existential and universal) and variables in propositions, while HOL is the combination of logic and functional programming [21]. Another definition of HOL is a predicate calculus where extension of variables can range over function and predicates, logic is typed, and there is no separate syntactic category of formulae [28].

Table 2.2.1 : Summarisation of different types of tools for theorem proving

Type	Description
Model Checker	Decision procedures for temporal propositional logic. Suitable to show properties of finite state machines. Limitations in the size of state space that a model checker can handle. Model checkers can check large formulas of easy proofs. Examples are SMV, SPIN, Step and Murphi.
Interactive Theorem Prover (ITP)	Systems that process logical formulas and apply inference rule on the formulas. Command script is created and executed to produce proof. Human intervention is needed when proving the formulas. High expressive power and flexibility in the type of logic to be used. HOL and FOL is the most common logic to be used and the chosen logic will determine the expressiveness of the theorem prover. ITP can prove complex proof in a large formula. The downside of ITP is proving a formula needs intervention from knowledgeable human in the domain area, logic and language of the prover. Well known ITP are NuPrl, PVS, Isabelle and HOL system
Automated Theorem	Programs that validate a formula without any intervention from

Prover (ATP)	user. Normally use FOL because there are a lot of effective inference rules and proof procedures. ATP can proof small but complex formulas but the complexity of proof will drop as the formulas gets larger. OTTER, SPASS, Gandalf and SETHEO are examples of ATP.

2.3.1 PVS

PVS is a combination of formal specification language and tools to conduct theorem proving. PVS specification language is a strongly typed HOL with a type system that includes built-in type (integers, real and Booleans), user define unintepreted types, enumerations, sets, tuple, records, functions, inductively defined abstract type (binary trees and list) and co-inductively defined abstract type (streams) [25,26]. Having a strong typed language allow specification written in PVS to have no type errors and a consistent higher-order logic. To support a strongly typed language, a type checker system is provided but it is not algorithmically decidable [25]. Users can also create a new type by sub-typing base types using predicates.

The main building block of PVS specifications are parameterised theories. Theories can contain axioms, assumptions, theorems, obligations, types, subtypes, variables, constants, functions, macros and etc. Explanation on theories and its components will be discussed in Chapter 4. Theories as the main construct in PVS specification imply that PVS specification language can be either model-based or property-based. Property-based specification languages describe a system by specifying its intended properties without creating model of the system, while model-based specification language creates a model of the system [24].

As mention earlier, PVS also include a proof checker. Theories are proved in a goal-directed manner and proofs are represented as a sequent [25]. To improve the performance/reliability of the proof checker, PVS uses powerful primitive inference rules and strategies [25]. Strategies are a group of frequently used proofs [25].

The next two subsections will discuss about two theorem proving tools; Isabelle and HOL-S; and two formal specification languages; Z and B.

2.3.2 Other Tools for Theorem Proving

HOL-S and Isabelle are two theorem provers that use HOL. Isabelle as a generic theorem prover also has versions that uses other type of logic such as Isabelle/FOL (Isabelle with FOL) and Isabelle/ZF (Isabelle with Zermelo-Fraenkel set theory) [29]. One similarity between PVS, Isabelle and HOL-S is they support a notion of theory where inside a theory types can be define, axioms and functions can be created, and variables or constants can be declared.

2.3.2.1 HOL-S

HOL-S is greatly influence by LCF (Logic for Computable Function) system and a lot of features in HOL-S are taken from LCF. New feature of HOL-S but not in LCF is separation of consistency-preserving definition principle from arbitrary axioms [28]. HOL-S is supported by powerful rewriting subsystem for forward proof technique but proof can also be generated in goal-directed manner using tactics [28]. In HOL-S tactics is a function that split a goal into sub-goals and records the reason why proving sub-goals will prove the goal [28].

There are four types of terms in HOL-S which are variables, constants (variables that cannot be bound by quantifiers and have a fixed type), function application (relationship between domain and range) and lambda-abstraction (denote a function/operation) [28]. HOL-S also allows theories to be combined or extended. Theory extension is a common way to create a new theory in HOL-S.

2.3.2.2 Isabelle

Isabelle/HOL (Isabelle) is the Isabelle version for HOL. Types that are supported by Isabelle are base types (Boolean, truth values and natural number), type constructor (list and set), function types (total function) and typed variables [21]. As in functional programming, terms in Isabelle are formed by applying functions to arguments. Formulae are terms of type Boolean, such as the constant True and False, logical connectors, quantifiers and equality [21]. Isabelle also has three types of variables, free, bound and schematic. Schematic are free variables that can be instantiated by another term during process [21].

2.3.3 Specification Languages

Section 2.2 discussed various work in verifying models by representing models into specification languages and using tools to verify the representation. The specification used in some of the work are written in either B or Z. Formal specification languages are used to specify what the system should do and avoid specifying how [23]. To be more precise, formal specification languages are used to specify the requirements but not the implementations. This is true for most specification languages but B and Z allow the use of the language to specify the implementations as well.

2.3.3.1 B-Method

B-Method is a method in developing software that encourages the use of B specification language not only in writing formal specification but throughout the development process. Use of formal mathematical notations throughout the software life cycle is achieve by stepwise refinement where formal specification is refine by including formal specification of design and implementation. With stepwise refinement, formal representation of design and implementation can be verified for correctness and conformance to specification. Stepwise refinement will also allow non-determinism rules introduce in the specification to be resolve when refining [22].

B is a model-based specification language where specifications are written by creating model of the system. Model of the system can be specified using Abstract Machine Notation (AMN) and it is the main building block for software specification [22]. With AMN, a large specification can be created by combining smaller specifications represented by machines.

A machine is similar to objects in object oriented programming and it has a name, local states (variables), invariant (condition that must be true at all times), initialisation state and operations (behaviour of the machine) [22]. Operations also act as an interface to change the state of a machine and other machines cannot change the state of other machines directly. However, a machine can have several relationships with other machines such as include (machine A is part of machine B), extend (machine A is part of machine B and all operation in machine A is promoted), uses and sees [22].

B seems to be a suitable specification language to be used with MDA and transformations. MDA introduce the idea of transformation from PIM to PSM this is similar to the term refinement in B-Method. This means the concept of stepwise refinement in B-Method is

similar in objectives to what is being proposed in MDA where each transformation from PIM-to-PSM or PSM-to-PSM is taken as a refinement from specification to design and finally to implementation.

2.3.3.2 Z

Similar to B, Z is also a model-based specification language base on FOL and set theory [23]. Z stresses on operation abstraction process using high-level mathematical structures such as arbitrary set, relation, function, sequence, bag and tree [23]. The operation abstraction process is the process of defining an operation by specifying what are the inputs, outputs, pre-conditions and post-conditions of the operations. Pre-conditions are rules that need to be true in order for the operation to be executed while post-conditions are the rules that must be true after the operation is executed.

The schema is the basic construct of a system specification written in Z. Spivey [24] explained a schema consist of a collection of named object with a relationship specified by some axioms. Similar to machines in B, schemas allow specification to be broken up into sub-specification and each sub-specification is represented by a schema. Complex specifications can be created by connecting a number of schemas using logical connectors (AND or OR). But, unlike machines, schemas represent a state in the system only using variables, and do not encapsulate any behaviour (operation). Operations in Z, or the result of executing an operation, are specified in a separate schema.

From the description in the previous paragraph, schemas in Z can be used in two ways. The first use of schemas is to specify a state of the system or an object in the system [23, 24]. For example, a schema can be created to represent a phone book that have two entries a name and a phone number. A schema can also specify rules of a valid phone book for example a name can only be associated to one phone number.

The second use of schema is to represent the change of state of the system cause by the execution of an operation [23, 24]. For example, change of state when adding a new entry (a pair of name and telephone number) in a telephone book. When using schema in this way, a schema can be parameterised and give output.

There is similarity in the concept of schema and the concept of structure in C programming language [30]. This similarity is obvious when schema is use to represent a complex type that have a state. Structure in C is also similar where when creating a new structure, the programmers actually create a new complex type by encapsulating variables of basic types and arrays. Although conceptually, structures unlike objects do not have a state but changes of values in variables inside a structure can be taken as a change of state, the same as object in object oriented, schema in Z and machine in B. This observation seems to be correct in certain situations such as the phone book examples given above. Situations may exist where schema used to represent complex type but does not resemble a structure.

Refinements are also supported by Z and there are two types of refinements, data refinements and operation refinements [30, 31 and 32]. Data refinements refine data structures in Z such as sets and sequence to data types or structure that is supported by programming languages. When refining data structures, the refine data structure must have the same properties of the data structure is Z [31]. Operation refinement in Z refines operations (usually represent by a schema) based on the new refined data structure. Preconditions schemas for the operation can also be defined as part of the process of refining an operation. Properties of the refined operation that needs to be check are applicability (refined operation can be applied safely whenever it is

safe to apply operation in the specification) and correctness (refined operation gave the same output and produce the same state change as the operation in specification) [31].

2.3 Model Transformation Technology

As mentioned earlier, model transformation technology that will be discussed here is chosen from Eclipse GMT and M2M project. Technologies inside these projects are QVT and ATL for M2M project and Epsilon, VIATRA2, MOFScript, and UMLX for GMT project. MOFScript is a Domain Specific Language (DSL) created to transform models to text specifically from models to implementation code or documents. UMLX project provide complementary tools and concrete graphical syntax to model transformation [2]. Currently UMLX is using ATL in transforming UMLX models to QVT [2].

Besides technology and languages in GMT and M2M project, other languages exists such as GreAT, ATOM, BOTL, Jamda, AndroMDA, OptimalJ, ArcStyler, XDE and etc. Czarnecki and Hilson [3] provide a classification on model transformation languages mentioned above. The classification is done based on design features of transformation (see Appendix A). Major categories of transformation languages are also proposed in [3] which are Model-To-Code language and Model-To-Model language. Model-To-Code is further divided into Visitor-Based approach and Template-based approach. Model-To-Model can use Direct-Manipulation approach, Relational approach, Graph-Transformation-Based approach, Structure-Driven approach or Hybrid approach.

Further discussion will be on QVT, ATL, Epsilon and VIATRA2. UMLX is excluded because UMLX is a supporting tool for transformation languages.

2.3.1 QVT

QVT is proposed by the OMG as part of the Model-Driven Architecture (MDA) initiative. QVT is a query, view and transformation language created by combining multiple proposals from organisations and groups. The result is a language and standards for tools specified in [4].

Figure 2.3.1 is the architecture of QVT transformation language. QVT is a hybrid of declarative and imperative constructs [5] where the declarative construct is divided into 2 layers. QVT declarative construct, Relations and Core, are somehow the same but in different level of abstraction. Between the two, Relations is a more complex language that supports object pattern matching and object template creation while Core language is simpler and only support pattern matching over a flat set of variables [4]. Transformation can be done from Relations language to Core language and in this case the Core language act as a reference for the semantics of Relations language [4, 5]. Traceability in transformation process is done automatically in the Relations language but need to be managed manually in the Core language just as any other MOF objects.

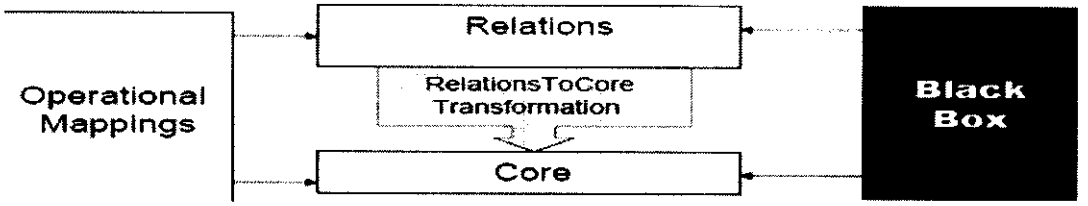


Figure 2.3.2 : QVT language architecture taken from [4]

QVT has two imperative languages, Operational Mappings and Black Box. Operational Mappings is the standard imperative language while Black Box is a plug-in mechanism for executing complex algorithm from external code programmed in any programming languages [5]. Operational Mappings can be used in two approach, (1) operational transformation and (2) hybrid transformation [4]. Operational transformation implement a transformation using only imperative constructs while hybrid transformation used imperative construct to initiate object patterns specified in relations. Black Box must be use with care because transformation using a black box is not control by the transformation engine. The transformation process is partially or fully executed by the black box program making traceability by transformation engine difficult [5]. In order to be able to trace the transformation process is by explicitly implement a Relations [4].

In [4], conformance of tools to the QVT standard can be at one or more of the 12 conformance points. There are two dimensions used in identifying conformance of a tool, Interoperability and Language. Language dimension represent the two declarative construct (Relations and Core) and Operational Mappings language that made up QVT. Interoperability dimension specify four functionalities [4]:

1. Syntax Executable: Be able to execute syntax of any language construct in Language dimension.
2. XMI Executable: Be able to import and execute QVT transformation in XMI serialisation
3. Syntax Exportable: Be able to export model-to-model transformation in any language construct in the Language dimension
4. XMI Exportable: Be able to export model-to-model transformation into its XMI serialisation

Conformance points are the combinations of elements in Interoperability dimension and Language dimension. The combinations that created the 12 conformance points are best shown in Figure 2.3.2. An implicit requirement that is not shown in Figure 2.3.2 are tools that is Syntax Executable must also be Syntax Exportable for the same language level and tools that is XMI Executable must also be XMI Exportable in the same language level.

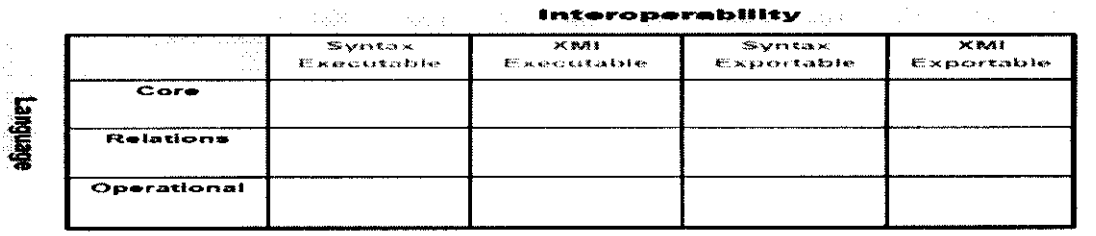


Figure 2.3.3: QVT conformance table

Although QVT is the standard transformation language as proposed by OMG, more and more transformation languages have been created since the publication of [4]. Instead of developing tools that conform to QVT requirements, research groups are creating new model management languages to fulfil the requirements in QVT RFP or [4] with the addition of other functionalities.

The existences of these languages are to overcome the shortcomings of QVT which is said to be immature for unidirectional transformation, and difficult to use [6]. Also, QVT specification only concentrates on model transformation which is just a small part of model management. There are no indications or explanations in [4] of facilities to extend other model management operation such as model comparison, model merging and management of multiple metamodels. Platform such as AMMA and Epsilon is currently working on providing support for these operations.

2.3.2 AMMA/ATL

ATL is a transformation language similar to QVT. It is now part of a model management called AMMA. Other than ATL, AMMA also consist of Atlas Model Weaver (AMW), Atlas MegaModel Management (AM3) and Atlas Technical Projectors (ATP). AMW proposed the idea of specifying relationship model between source model and target model. The relationship model then can be use to create transformation script in any direction (source to target, target to source or bi-directional) [7, 8]. Before AMW is proposed, the process of weaving a source model to a target model is done implicitly by human when they create a transformation script. AMW initiative extracts the task of identifying these relationships and creates metamodels and tools to weave source and target models.

The advantage of using AMW is the creation of transformation scripts can be done automatically in any direction instead of creating manually two transformation script one for each direction [8]. Another advantage of using AMW instead of only transformation is the reuse of transformation patterns in specifying the relationship between model elements [8]. The patterns can be reuse for any source or target models that confirms to any metamodel that is woven using the same relationship. Weave models can also be used as a source to trace the transformation.

The AM3 project is an initiative related more to model management than transformation. Atlas proposed the term megamodel to represent a M1 level model that contains other models. A new meaning for *Zoo* is also introduced, which is a type of megamodel that contains all models that confirms to the same metamodel [9]. ATP is a new project that tries to identify the projections between different technological spaces and incorporate it in AMMA platform. The projections will be used in transformations applying ATL.

ATL transformation language supports two components of QVT which is query and transformation. A query in ATL is done using OCL, in much the same way as in QVT. ATL can only transform models that conform to MOF and that have fully initialised model elements. This means that a modeller can navigate through elements from source model, source metamodel, target metamodel, ATL metamodel and current transformation model [10]. ATL can be considered as a hybrid language that allows the modeller to construct fully declarative rules, fully imperative rules (procedure) or hybrid rules that combine elements of both declarative and imperative rules.

There are two types of transformation rules in ATL, *called* rules and *matched* rules. Called rule are rules that call ATL predefine operation or modeller-define operation while matched rule is used in declarative rule [10]. Declarative rule have source (inPattern) and target (outPattern) pattern. Source patterns used match rule and OCL to assign elements of source metamodel to variables while target patterns used match rule to bind values from variables to elements in target metamodel. Rules can be inherited and modeller can also create abstract rules which have similar concepts as abstract class in object oriented programming [10]. ATL abstract syntax mention above can be represent using textual or graphical concrete syntax.

Traceability information can be added to ATL transformation scripts in order to generate traceability model elements. Traceability model just like any other model have traceability metamodel [11]. Traceability is also supported internally by ATL transformation engine. ATL transformation engine will store runtime information on transformation being done [10]. Atlas used Model-In-Action (MIA) transformation engine developed by Soft-Maint for ATL instead of creating its own engine [12]. Table 2.3.1 shows the complete design features of ATL taken from [3].

Table 2.3.2: Design features of ATL

Design Features	
Transformation Rules	Full declarative, full imperative or hybrid
Rule Application Scoping	Scoping for both source and target model
Source-Target Relationship	Separate source and target model but support in-place transformation
Rule Application Strategy	Deterministic
Rule Scheduling	Implicit and explicit, internal scheduling
Rule Organisation	Support rule inheritance
Tracing	Separate storage location and automatic
Directionality	Unidirectional

2.3.3 Epsilon

Epsilon is a suite of DSL for model management contains in the same platform and base on one common language, Epsilon Object Language (EOL). The objective of Epsilon is to provide platforms for development of model management DSL without the need to re-implement the same basic language features and redeveloped tools [13]. To achieve this, two base layers, Epsilon Model Connectivity (EMC) and EOL, are created. EMC layer will interface with different modelling technology by providing driver specification [13]. Currently EMC driver exist for Eclipse Modelling Framework (EMF) and Meta-Data Repository (MDR).

EOL built upon OCL for model navigation and adds features to overcome the deficiencies of OCL (because OCL is a constraint language, not a model management language) [13, 15]. EOL supports statement sequencing and grouping, access to multiple models, model modification, predefine operation for debugging and error reporting and reusability of user-define operation [15]. The most notable new feature of EOL compared to other transformation languages are the reusability of user-defined operation. User-defined operation can be reused in any DSL in Epsilon platform and can be exported and imported into external model management programs.

Figure 2.3.3 is the latest architecture of Epsilon and currently Epsilon has five DSL, Epsilon Transformation Language (ETL), Epsilon Comparison Language (ECL), Epsilon Validation Language (EVL), Epsilon Wizard Language (EWL) and Epsilon Merging Language (EML) [14]. There is another language that is not in the figure, the Epsilon Generation Language (EGL). EGL is a template based model-to-text transformation language that uses EOL and is similar to MOFScript [15].

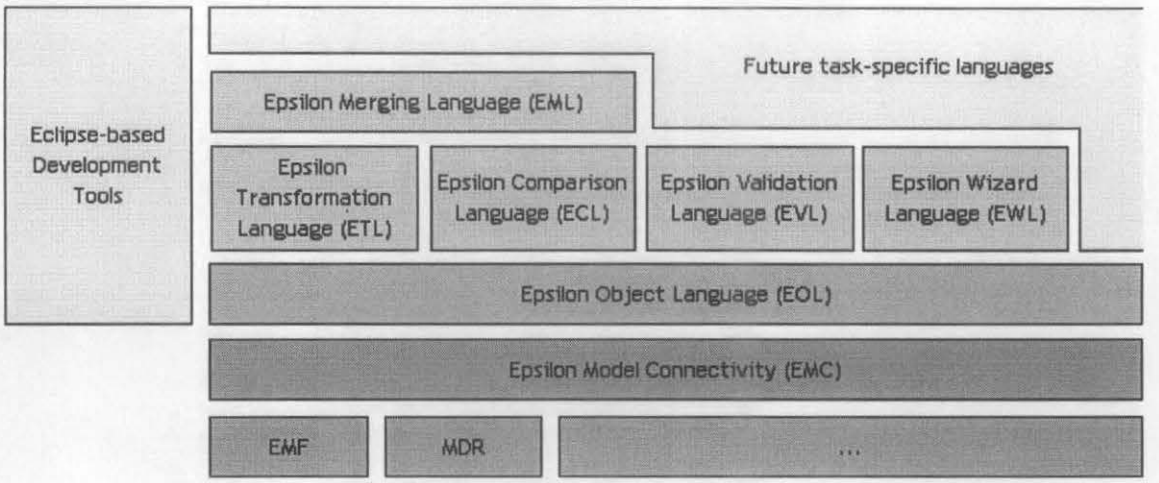


Figure 2.3.4: Epsilon Architecture

Below are brief descriptions of the model management DSLs mentioned above:

- ETL [13]: a rule based transformation language that transforms elements in source model to elements in target model.
- ECL [16]: a language that compares source and target model elements. The need for ECL arose when researchers found that comparison is an essential process for other model management processes such as model merging and model transformation testing. Model comparison is based on the fundamental notion of a *match* defined between model elements. There are four categories of matched elements, (1a) exist elements in target model that match and conform to elements in source model, (1b) exist elements in target model that match but not conform to elements in source model, (2a) no elements in target model exist in target model but exist in domain of comparison operation and (2b) no elements exist in both domain of comparison operation and target model.
- EVL [13]: a language for constraint and consistency checking amongst models; in particular, it supports inter- and intra-model consistency checking, and also improves error/warning reporting features in other DSLs in Epsilon.
- EWL [14]: a transformation language created specifically for transformation in the small. The term transformation in the small means transformations that updates selected group of model elements. The selected element is explicitly chosen by the user. The opposite of transformation in the small is obviously transformation in the large, which is generally supported by ETL. Transformation in the large executes batch transformation on model elements specified in transformation programs.
- EML [17]: Merging language that is built on top of ETL and ECL. A merging process requires model comparison, conformance on feasibility of merging, merging all models and lastly removing inconsistencies in the result model. Comparison and conformance of merging is done using ECL *compare* and *conform* rules. Merging is done using merge and transform (from ETL) rules depending on type of elements. If there exist matching model elements in source models, elements will be merge but if there is no matching

elements, the elements will be transform. EML also allows the use of strategies which is a merging rule define from metamodels. Strategies can be created for comparison and merging process.

EML also support traceability features by creating a second result model (besides the merging result model) that have annotated traceability stereotype. These stereotypes will specify from which source model the model element came from and if it is located in all source models *<<common>>* stereotype will be used [18]. The annotated result model is a solution that combines embedded traceability (traceability link embedded in result model) and external traceability (traceability model that confirms to traceability metamodel) [18]. Embedded traceability has the problem of flooding the result model with stereotypes while external traceability requires human to inspect two models to trace a transformation. Loosely coupled traceability used in ATL [11] is an example for external traceability.

ETL can be considered as a new transformation language if compared with ATL and VIATRA. Categorisation work in [3] does not include ETL because at that time ETL was still in its development. The categorisation is also suitable for EML, EWL and EGL. Table 2.3.2 will describe the design features of ETL and features of EML, EWL and EGL where relevant.

Table 2.3.3: Design features of ETL

Design Features		**ETL features**
Transformation Rules	LHS/RHS	Syntactically typed variables. Patterns are in string form, represented in abstract and textual concrete syntax. Logics are executable and have imperative and declarative constructs.
	LHS/RHS Syntactic Separation	LHS and RHS is syntactically separated using ':=' symbol
	Bidirectional	Only support single direction rule
	Parameterization	Rules can be parameterise
	Intermediate Structures	There is no immediate structure.
Rule Application Scoping		Scope of transformation can be set to certain model element. ETL does this by specifying in transformation script. EWL allow user to explicitly choose model elements to transform
Relationship between Source & Target		ETL and EML will create new target model but EWL can have in-place transformation
Rule-Application Strategy		ETL, EML rule application strategy is deterministic. EWL use interactive strategy
Rule Scheduling	Form	User has no control of scheduling algorithm (implicit)
	Rule Selection	ETL and EML can have *guards* as an explicit condition. EWL allow interactive selection of rule
	Rule Iteration	Rules can be iterate using recursion, looping and fixpoint iteration
	Phasing	ETL does not have any phase when executing transformation but EML has.

Rule Organisation	Modularity Mechanism	Support modularity by grouping rules into module
	Reuse Mechanism	Rules can be inherit and can be called from other rules (logical composition)
	Organisational Structure	ETL is source oriented but Epsilon allow creation of DSL with independent structure
Tracing	Storage Location	Traceability information is stored in a separate model
	Control	Traceability links is created automatically for ETL.
Directionality		Transformation can only be done in one direction. Bidirectional transformation can be done using two complementary unidirectional transformation scripts.

Based on Table 2.3.2, ETL is a unidirectional and hybrid model-to-model transformation language. It is also very useful to specify to what extent Epsilon languages fulfils the main requirements of QVT RFP. This will show the suitability of using Epsilon as a QVT tool. Some of requirements of QVT RFP are fulfil by design features specified in Table 2.3.2.

Table 2.3.4: Degree of fulfilment of Epsilon to QVT RFP

QVT RFP Requirements	**Epsilon**
Proposal shall define a query language	Epsilon develop EOL as its query language
Proposals shall define a language for transformation definitions	Currently Epsilon have ETL, EWL and EML
The abstract syntax of transformation will be define in MOF	Abstract syntax of Transformation is done using Context Free Grammar (CFG)
Transformation language can create target model from source model automatically	Transformation using ETL, EWL and EML is done automatically by providing tools
Transformation language enable the creation of a view of a metamodel	Not supported
Incremental changes in source model may be transformed into changes in target model immediately	Not supported
Proposals shall operate on model instance defined using MOF	Epsilon can transform models that confirms to MOF
Proposal may support bidirectional transformation	Bidirectional is only supported by having two unidirectional transformation
Transformation may support traceability of transformation execution	Traceability information is recorded automatically in a separate model
Proposals may support mechanism for reusing and extending generic transformation definition	Transformation rules can be extend, can be called from other rules or by creating strategies (only for EML)
Proposals may support transactional transformation definition	Currently Epsilon does not provide database like transactional transformation
Proposals may support the definition of transformation where the source and target model is the same	EWL support in-place transformation but ETL and EML will create a new model
Provide support for black-box implementation.	EOL can callout to java codes using wrappers

Another approach to show conformance of ETL to QVT is by using a QVT conformance table (Figure 2.3.2). Although the conformance table from [4] is intended to be used with tools, it can be used for transformation languages with minor modification. In [4] there is an explicit explanation that conformance for language dimension means only for the concrete syntax in [4] but for the table to be used for Epsilon (or other transformation languages), this constraint will be disregarded. Removing the constraint allows ETL to be categorised as QVT-Operational-Syntax Executable and QVT-Operational-XMI Executable.

2.3.4 VIATRA2

The VIATRA2 model management framework is aim for a more specific market compared to AMMA and Epsilon. VIATRA2 is targeted for safety critical, embedded, and robust e-business application, while AMMA and Epsilon is developed without any constraint on application.

All VIATRA2 targeted applications require precise system development and VIATRA2 used formal methods to support precise model-based system development [19]. Because of this, all transformation in VIATRA2 must be specified in a precise way [20].

VIATRA2 proposed the Visual and Precise Metamodeling (VPM) instead of using MOF metamodeling standard. This is because MOF is claimed to be unsuitable for multi-level metamodeling, which is important for working with multiple technological spaces [19].

VIATRA2 transformation language can support both model-to-text and model-to-model transformation. Model-to-text transformation is done using a template-based approach while model-to-model transformation is done using graph-transformation-based approach [3, 19]. Viatra Textual Command Language (VTCL) is created for model-to-model transformation and Viatra Textual Template Language (VTTL) for model-to-text transformation. VTCL combines graph transformation (GT) and abstract state machine (ASM) technique where GT is used to define rule for model manipulation and ASM is used to define control structure [19].

VIATRA2 currently conforms to all mandatory and optional requirements specified in QVT RFP except for (1) support for declarative and bidirectional transformation and (2) support for view generation [19].

2.4 Hypotheses

From the study of specification languages and theorem proving tools, two concerns arise:

1) Is there going to be any difficulties when transforming OCL, a specification language based on FOL, to PVS which uses HOL?
2) Can OCL be translated into a property-based specification language, or must it be translated into model-based specification language?

From these concerns, two hypotheses are drawn and are shown in Listing 2.4.1.

Listing 2.4.1: Project Hypotheses

H1: There is no problem transforming from OCL to PVS cause by the nature of OCL that use FOL and PVS that use HOL

H2: OCL can be represented in property-based form languages

Next chapter: Project methodology, requirements and planning

3. REQUIREMENT ENGINEERING

Chapter 3 will discuss the first project activities which are requirement identification, selecting a project methodology and planning the design implementation. Two development concerns will also be discuss in the final section.

3.1 Requirements

This project has two types of requirements, functional and non-functional. Functional requirements are the capabilities of the transformation or a specific list of what the transformation supposed to do and the model that comes out of the transformation. Non-functional requirements are the properties of interest that the transformation in this project need to have. Table 3.1.1 is the list of functional requirements and Table 3.1.2 is the list for non-functional requirements. Both functional and non-functional requirements will be given priorities. These priorities will be used in planning the implementation.

Requirement priorities are determined by its importance in transforming OCL to PVS. Priority for requirements in transforming OCL types are determined by the probability of it being used. Priority for requirements in transforming OCL expressions is based on its type and the probability of it being used. For example, OCL Property expression is more likely to be used than OCL Let expression and requirements for expression in OCL standard library has high priority because some of the expressions are related to types that have high priority.

Table 3.1.5: Functional Requirements

Req.#	Functional Requirements	Priority
FR01	***Transformation shall transform OCL types to its equivalent in PVS***	
FR01.1	Transformation shall transform OCL Integer type to its equivalent in PVS	High
FR01.2	Transformation shall transform OCL Real type to its equivalent in PVS	High
FR01.3	Transformation shall transform OCL Boolean type to its equivalent in PVS	High
FR01.4	Transformation shall transform OCL String type to its equivalent in PVS	High
FR01.5	Transformation shall transform OCL Ordered Set type to its equivalent in PVS	High

FR01.6	Transformation shall transform OCL Set type to its equivalent in PVS	High
FR01.7	Transformation shall transform OCL Sequence type to its equivalent in PVS	High
FR01.8	Transformation shall transform OCL Bag type to its equivalent in PVS	High
FR01.9	Transformation shall transform OCL Tuple type to its equivalent in PVS	High
FR01.10	Transformation shall transform OCL Void type to its equivalent in PVS	Medium
FR01.11	Transformation shall transform OCL Invalid type to its equivalent in PVS	Medium
FR01.12	Transformation shall transform OCL Any type to its equivalent in PVS	Medium
FR01.13	Transformation shall transform OCL Type type to its equivalent in PVS	Medium
FR01.14	Transformation shall transform OCL Message type to its equivalent in PVS	Low
FR01.15	Transformation shall transform OCL Element type to its equivalent in PVS	Low
FR02	***Transformation shall transform OCL expression to PVS***	
FR02.1	Transformation shall transform OCL State Expression to PVS	Low
FR02.2	Transformation shall transform OCL Message Expression to PVS	Low
FR02.3	Transformation shall transform OCL Type Expression to PVS	High
FR02.4	Transformation shall transform OCL Variable Expression to PVS	High
FR02.5	Transformation shall transform OCL If Expression to PVS	Medium
FR02.6	Transformation shall transform OCL Let Expression to PVS	Medium
FR02.7	Transformation shall transform OCL Literal Expression to PVS	High
FR02.8	Transformation shall transform expression in OCL standard library	High
FR02.9	Transformation shall transform OCL Iterator Expression to PVS	Medium
FR02.10	Transformation shall transform OCL Iterate Expression to PVS	Medium
FR02.11	Transformation shall transform OCL Property Expression to PVS	High
FR02.12	Transformation shall transform OCL OperationCall Expression to PVS	High
FR02.13	Transformation shall transform OCL Navigation Expression to PVS	High
FR03	Transformation result shall be able to be prove by PVS theorem prover	High

A full list of OCL expressions and descriptions are given in Appendix D. The semantics and well-formedness rule of OCL expressions and types can be found in [57]. All sub-requirements of FR01 and FR02 are based on the OCL metamodel that is taken from [57]. There are other metamodels proposed in [61] and [62] but metamodel from [57] is chosen because it comes from the OMG and this will increase the applicability of the projects findings.

Table 3.1.6: Non-Functional Requirements

Req. #	Non-Functional Requirement	Priority
NFR01	Correctness – Semantics of OCL will be preserved after transformation to PVS	High
NFR02	Deterministic –The same OCL will always generate the same PVS when running the same transformation program	High
NFR03	Simplicity – Transformation result shall be simple in order to ideally allow PVS theorem prover to carry out proof without human intervention	Medium
NFR04	Traceability – Result of transformation shall be able to be trace back to source model	Medium

For this project, the transformation will be done using Epsilon (ETL to be specific) in Eclipse. Epsilon is chosen over other transformation languages because 1) experts in Epsilon are available for assistance and guidance in the university and 2) Epsilon has never before this been used in analysing model properties through transformation, thus increasing the research value of this project.

A decision has also been taken on what kind of transformation is suitable for our purpose. We have three choices, which are model-to-model transformation, model-to-text transformation and text-to-text transformation. The first option is chosen because some of the information needed can only be obtained from UML models that the OCL being transform constraints. The most obvious one is class attributes.

Class attributes have a type and in OCL this information is not explicit. To know the type, one must look at the UML model or make a judgment based on the value used with the attributes in OCL expression. To transform OCL to PVS all elements of OCL that is being transformed must have a type. Although types for attributes can also be identified from OCL init or derive statement where the type of the attribute is specified in the context, only type for attributes that have init or derive statement can be identified. Other attributes and its type will still be unknown if UML model were not referred.

From the justification above, an assumption can be made that transformation of OCL to PVS can only be done without the transformation of the UML model if and only if the OCL is complete. The definition of complete here is the OCL must specify invariants of all classes, initial or derive value for all attributes, and operation's body, pre or post conditions for all operations. But complete OCL is seldom created, thus it is safer to include the transformation of UML model with the transformation of OCL. Table 3.1.3 adds new functional requirements to the list of requirements in Table 3.1.1 based on the explanation above.

Table 3.1.7: Requirements from UML model

Req.#	Requirements	Priority
FR04	UML class concepts will be transform into PVS	High
FR04.1	UML class will be transform into PVS	High
FR04.2	UML class attribute will be transform into PVS	High
FR04.3	UML class operation will be transform into PVS	High
FR04.4	UML association end will be transform into PVS	High
FR05	OCL constraints will be transform into PVS	High
FR05.1	OCL invariant will be transform into PVS	High
FR05.2	OCL initial will be transform into PVS	High
FR05.3	OCL derive will be transform into PVS	High
FR05.4	OCL pre-condition will be transform into PVS	High
FR05.5	OCL post-condition will be transform into PVS	High

The reason given above eliminates text-to-text option. Model-to-text transformation is also not the best option because PVS specification language cannot completely represent all OCL expressions. In order to have model-to-text transformation, all OCL expressions should exist in PVS similar expressions. This requires the creation/extension of PVS libraries in addition to creating the transformation program. We chose model-to-model transformation so that the project doesn't need to extend PVS libraries and still have the possibility of maintaining the completeness of the transformation. Besides, having a PVS model as the result of the

transformation means the result will be available for other model management activities such as comparison, merging and validation.

Because of the decision to use model-to-model transformation and to transform both UML models and its OCL constraint, the process that we explain in Chapter 1 is change. Now the process will accept not only imported OCL constraints but also UML models. The transformation result will be a PVS model that will then be serialised to PVS specification. Figure 3.1.1 show the new process.

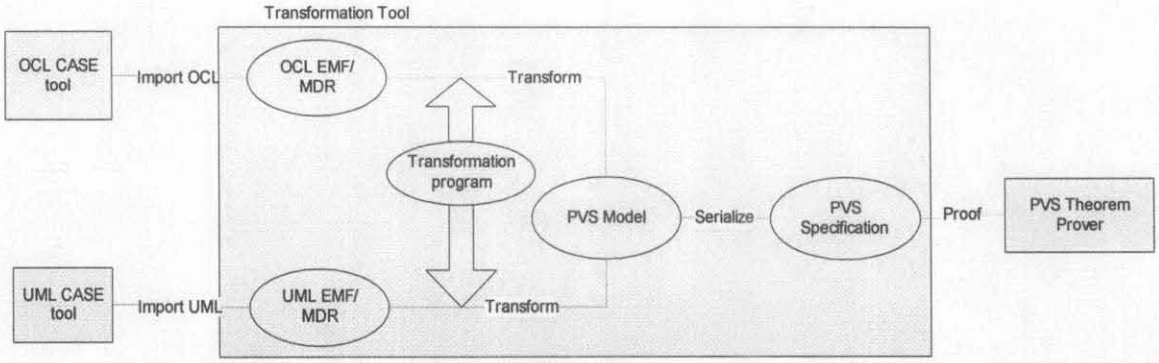


Figure 3.1.5: Process in proving properties of OCL constraints

3.2 Methodology

A Spiral model (with modification) is chosen for this project. The modification made to the model is the activities that will be done in iteration. In each cycle, there are three main phases which are planning, implementation and testing. During the planning phase, requirements to be implemented in that iteration are chose/identified. Test cases will also be prepared for each requirement. The second phase of the development is the implementation where the transformation is designed and implemented. After implementing the transformation, it will be tested using test cases created in the planning phase. Figure 3.2.1 is the diagrammatic form of our methodology.

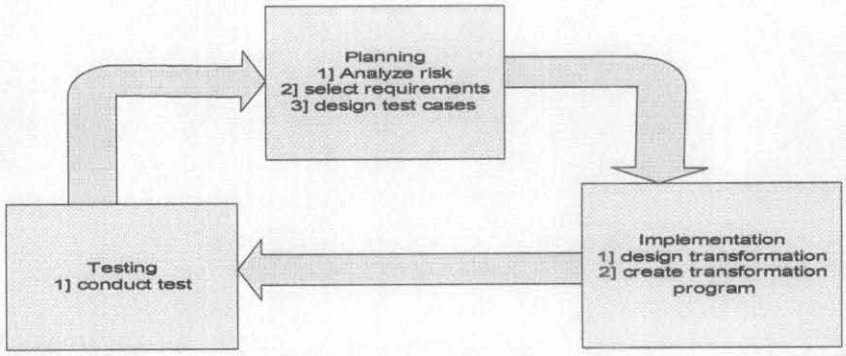


Figure 3.2.6: Methodology

Compare to the spiral model proposed by Boehm [63], this project spiral model strips down some of the features of the original model. Features that are removed are prototypes, concept of operation, requirement validation and design verification. Instead of multiple levels of design and test, the strip down spiral will conduct only one level of design and test for a cycle.

This project has only one risk, which is the inability to complete all the requirements because of time constraint. During the planning phase of each cycle, priority of requirements and the order of implementation may change as a reflex to risk analysis.

There are other methodologies available for developing software. Since this project is about model transformation and the product is a transformation program written in a transformation language, existing methodology does not quite fit into the project. Table 3.2.1 summarise the investigation that has been done on identifying methodologies suitable for this project.

Table 3.2.8: Methodologies evaluation

Methodology	Reason for unsuitability
Waterfall	This project is experimental in nature and more suitable with methodology that have iterations
Incremental	This project have evolutionary characteristic where priorities of requirements may change over time
RAD	RAD separate the project into teams but this project must be done by one person
Prototyping	There is no prototype for this project
Spiral	Seems to be the best candidate for its evolutionary nature but the activities (as explained in [1]) need to be change
COTS	There is no existing libraries or components that can be reuse for this project
RUP	More suitable for large commercial project [58] compared to experimental project such as this.
Agile (XP, SCRUM, ASD, DSDM)	Agile encourage collaboration and communication among teams and many techniques in Agile methodologies (paired programming in XP and scrum meeting) is catered towards this. This special technique is not needed in a single-person project

Since none of the existing methodologies fit directly with this project, the proposed methodology took some of the concepts from methodologies specified in Table 3.4. The spiral structure is chosen because it allows the project to change its priorities in selecting which part of OCL need to be transformed. Spiral structure is evolutionary in nature also means that it is incremental. Incremental characteristic allow flexibility in the completeness of the final product. The project can stop implementing the transformation either because the project already meet its objectives or because of time constraints.

Another concept that is taken from existing methodology is Test-Driven Development (TDD). TDD encourages the creation of test cases before development and the implementation process is done based on the test cases. This will lead to the development of products/components that only passes the test, avoiding gold-plating features that are not needed, and increases the confidence of the developer in the product.

Next section will discuss the project's implementation plan. The plan will divide the requirements into cycles.

3.3 Project Implementation Plan

After identifying and classifying requirements, the rest of the project will be divided into several cycles where in each cycle; requirements will be selected for implementation in that cycle, design of the transformation and implementation of the transformation will be created and the transformation will be tested. This subsection will divide the functional requirements into cycles as shown in Table 3.3.1. Non-functional requirements will be fulfilled in all cycles.

However, the first cycle is unique where it will not follow the project development phase. In the first cycle, a metamodel for PVS specification language will be created, thus there is no need for planning, design and testing.

Table 3.3.9: Project implementation plan

Requirement	Cycle	Requirement	Cycle
FR01.1	2	FR02.2	5
FR01.2	2	FR02.3	2
FR01.3	2	FR02.4	2
FR01.4	2	FR02.5	4
FR01.5	3	FR02.6	4
FR01.6	3	FR02.7	2, 3, 4
FR01.7	3	FR02.8	2, 3, 4
FR01.8	3	FR02.9	3
FR01.9	3	FR02.10	3
FR01.10	4	FR02.11	2
FR01.11	4	FR02.12	2
FR01.12	4	FR02.13	2
FR01.13	4	FR03	2-5
FR01.14	5	FR04	2
FR01.15	5	FR05	2
FR02.1	5		

Requirement FR02.7 and FR02.8 will be implemented in 3 cycles because OCL Literal expression and OCL standard library contains expression that support different types; primitive (cycle 2), collections (cycle 3) and general (cycle 4) types. Requirement FR03 will be implemented in all cycle except cycle 1 in order to say that all transformation result must be able to be proved by PVS theorem prover. Requirements in cycle 5 will only be implemented if there is enough time because types (FR01.14 and FR01.15) and expression (FR02.1 and FR02.2) are related to UML behavioural diagrams while other requirements are more frequently use in UML structural diagram.

3.4 Development Concerns

Model transformation is a new approach to software development and the artefact in model transformation is the transformation program. In order for software development to remain as an engineering discipline, engineering practises must also be applied to developing model transformation. Thus, the development of model transformations also has specification, design and testing phases. This subsection focuses on the issues of testing and design of transformation.

3.4.1 Testing

Testing transformations has started to gain attention from the MDA research community. Current research focuses on applying and customizing existing testing technique for software to test model transformation programs. Functional testing which is a kind of dynamic testing is currently the technique being customized and adapted for model transformation [58, 59].

Both [58] and [59] proposed a technique on how to generate test data for functional testing. Test data generation is one of the major concerns in generation of test cases for functional testing, other than identifying the objective of the test cases and how to measure the result of the test [58]. One of the methods proposed is, generating test cases from model transformation specification (MTS) written in formal language [58]. MTS are a mapping of elements from the source model to elements of the target model [58]. [58] proposed a formal language called MTSpecL for writing MTS.

Another method in generating test data is by conducting partition analysis to source and target metamodel. From a metamodel, test data can be generated from association multiplicities, class attributes and generalization [59]. Comparing the work in [58] and [59], [59] explicitly mention the use of category partitioning method and cover the issue of test data coverage and representative values. [58] pays little attention to these two issues and did not constrain their partition analysis technique to any method. It concentrates more on the use of MTS in generating test cases.

Other contributions of [59] are using OCL to constrain the scope of metamodel being analyse, conducting static analysis on model transformation program for identifying representative values, and automatic generation of models of test data. Using static analysis, representative values can be identify by checking the transformation program for values that are being assigned or compared to attributes. Drawbacks of static analysis are human need to understand the transformation language and result of the analysis depends greatly on the transformation language being used [59]. In the automation of generating test data models, [59] proposed the use of systematic approach or bacteriologic approach.

A framework for testing transformation has been proposed in [60]. The framework consists of Test Cases Generator, Testing Engine and Test Analyzer. Test case generator accepts input of source and target metamodel, transformation specification and test specification to generate executable test cases. Generated test cases, source model, expected model and transformation specification is given to test engine where executor component inside the test engine will generate the result model and comparator component will compare the expected model with the result model. From the result, test analyzer will highlight the errors in the transformation program.

The comparator component in the testing engine compares result model with expected model using graph matching algorithms [60]. Another method of comparing models has been proposed in [16] where comparison is being done using the Epsilon Comparison Language (ECL). Using ECL requires human to write a comparison program in ECL, and given the source and target model the program will compute the compatibility of source and target model. Comparison result from ECL can be group into one of the four categorisations shown in Table 3.4.1 [16].

Table 3.4.10: Interpretation of ECL results

Result Categorisation	Interpretation
Matching and conforming	Transformation has correctly mapped from source to target model
Matching but not conforming	Transformation may be incorrect
Not matching but belonging to domain of comparison	Transformation is incomplete
Not matching and not belonging to domain of comparison	Comparison in incomplete

From the studies on transformation testing, research on transformation testing is still in its infancy and there are no tools that can be used to automatically test a transformation. The framework proposed in [60] is targeted for C-SAW transformation engine that uses Embedded Constraint Language as its transformation language. For this project, transformations will be tested using a comparison program written in ECL because it is easier to use the same tool (Epsilon) for both transformation (written in ETL) and comparison.

3.4.2 Design

Other than transformation testing, transformation design is a more pressing concern because as of now there is no published research for this topic. For the purpose of this project, a design notation will be proposed. The notation is neither complete nor universal in that it can cover the need of all types of transformation. The notation is only build to cater the need for this project.

Design language for transformation is needed because of the same reason why developers need design/modelling notation to design software. It is obvious that transformation program is not abstract for people to understand and rely heavily on the language construct of the transformation language, the same reason why programmers communicate through diagrams and not through source codes.

The design notation is influence by Atlas Model Weaver (AMW) [7, 8] that was previously mentioned in Chapter 2. AMW map the relationship between source and target model using links. Links is associated with strategies that can be reuse to show mapping of other elements. AMW has a metamodel but there is no example of AMW model in [7] and [8].

The proposed design notation also shows the source model, target model and the mapping between them. The mapping can have information that show how the source element is being transformed to the target element. Metamodel for the design notation is shown in Figure 3.4.1.

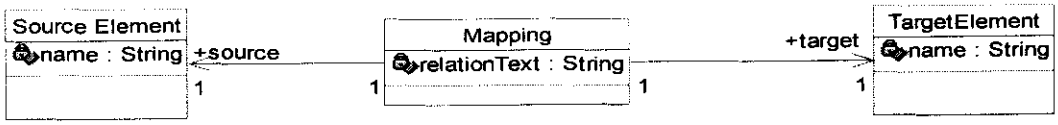


Figure 3.4.7: Design metamodel

To better show the proposed design notation, a small example will be given in this section. The example is a transformation of class attributes to PVS (requirement FR04). An

attribute has a name, a type and an assumption is made that it does not have any initial value. The attribute will be transform into a variable in PVS of the same type. Figure 3.4.2 show the design of UMLAttribute-to-PVS transformation.

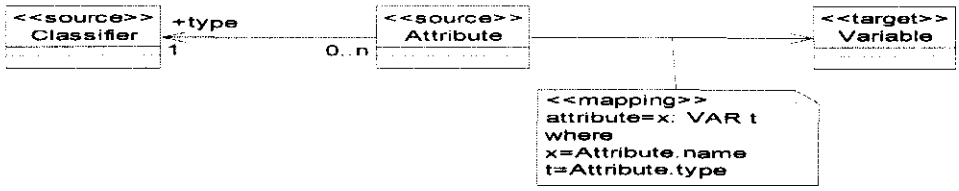


Figure 3.4.8: Design for UMLAttribute-to-PVS

Next Chapter: Designing & Implementing Transformation

4. DESIGN & IMPLEMENTATION

This chapter will focus on the transformation of OCL to PVS. The implementation will be broken into several cycles (section 3.3) and each section in this chapter will explain the work that had been done in a single cycle. Objective of each cycle are:

- **First Cycle**: to create a PVS metamodel
- **Second Cycle**: to design and transform classes, properties, operations, constraints, OCL basic types (and operations), and OCL expressions (property call, navigation call and operation call)
- **Third Cycle**: to design and transform tuple type, collection types (and operations), iterator expressions and iterate expression
- **Fourth Cycle**: to design and transform Void type, Type type, Invalid type, and Any type (and operations).

The implementation is broken down into cycles to follow the project plan that have been made during the requirement phase. Please see Chapter 3 and Appendix D for a more detailed list of transformation in each cycle.

4.1 First Cycle

The first cycle of the implementation creates a metamodel for PVS specification language. The task of creating a metamodel is not part of the requirements that were identified in the requirement phase. The metamodel is a prerequisite for the transformation and must be prepared before transformation can start. The metamodel can be created based on reference to the main language constructs of PVS or by referring to the PVS syntax tree [64].

The PVS syntax tree explains the syntactic structure of PVS where a syntax structure is composed of other syntax structure. A metamodel created based on a syntax tree will represent the composition of the syntax structure; the result is a model with elements that have a composition relationship with other elements. The advantage of creating a metamodel based on a syntax tree is the metamodel will have elements that represent the detailed structure of the language. However, this detailed model is created at the expense of simplicity. For this reason, this project creates PVS metamodel based on the main language constructs which results in a more abstract metamodel.

There are five groups of language constructs in PVS: theories, types, type expressions, declarations and expressions. As previously mentioned in Section 2, theory is the main language construct of PVS specifications. Inside a theory, one can have declarations of different types and language construct. Declarations consist of PVS expressions and PVS types are declared using type expression.

Figure 4.1.1 shows the main packages in the PVS metamodel. The next subsection will explain the contents of each package.

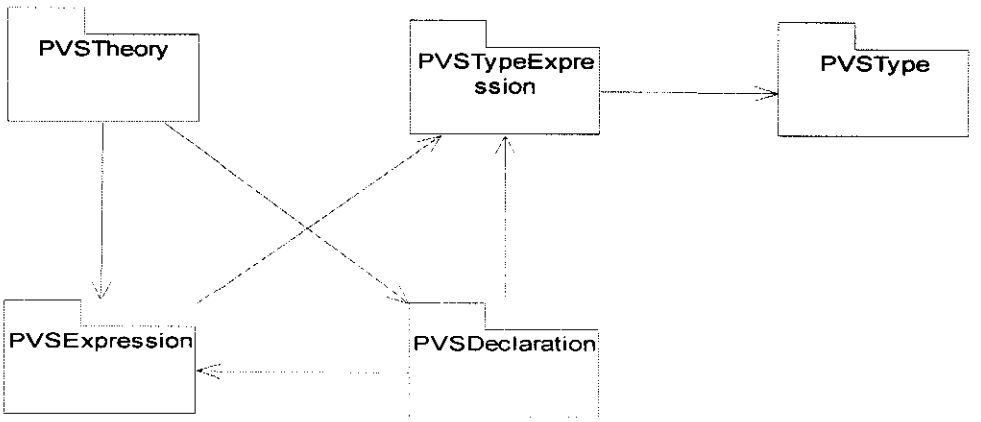


Figure 4.1.9: Packages in PVS metamodel

4.1.1 PVS Types

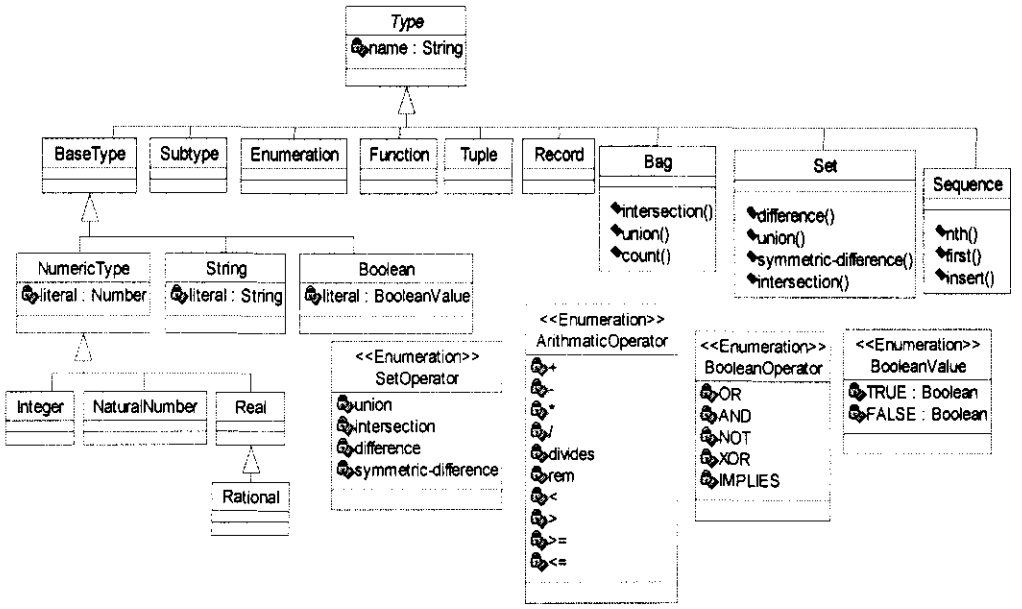


Figure 4.1.10: Metamodel of PVS Types

Figure 4.1.2 is the metamodel for PVS types. This is not a complete set of all PVS types because the type system in PVS can be extended by importing libraries. Subtype is based on a type but the values are constrained by predicates. Subtype is an important part of PVS language because it helps simplify predicates, and functions are expressed over a subtype in order to make it total [64]. Functions are functions in set theory and a tuple is a list of values. Record is the same as tuple but order is not important [64]. Set, Bag and Sequence are PVS libraries for Set type, Bag type and Sequence type respectively. Set and Sequence library is provided in PVS prelude while Bag is taken from the NASA PVS library.

4.1.2 PVS Theory

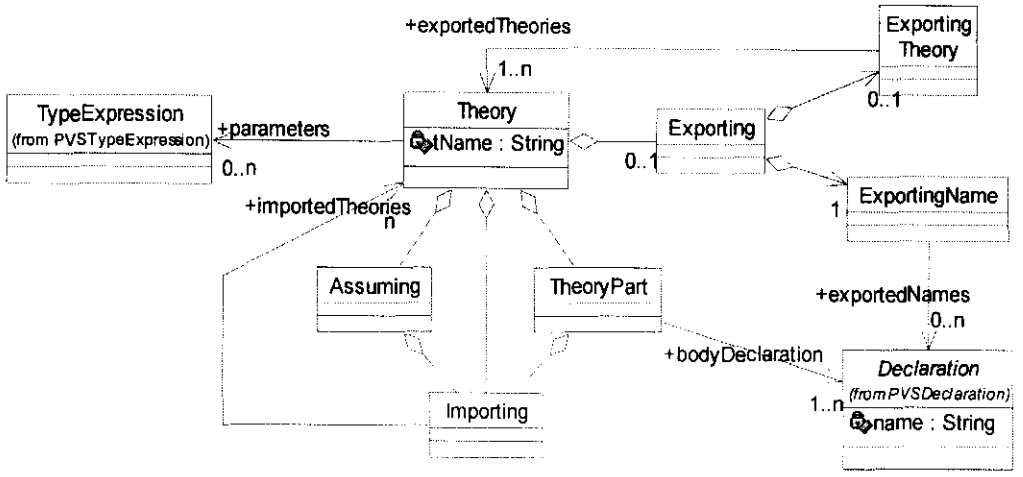


Figure 4.1.11: Metamodel of PVS Theory

PVS theories consist of importing clauses, exporting clauses, assuming block and theory block. Importing clauses are statements to import external theories while exporting clauses specify part of the theory that can be shared with other theories that imports the theory. Export clauses can be used to specify a public interface of the theory and control the accessibility of elements in the theory. Assuming block contains Assumptions that are used to specify constraints for the theory while the theory block is the body of the theory where axioms and proof of the theory will be written.

4.1.3 PVS Type Expression

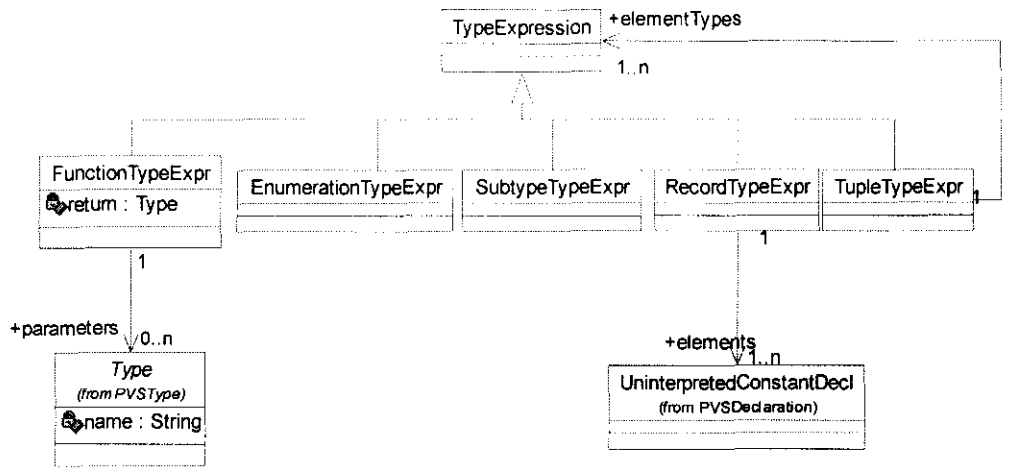


Figure 4.1.12: Metamodel for PVS Type Expression

Type expressions are the general form used to construct different types in the PVSType package. Type expression is used in PVS declaration syntax and PVS expression. TypeExpression class represent a general form of a type in PVS and TypeExpression has subclasses to denote forms for different types in the PVSType package that are different from

the general form. Other types that are not represented by subclasses are represented by TypeExpression. Table 4.1 show the general form for each subclass of TypeExpression.

Table 4.1.11: General form for type expression

Type expression	General form
Function	[parameter type -> return type]
Enumeration	{value1, value2, value3, ...}
Subtype	{variable : type | predicates}
Tuple	[element1, element2, element3, ...]
Record	[# element1: type, element2: type, element3: type ... #]

4.1.4 PVS Declaration

Most PVS statements in theories are declarations. Declarations are statements that introduce PVS entities, where each entity will have an identifier that can be overloaded [64]. Declarations use type expressions in introducing entities. Entities that are typed (have type expression) are variable, macro, inductive, constant and judgement.

Constants in PVS can represent a function, relation or a fixed value. Unlike some programming languages, PVS constants do not need to have a value during its declaration (un-interpreted constants). Functions and relations are interpreted constants (and must have a value) and the value is the function/relation body. Function can also be declared as a type but the difference between a function declared as a type and as a constant is a function that is declared as a type only specify its signature while constant function is the actual function with signature and body. For example, *func:TYPE=[int -> int]* is a function type and we can create function *g:func=(lambda (x:int): x+1)* and function *n:func=(lamda (y:int): y*y)*. This feature is very helpful in representing abstract methods.

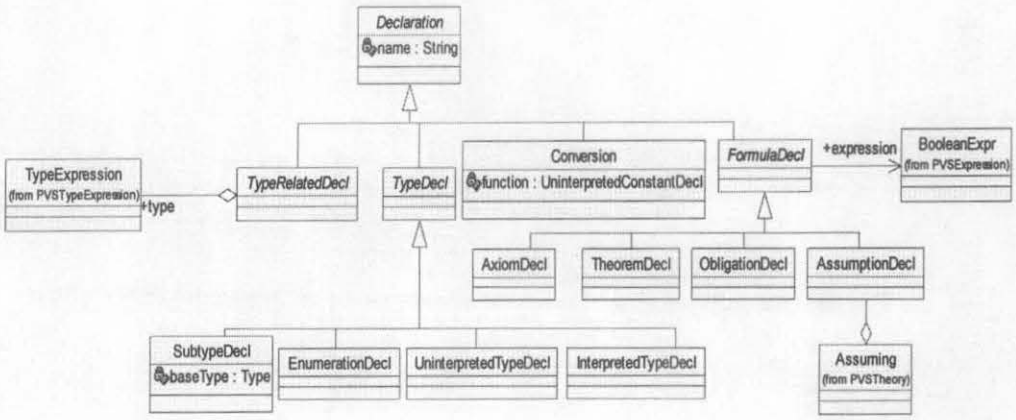


Figure 4.1.13: Metamodel for PVS Declaration

Judgements are constructs that will help simplify the process of type checking by explicitly stating that a value will be of a specific type. There are different types of judgement as shown in the hierarchy of judgement declaration. Subtype judgement is use to specify that a type

is a subtype of another type, simple judgment use to specify a value is of a given type and function judgement specify that the function will only work with the specified type [64].

Formulas and type are declared without any relation to type expression. Formulas are used when proving a theory and the body of a formula must return a Boolean value [64]. Assumptions can only be declared in the assuming part of a theory.

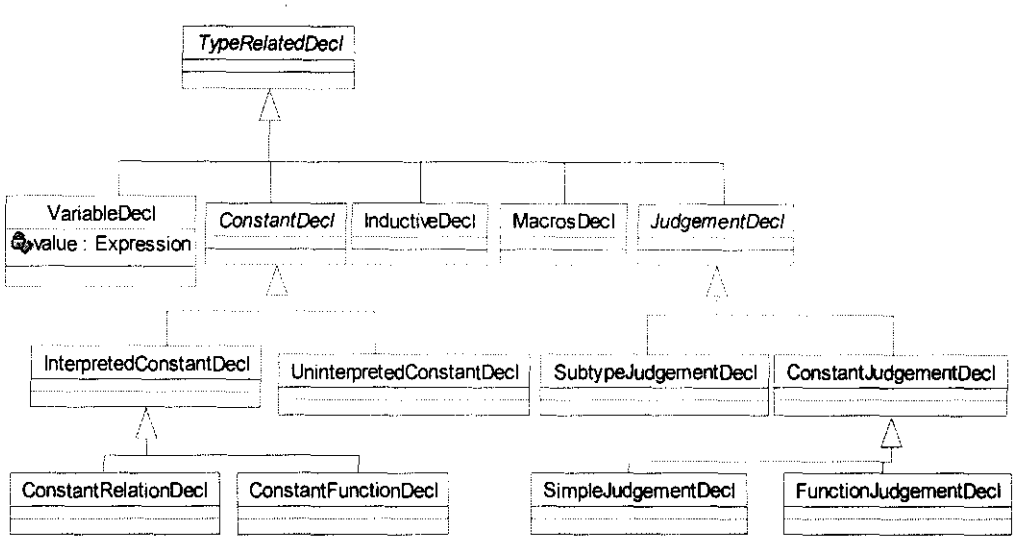


Figure 4.1.14: Metamodel for TypeRelated Declaration

Type declaration introduces a type, where each type that is being introduced must follow the form (to use the type expression) of its type. For example, syntax to declare an enumeration type is *T:Type = {r,g,b}* where *{r,g,b}* is the form for enumeration type in PVS that is taken from PVSTypeExpression package. Interpreted types are a method to give a name to a specific type while uninterpreted types introduce a type that has the minimum constraint on the value it can have [64]. When a new type is declared by constraining the values it can have using predicates on a type, then the new type being declared is a subtype of the type use in its declaration. Subtype can be uninterpreted or interpreted. Interpreted subtypes are followed by a PVS type expression while uninterpreted subtypes only define its parent type.

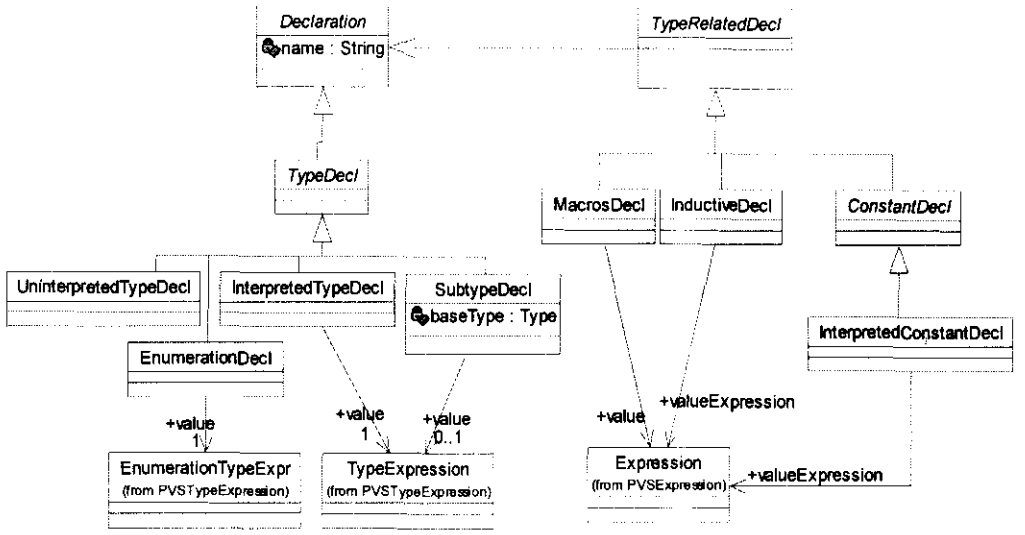


Figure 4.1.15: Detail metamodel for Declaration hierarchy

4.1.5 PVS Expression

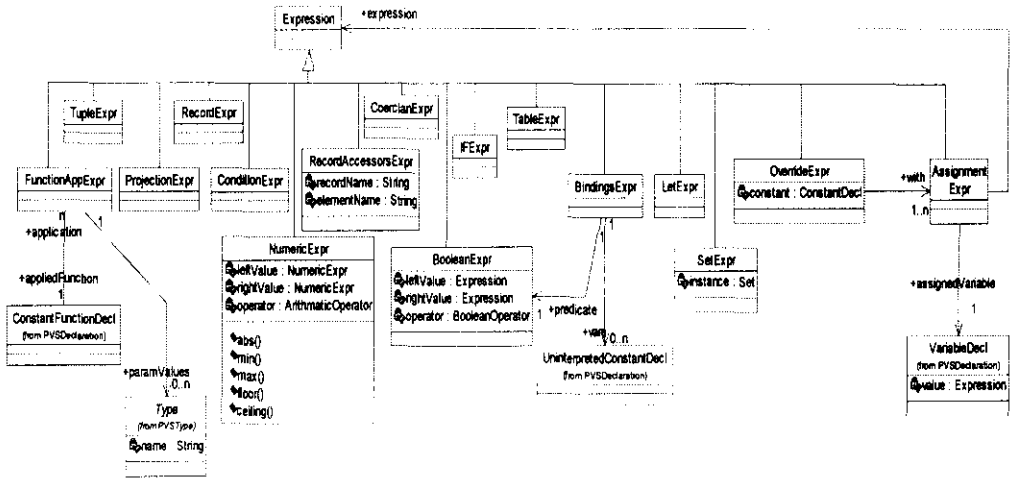


Figure 4.1.16: Metamodel for PVS Expression

The PVSExpression package contains classes that represent different types of expression in PVS. Boolean expressions represent TRUE and FALSE value and operators on Boolean such as AND, OR, IMPLIES and etc [64]. Numeric expressions include numerals and arithmetic operators (+, -, * and /). Tuple and record expressions are expression that gives value to elements of a tuple and record respectively. Values in a tuple and record can be access using projection expression and record accessor expression respectively. Set expressions are expressions that create a set and table expression creates a table.

IF and Condition expressions represent choices. It is equal to condition statements (IF-ELSE and SWITCH-CASE) in programming languages. Override expression is used to override the values of functions, tuples and records. Coercion expressions have the same purpose as judgement, it simplifies type checking. Since PVS allow overloading of identifier, using

coercion expressions will explicitly state to the type checker what the expected type for the identifier.

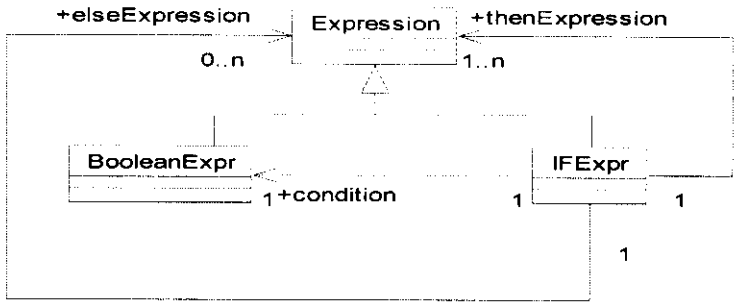


Figure 4.1.17: Structure of PVS IF expression

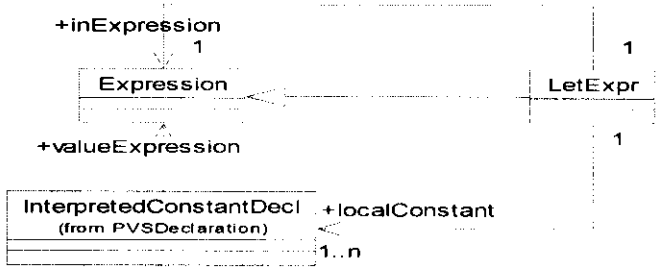


Figure 4.1.18: PVS LET expression

The produced metamodel above is the final PVS metamodel for this project. The metamodel was not completed in just one cycle; it was developed over several cycles base on the incompleteness that is discovered. Next subsections will explain the transformation for each cycle and changes that have been made on the metamodel based on the transformation.

4.2 Second Cycle

The second cycle will focus on meeting requirement FR01.1, FR01.2, FR01.3, FR01.4, FR02.7, FR02.8, FR02.11, FR02.12, FR02.13, FR04 and FR05. The following subsections will discuss about the test plan, design and evaluation for this cycle.

4.2.1 Test Plan

Test Content Transform UML class to PVS theory
Pass Criteria A theory is created with the same name as the class
Test Content Transform UML Property to PVS variable or type

coercion expressions will explicitly state to the type checker what the expected type for the identifier.

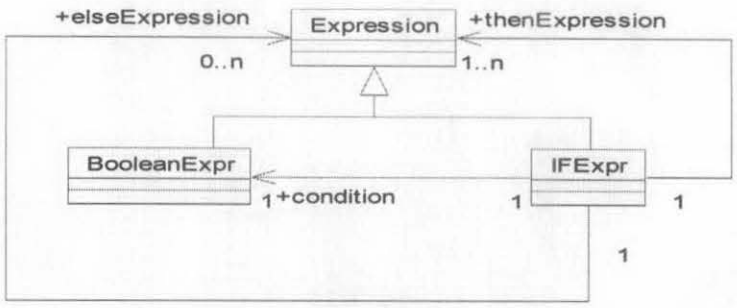


Figure 4.1.17: Structure of PVS IF expression

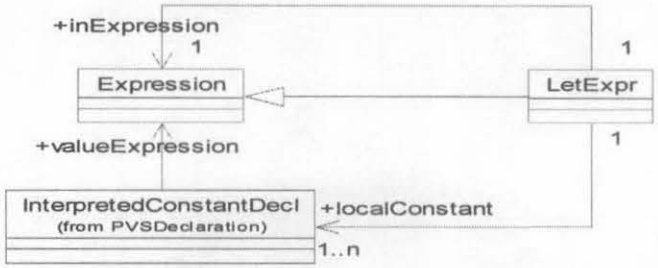


Figure 4.1.18: PVS LET expression

The produced metamodel above is the final PVS metamodel for this project. The metamodel was not completed in just one cycle; it was developed over several cycles base on the incompleteness that is discovered. Next subsections will explain the transformation for each cycle and changes that have been made on the metamodel based on the transformation.

4.2 Second Cycle

The second cycle will focus on meeting requirement FR01.1, FR01.2, FR01.3, FR01.4, FR02.7, FR02.8, FR02.11, FR02.12, FR02.13, FR04 and FR05. The following subsections will discuss about the test plan, design and evaluation for this cycle.

4.2.1 Test Plan

Test Ref. TR01	Req. FR04.1	Input: Class	Output: PVS Theory
Test Content Transform UML class to PVS theory			
Pass Criteria A theory is created with the same name as the class			
Test Ref. TR02	**Req. FR04.2**	**Input: Property**	**Output: PVS variable or type**
Test Content Transform UML Property to PVS variable or type			

Pass Criteria If the property is private, a variable is created. If property is public, an interpreted type is created and included in theory export clause. Variable or type name must be the same as property name. Variable or type base type must be the same as property type			
Test Ref. TR03	**Req. FR04.3**	**Input: Operation**	**Output: PVS function**
Test Content Transform UML method to PVS function, variable or constant			
Pass Criteria If the operation has arguments, PVS function created must have the same name, parameters and type with the operation. If the operation does not have arguments and visibility is private, PVS variable created will have the same name as the operation and the same type. If visibility is public and have no arguments, PVS constant created will have the same name and type			
Test Ref. TR04	**Req. FR02.12**	**Input: Operation Call Expression**	**Output: PVS function application expression**
Test Content Transform operation call expression to PVS function application expression			
Pass Criteria Function application expression created should have the correct function name and parameters in the correct order and value.			
Test Ref. TR05	**Req. FR02.13**	**Input: Navigation Expression**	**Output: PVS expression**
Test Content Transform navigation expression to PVS expression that uses the variable created for the association end			
Pass Criteria If the association end is private, then use the created variable. If the association end is public, create a variable for the association end and use the newly created variable.			
Test Ref. TR06	**Req. FR02.11**	**Input: Property Expression**	**Output: PVS expression**
Test Content Transform property expression to PVS expression that uses the variable created for the property			
Pass Criteria If the property is private, then use the created variable. If the property is public, create a variable for the property and use the newly created variable.			
Test Ref. TR06	**Req. FR05.1**	**Input: Invariant**	**Output: PVS assumption**
Test Content Transform class invariants to PVS assumption			
Pass Criteria Created assumption must have semantically similar expression as the invariants.			
Test Ref. TR07	**Req. FR05.2**	**Input: Initial Clause**	**Output: PVS assignment expression**
Test Content Transform OCL initial clause to PVS assignment expression			
Pass Criteria Variable name use is assignment must be the same as property name and the expression must be semantically similar to the expression in OCL initial clause			
Test Ref. TR08	**Req. FR05.3**	**Input: Derive clause**	**Output: PVS assignment expression**
Test Content			

Transform OCL derive clause to PVS assignment expression			
Pass Criteria Variable name use is assignment must be the same as property name and the expression must be semantically similar to the expression in OCL derive clause			
Test Ref. TR09	**Req. FR05.4**	**Input: Body clause and pre-condition clause**	**Output: PVS IF expression**
Test Content Transform OCL body clause to THEN section of PVS IF expression and OCL pre-condition clause to IF section of PVS IF expression			
Pass Criteria A PVS constant function is created with parameters the same as the operation and return a Boolean or type that is returned by the operation. Inside the function there is a PVS IF expression for operation's pre-conditions (IF section) and body (THEN section). In the THEN section, TRUE will be return and in the ELSE section FALSE will be return.			
Test Ref. TR10	**Req. FR05.5**	**Input: Post-condition clause**	**Output: PVS axiom**
Test Content Transform post condition clause to PVS axiom			
Pass Criteria PVS axiom is created. Format of the axiom body is precondition function | TRUE IMPLIES post-condition expression. The created axiom will call precondition function created for the precondition clause or if there is no precondition clause TRUE is used. Post-condition expression must have similar semantic with OCL post-condition clause.			
Test Ref. TR11	**Req. FR01.1**	**Input: Integer type**	**Output: PVS numeric expression**
Test Content Transform integer literal and operation to PVS numeric expression.			
Pass Criteria Integer literal is transform to the same literal in PVS and OCL div and mod operation will be transform to PVS function			
Test Ref. TR12	**Req. FR0 1.2**	**Input: Real type**	**Output: PVS numeric expression**
Test Content Transform real type, literal and operation to PVS numeric expression.			
Pass Criteria Real literal is transform to the same literal in PVS and OCL floor, round, <, >, <=, >= operation will be transform to PVS equivalent operator and function			
Test Ref. TR13	**Req. FR01.2, FR01.1**	**Input: Number type**	**Output: PVS numeric expression**
Test Content Transform general numeric operation to PVS numeric expression			
Pass Criteria +, -, *,/ operator and abs, max and min function to PVS equivalent operator and function			
Test Ref. TR14	**Req. FR01.3**	**Input: Boolean type**	**Output: PVS Boolean expression**
Test Content Transform Boolean literal and operator to PVS Boolean expression			
Pass Criteria TRUE, FALSE value and NOT, AND, OR, IMPLIES operator in OCL will be transform to its equivalent value and operator in PVS			

Test Ref. TR15	Req. FR01.4	Input: String	Output: PVS string expression
Test Content Transform String literal and operation to PVS string			
Pass Criteria OCL string operation such as size(), concat(), toReal(), toInteger() and substring will be transform to equivalent PVS string function.			

4.2.2 Design

For our design, a decision has been made to change the notations. Instead of using class diagram notation as shown in Chapter 3, a collaboration diagram will be used instead. Reasons for the changes are discussed in Chapter 5. Figures 4.2.1 to 4.2.19 except for 4.2.13 and 4.2.14 are the design for transformation done in the second cycle.

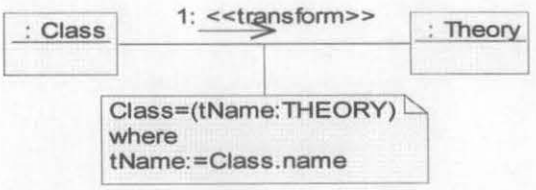


Figure 4.2.19: Class-to-PVS design

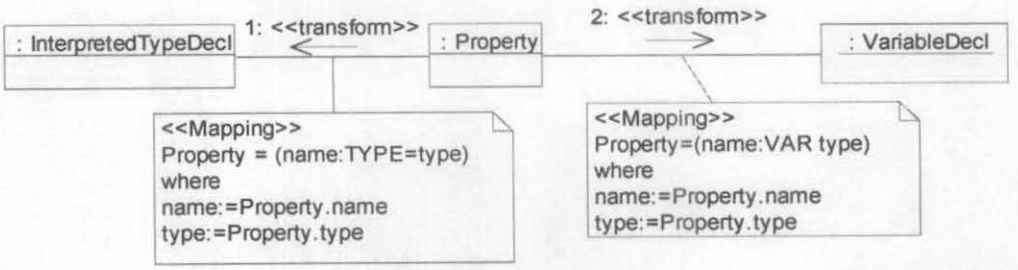


Figure 4.2.20: Property-to-PVS and Association end-to-PVS design

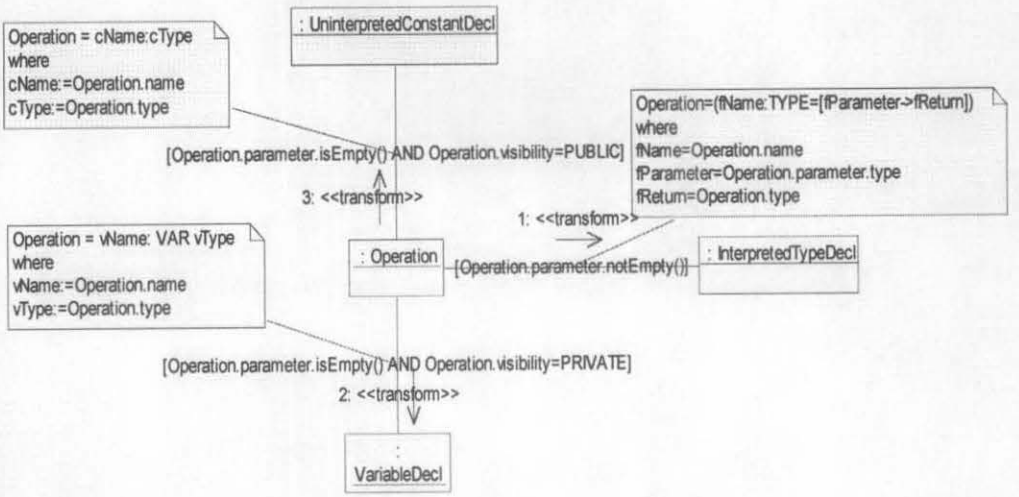


Figure 4.2.21: Operation-to-PVS design

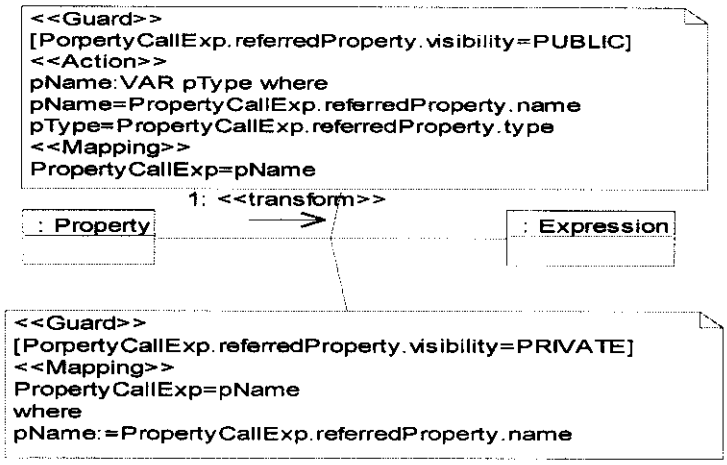


Figure 4.2.24: PropertyCallExp-to-PVS design

Transformation of property call and navigation call expression is similar. Both expressions will be transformed into a PVS expression depending on its visibility. If the visibility is public a variable is created first and the variable name will be used to represent the navigation call or property call expression. This condition exists because public properties are transformed into a type while private properties are transformed into variable. Since private properties are already transformed into variable, property call or navigation call expression for those properties will only be represented by the variable name.

Figures 4.2.7 to 4.2.10 are the design for transformation of OCL constraints (invariants, pre conditions, post conditions, initial clause and derive clause).

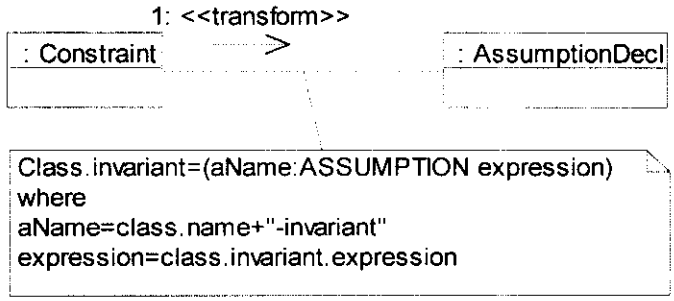


Figure 4.2.25: Invariant-to-PVS design

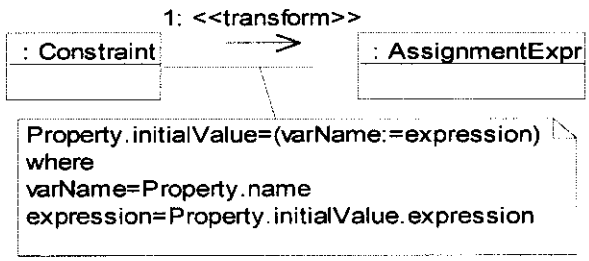


Figure 4.2.26: Initial clause-to-PVS design

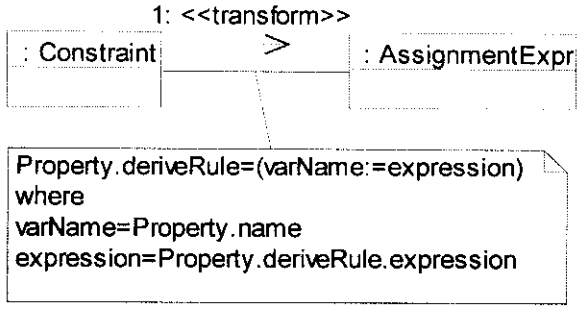


Figure 4.2.27: Derive clause-to-PVS design

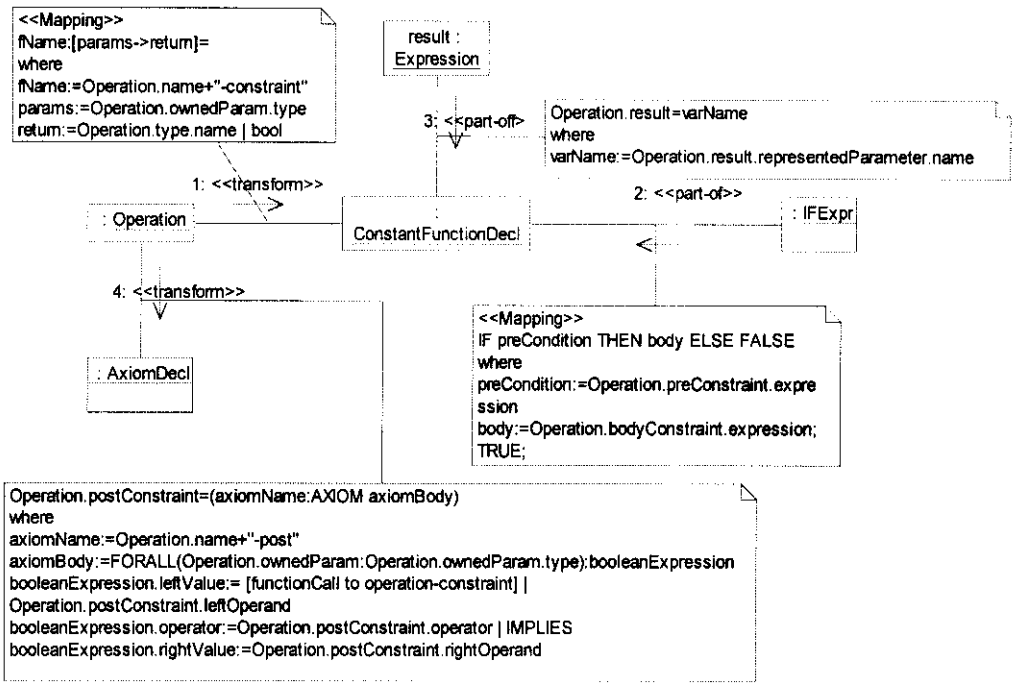


Figure 4.2.28: OperationConstraint-to-PVS design

Pre conditions, body clause and result are transform into a function where inside the function there is an If expression. Pre conditions will be transformed into condition part of the IF expression while body clause will form an expression in THEN block. Post conditions are transformed into axioms over the functions created for the preconditions.

Class invariants are transformed into assumptions because, same as invariants, assumptions must be true at all times [64]. Initial and derive expression for properties are transformed into PVS assignment expression.

Changes has been made to UML metamodel used in this project where an association from operation to OCL expression has been added. The OCL expressions represent the result of the operation using the result keyword.

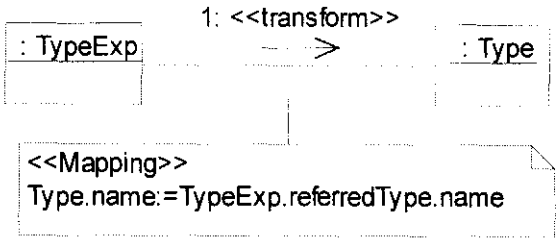


Figure 4.2.29: TypeExp-to-PVS design

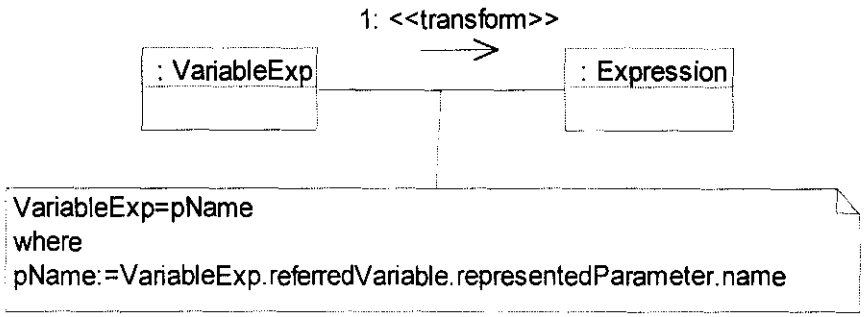


Figure 4.2.30: VariableExp-to-PVS design

For the transformation of OCL basic types (and its operations) changes have been made to the OCL metamodel used in this project. A hierarchy of OCL standard library has been added to OCL expression package. Enumerations have been added to the metamodel to represent operators for OCL base types. Figure 4.2.13 and 4.2.14 show the OCL standard library enumerations and metamodel respectively.

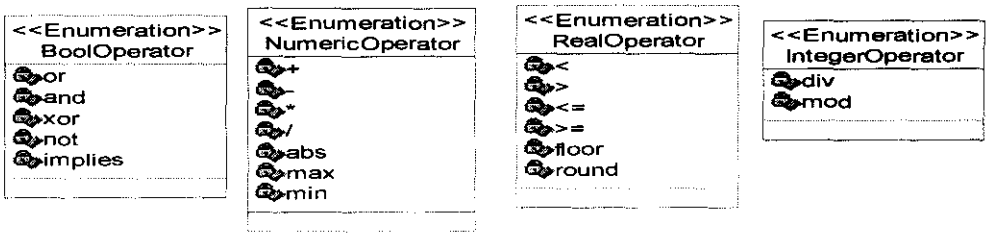


Figure 4.2.31: Enumerations added to OCL metamodel

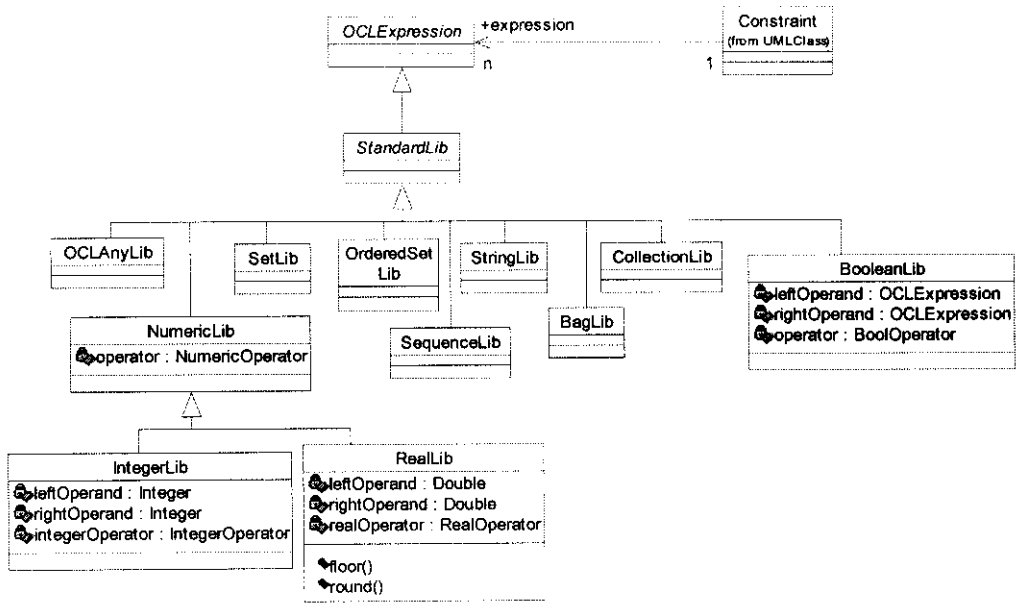


Figure 4.2.32: OCL standard library hierarchy

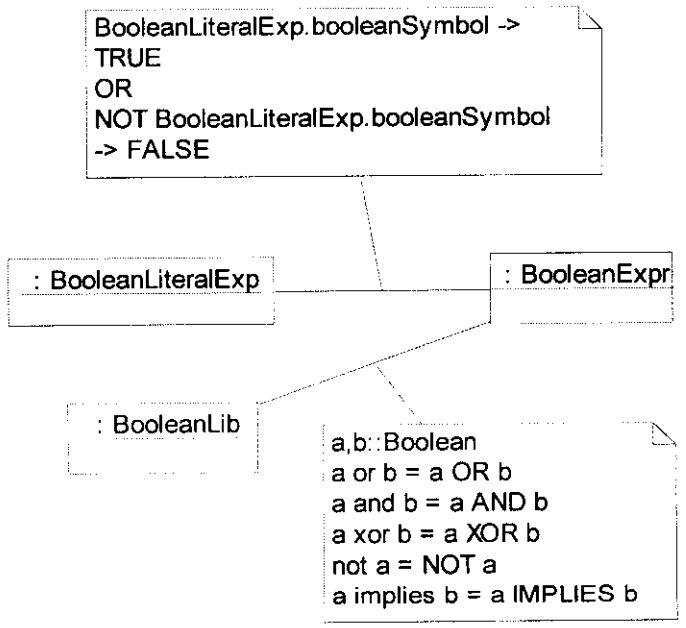


Figure 4.2.33: Boolean-to-PVS design

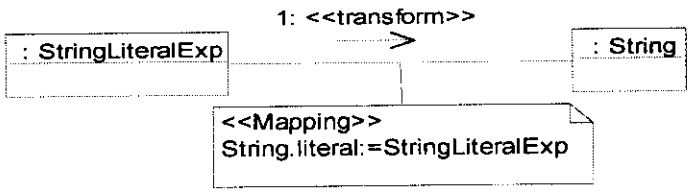


Figure 4.2.34: StringLiteral-to-PVS design

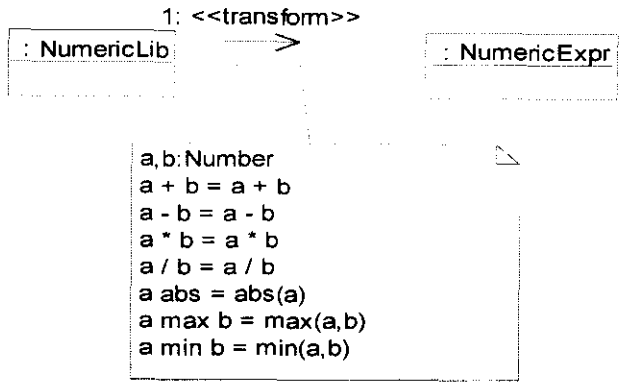


Figure 4.2.35: General Numeric Operation-to-PVS

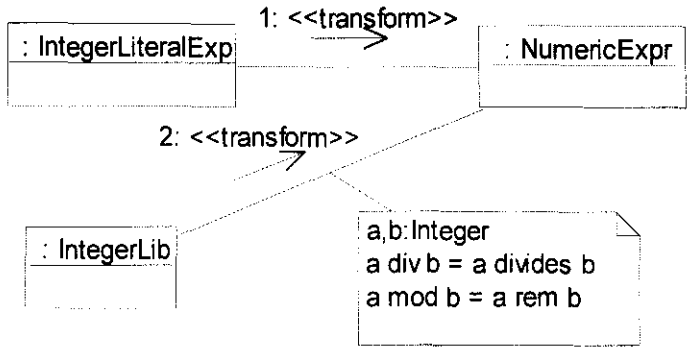


Figure 4.2.36: Integer-to-PVS design

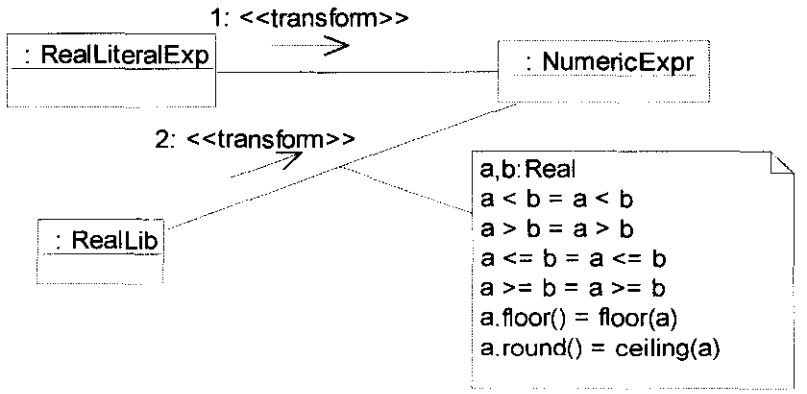


Figure 4.2.37: Real-to-PVS design

4.2.3 Evaluation

Changes have been made to the PVS metamodel created in the first cycle. Enumerations have been added to PVS metamodel to represent operators for Boolean and arithmetic. Other main changes to the PVS metamodel include the identification of association for PVS expression, declaration and type expression to other elements in the metamodel, adding AssignmentExpr elements that inherits from Expression, adding ExportingTheory and

ExportingName elements and adding attributes and methods to NumericExpr and BooleanExpr. ExportingTheory represents expressions that exports theory and ExportingName represent expression that exports other elements such as type and formulas. Table 4.2.1 summarise the associations identified. Multiplicities, directions and roles for these associations can be viewed in the metamodel presented in section 4.1.

Table 4.2.12: Association among PVS elements identified in the second cycle

Elements	Associations
InterpretedTypeDecl and SubtypeDecl	Associated to TypeExpression
EnumerationTypeDecl	Associated to EnumerationTypeExpr
MacrosDecl, InductiveDecl and InterpretedConstantDecl	Associated to Expression
FunctionAppExpr	Associated to ConstantFunctionDecl and Type
AssignmentExpr	Associated to Expression
Theory	Associated to TypeExpression
FunctionTypeExpr	Associated to Type
Exporting	Associated to ExportingTheory and ExportingName
ExportingTheory	Associated to Theory
ExportingName	Associated to Declaration

Transformation program and comparison program had been created for all the transformation in this cycle as specified in the requirements in Section 3.1 and OCL libraries in Appendix D except transformation of OCL String standard library, and self keyword.

The Self keyword cannot be transformed because there is no equivalent language construct in PVS. The Self keyword is used to represent the current instance of the OCL context. If the context is a class, self can be used the same as *this* keyword in Java. Since PVS does not directly support OO concept such as encapsulation and instances (the concept that the self keyword is based on), there is no transformation of the self keyword. Instead transformation of OCL statement such as self.age or self.getSalary() will be considered as a PropertyCallExp or NavigationCallExp for the first statement and OperationCallExp for the second statement. The Self keyword in both examples will be omitted.

Operation in OCL standard library for String cannot be transformed because the String library provided in PVS is not suitable to represent OCL String. In PVS, the String library represents a string as a finite sequence of character and there are no functions associated with the String library that is equivalent to operations on OCL String.

4.3 Third Cycle

OCL main concepts that will be transformed in this cycle are Tuple (requirement FR01.9), Set (requirement FR01.6), Ordered Set (requirement FR01.5), Sequence (requirement FR01.7), and Bag (requirement FR01.8). Each type has their own set of functions in OCL standard library (requirement FR02.8) and OCL Iterator expressions (requirement FR02.9). This cycle will also transform OCL Iterate expression (requirement FR02.10) which is the *iterate* function that can be used with any Collection subtype.

Similar to what we did in previous subsection, this section will be divided into three main parts; Test Plan, Design and Evaluation.

4.3.1 Test Plan

Test Ref. TR15	Req. FR01.9	Input: Tuple	Output: PVS Record
Test Content OCL Tuple type shall be transform into RecordTypeExpr and accessing tuple element expression shall be transform into PVS RecordAcessorExpr			
Pass Criteria OCL tuple shall be transform into PVS record. Element of the tuple will be transform into element of the record that have the same name and type			
Test Ref. TR16	**Req. FR01.8**	**Input: Bag**	**Output: PVS Bag**
Test Content OCL Bag shall be transform into PVS bag type taken from Bag library			
Pass Criteria A new bag shall be created from OCL expression that returns a bag.			
Test Ref. TR17	**Req. FR01.7**	**Input: Sequence**	**Output: PVS Sequence**
Test Content OCL Sequence shall be transform into PVS sequence			
Pass Criteria A new bag shall be created from OCL expression that returns a sequence			
Test Ref. TR18	**Req. FR01.6**	**Input: Set**	**Output: PVS Set**
Test Content OCL Set shall be transform into PVS Set			
Pass Criteria A new set shall be created from OCL expression that returns a set			
Test Ref. TR19	**Req. FR01.5**	**Input: Ordered Set**	**Output: PVS Sequence**
Test Content OCL Ordered Set shall be transform into PVS Sequence			
Pass Criteria A new sequence shall be created from OCL expression that returns an ordered set			
Test Ref. TR20	**Req. FR02.10**	**Input: Iterate Expression**	**Output: Recursive Function**
Test Content Iterate expression shall be transform to a recursive function			
Pass Criteria A constant function declaration called iterate will be created. Iterate function receive a set as a parameter and returns a set. Inside the recursive function there will be a recursive called to the function and a termination rule that state the function will stop calling itself when there are no other elements in the set.			
Test Ref. TR21	**Req. FR02.9**	**Input: Iterator Expression**	**Output:**
Test Content Call to iterator expression shall be transform to functions taken from libraries for the specified type (Set, Sequence or Bag)			
Pass Criteria OCL iterator expression shall be transform to equivalent function that can return output of the correct type			
Test Ref. TR22	**Req. FR02.8**	**Input: Standard library function for Collection**	**Output: Functions from PVS libraries**

and its subtype
Test Content Call to function for collection and its subtype shall be transform into functions or lemmas
Pass Criteria The PVS functions or lemmas must return the same result of the same type as the one return by OCL functions.

4.3.2 Design

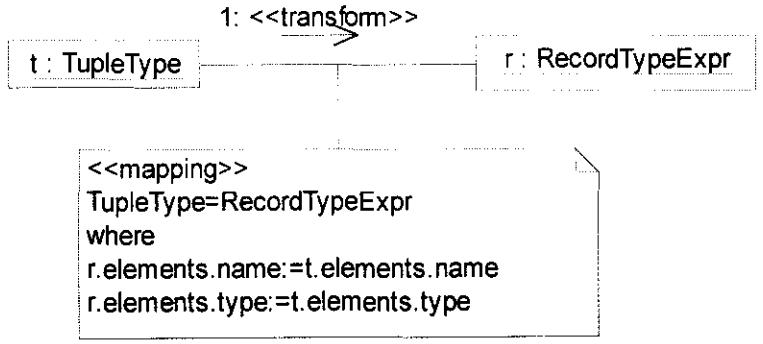


Figure 4.3.38: Tuple Type-to-RecordType Expression

From Figure 4.3.1 OCL tuple type is transform PVS record type. The PVS record will have elements with same name and type as in the OCL tuple. For this purpose change is made to OCL metamodel used in this project. An association from TupleType to Variable is added to the metamodel. The Variable represents the elements in the tuple. Changes are also made to the PVS metamodel. An association between RecordTypeExpr to UninterpretedTypeDecl is added. In this relationship UninterpretedTypeDecl represents the elements in the record. For the purpose of completeness an association is also created from TupleTypeExpr to TypeExpression where the TypeExpression represent elements in the tuple.

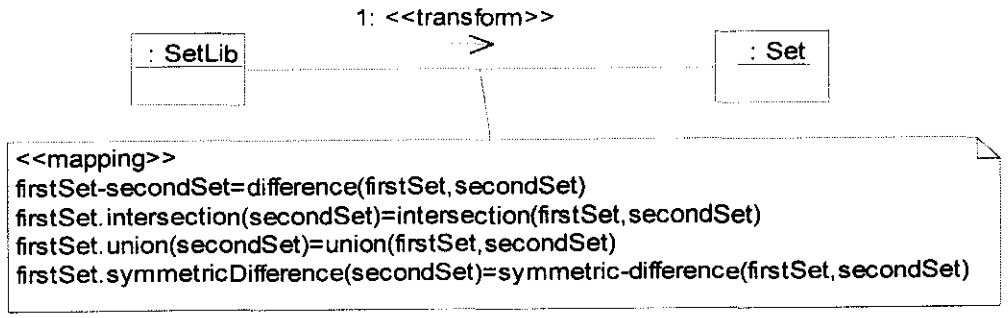


Figure 4.3.39: Set library-to-Set Expression

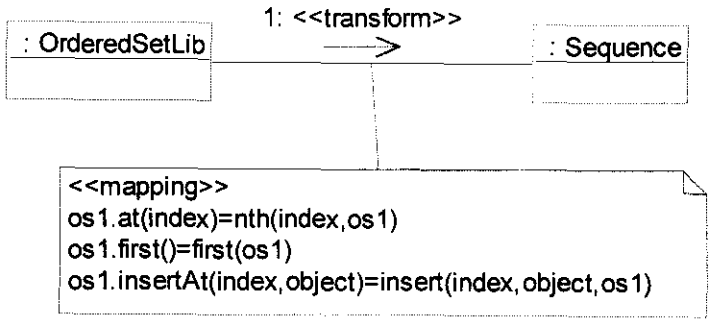


Figure 4.3.40: OrderedSet Library-to-Sequence

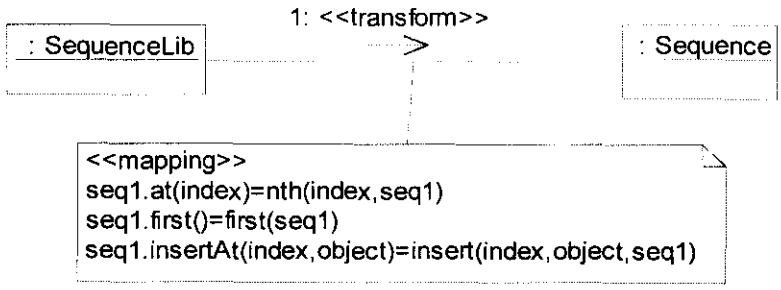


Figure 4.3.41: Sequence Library-to-Sequence

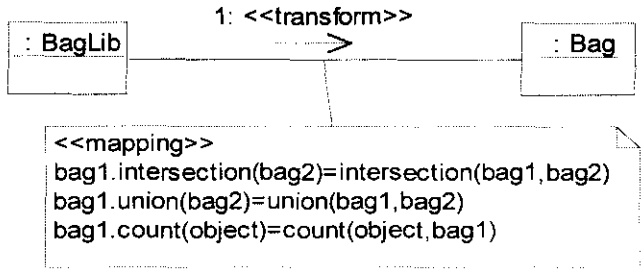


Figure 4.3.42: Bag Library-to-Bag

Figure 4.3.2, 4.3.3, 4.3.4 and 4.3.5 is the transformation of OCL Set, Ordered Set, Sequence and Bag. OCL Set, Sequence and Bag are transformed into PVS Set, Sequence and Bag. OCL Ordered Set is also transform into PVS Sequence. As part of transforming Set, Ordered Set, Sequence and Bag, their operations in OCL standard library will also be transformed to their equivalent PVS function. To transform this operations, new elements is added to the OCL metamodel as shown in Figure 4.3.6, 4.3.7, 4.3.8 and 4.3.9.

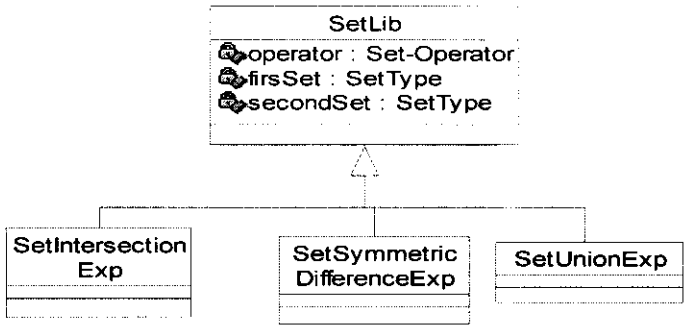


Figure 4.3.43: Subclasses of operation for sets in OCL standard library

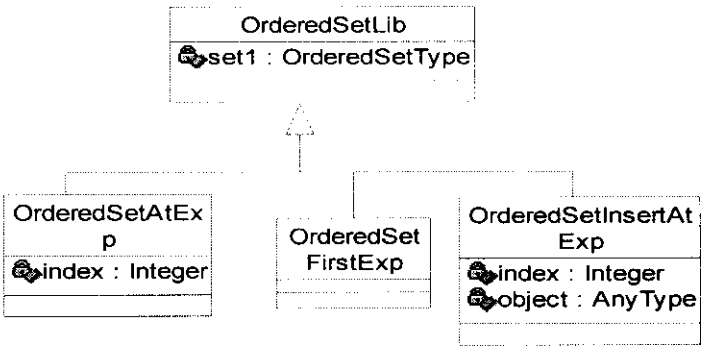


Figure 4.3.44: Subclasses of operation for ordered sets in OCL standard library

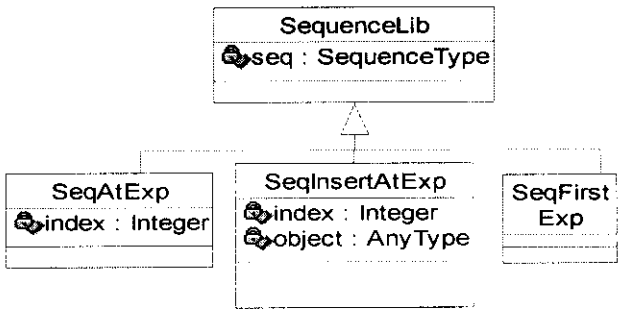


Figure 4.3.45: Subclasses of operation for ordered sequences in OCL standard library

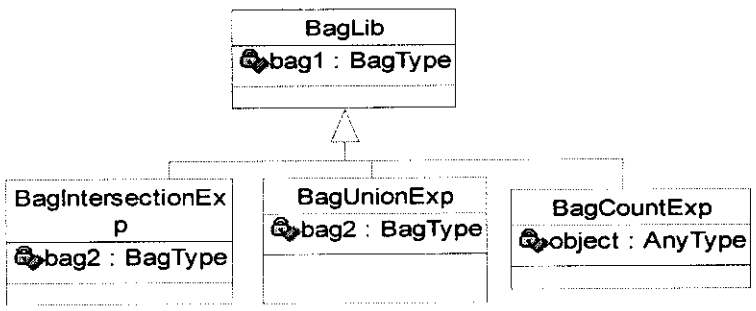


Figure 4.3.46: Subclasses of operation for ordered bags in OCL standard library

Figure 4.3.6, 4.3.7, 4.3.8 and 4.3.9 only show the operation that has been successfully transformed. Besides the changes mention earlier an enumeration is also created for Set operators and the only operator for a set is the difference (-) operator. The enumerations and new elements added to represent operations for Set, Ordered Set, Sequence and Bag are part of a hierarchy of elements that represent all the operations in OCL standard library.

For the purpose of transforming Set, Ordered Set, Sequence and Bag new elements are also added in the PVS metamodel to represent operations provided by PVS Set, Sequence and Bag library. This will be discussed further in the evaluation subsection.

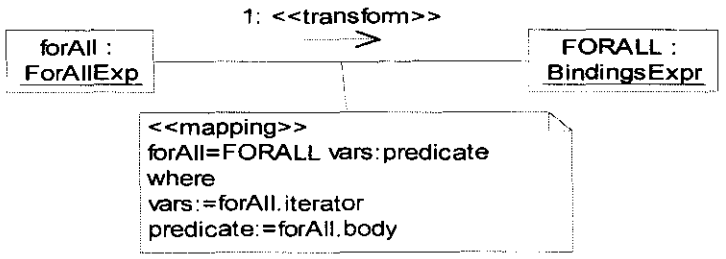


Figure 4.3.47: ForAll Expression-to-Binding Expression

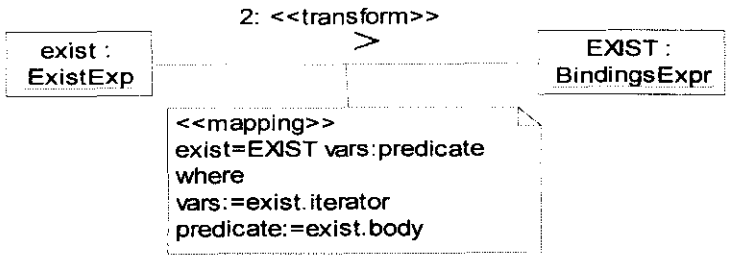


Figure 4.3.48: Exist expression-to-Binding Expression

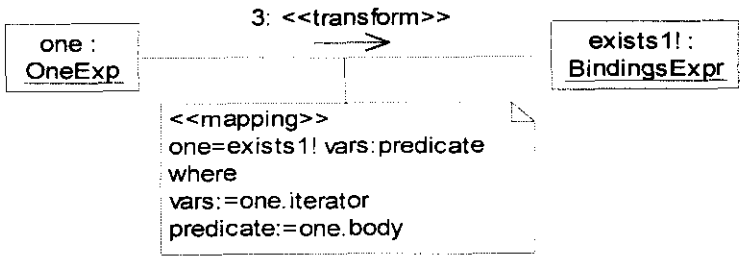


Figure 4.3.49: One expression-to-Binding Expression

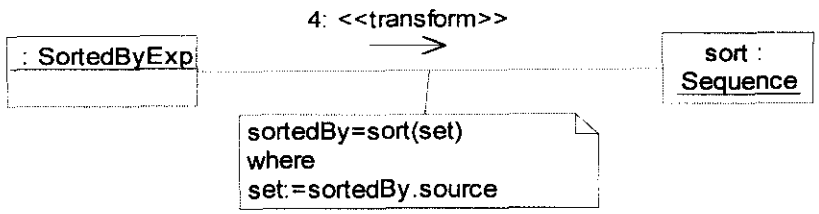


Figure 4.3.50: SortedBy expression to sort operation in Sequence library

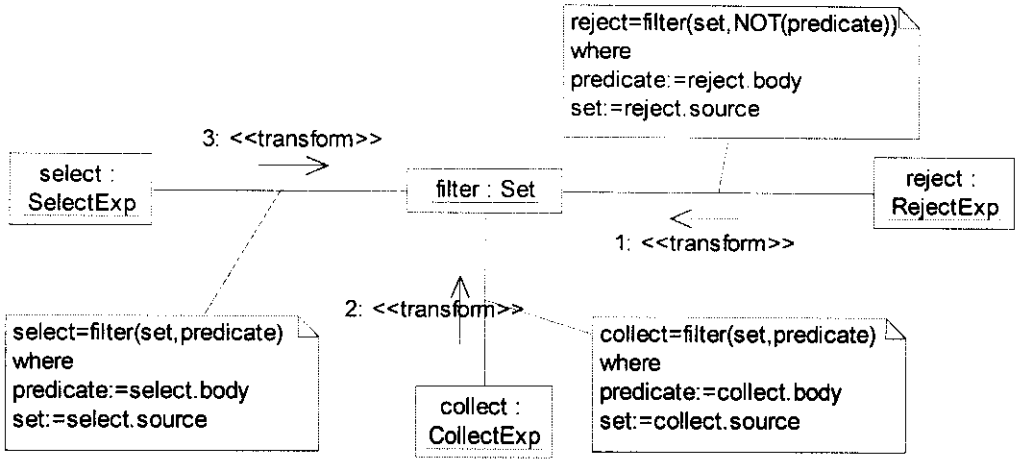


Figure 4.3.51: Select Reject & Collect expression to filter operation in Set library

In order to transform OCL iterator expressions, a hierarchy of OCL iterator expression has been created (Figure 4.3.15). The need for subclasses is to simplify the transformation and it arises when trying to transform OCL IteratorExp to PVS BindingExpr. Not all expressions in IteratorExp are suitable for transformation to BindingExpr. Only *forAll*, *exist* and *one* can be transform into binding expression. This solution is also the solution taken in the metamodel created by Richters and Gogolla [61].

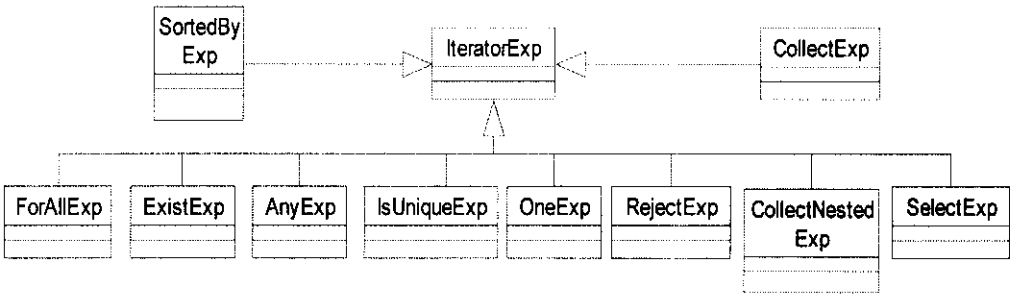


Figure 4.3.52: Subclasses of IteratorExp

For the purpose of the transformation, two new relationships are created for PVS BindingExpr. The first relationship is a directed link from BindingExpr to UninterpretedTypeDecl that represents variable in the binding expression. The second relationship is also a directed link to BooleanExpr. The BooleanExpr represents the predicate in the BindingExpr.

forAll, exists and one operation are transform to PVS binding expression. sortedBy operation are transform to sort operation in PVS Sort_Seq library while select, reject and collect are transform into filter operation in PVS Filters library.

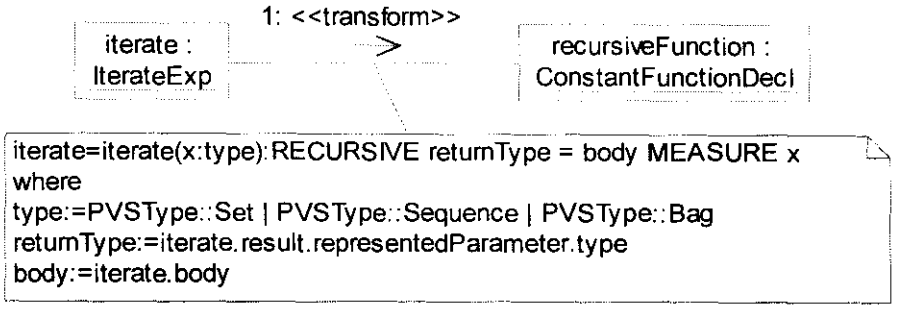


Figure 4.3.53: Iterate Expression to Recursive Function

An OCL iterate expression iterates through all the elements in a collection and generate results from the application of OCL expression inside the iterate expression. OCL iterate expression will be transform into PVS recursive function. PVS recursive function must be total and be able to terminate. This is not a problem because all OCL collections are finite and the termination rule for iterate expression can be the size of the collection. The problem in transforming Iterate expression to recursive function is there are no operations or expressions in PVS that returns the length of a sets, sequences or bags. Also, there are no operations or expression to check if an element is the last element in the set, sequence or bag. Because of this, recursive function cannot be created because there is no way to detect the end of a set, sequence or bag.

A solution to this is creating a PVS function that takes a collection as a parameter and returns the size of the collection. This function can also be used in the transformation of *size* operation in Collection standard library.

4.3.3 Evaluation

The PVS metamodel for Set, Sequence and Bag are refined to include operations from Set, Sequence and Bag libraries. In PVS, operations for a type can be taken from different libraries. For example, filter operation for sets is taken from Filters library. In PVS metamodel that is used in this project, a simple solution to model the library is taken. The project will pretend that there exist a number of classes (one for each type) to represent all the related libraries. The class will have operations from multiple PVS libraries. This solution will cause problems in testing the metamodel because there is no way to compare operations in ECL.

A more suitable and long term solution is to create a model for each library and use something like the Epsilon Merging Language (EML) to merge all related libraries into one model. This solution requires more work to create models for each library and then create merging rules for the models.

Another solution is to manually create a unified model for all the libraries. This solution does not require the development of an EML program but the complexity of creating the model increases because we are trying to merge multiple models manually.

The transformation from OCL to PVS for this cycle is very limited because not all OCL standard library and iterator expression can be transformed to PVS expression. Table 4.3.1 is a list of OCL expressions that can be transformed to PVS.

Table 4.3.13: Transformation of OCL expressions to PVS

OCL Expression	PVS expression
Collection::exist	Binding Expression::EXIST
Collection::forAll	Binding Expression::FORALL
Collection::one	Binding Expression::exists1!
Set::select	Filters::filter
Set::reject	Filters::filter
Set::collect	Filters::filter
Set::"-"	Set::difference
Set::intersection	Set::intersection
Set::symmetricDifference	Set::symmetric-difference
Set::union	Set::union
Bag::"="	NASA-Bag::Bag-equality
Bag::intersection	NASA-Bag::intersection
Bag::union	NASA-Bag::union
Bag::count	NASA-Bag::count
Sequence::sortedBy	NASA-sort_seq::sort
Sequence::at	Sequence::nth
Sequence::first	Sequence::first
Sequence::insertAt	Sequence::insert
OrderedSet::insertAt	Sequence::insert
OrderedSet::at	Sequence::nth
OrderedSet::first	Sequence::first

Some of the OCL expressions that should be transformed in this cycle have similar expression in PVS but are not suitable because the operation return a different type that what is required in the OCL expression. The similarity is in the purpose of the PVS operation. For example, OCL append operation for sequence is similar to PVS *append* operation in list_props library but PVS append can only be used with a list. Table 4.3.2 list out PVS expressions that is similar in purpose but not suitable for transformation and Table 4.3.3 list out the OCL expression that does not have equivalent expressions in PVS.

Table 4.3.14: Non suitable PVS expressions and its OCL expressions

PVS expression	OCL expression
list_props::length	Collection::size
Set::empty?	Collection::isEmpty and Collection::notEmpty
Filters::filter	Bag::select, Bag::reject, Bag::collect, Bag::excluding, Sequence::excluding, Set::excluding
NASA-seq_sort::sort	Set::sortedBy and Bag::sortedBy
list_props::append	Sequence::append and OrderedSet::append

Table 4.3.15: OCL expressions that cannot be transformed

OCL Type	OCL expression
Collection	includes, excludes, includesAll, excludesAll, any, isUnique, iterate, sum and product
Set	asSequence, asBag, asOrderedSet, collectNested, including, count, flatten

Bag	asOrderedSet, asSet, asSequence, collectNested, including, flatten
Sequence	asBag, asSet, asOrderedSet, collectNested, including, prepend, last, subsequence, count, flatten, indexOf
OrderedSet	prepend, subOrderedSet, indexOf, last

As a result of the transformation and investigation in the third cycle, changes to the transformation on UML class elements have been done. Class properties, operations and association ends are transformed into record elements. The record will represent the class that encapsulate the elements. This changes is needed to represent OCL self keyword. By creating a record for each theory and transforming properties, operations and association end to elements of the record, expression such as self.age and self.open() can be transform to record accessor expression.

As a result of changing the transformation of class properties and association end, changes also need to be done on OCL navigation call expression and property call expression. Both expressions are transformed into record accessor expression. If the result of the navigation call expression creates a set, a PVS set is created, if a sequence or ordered set is created, it is transform into a PVS sequence, and if a bag is the result, it will be transform into a PVS bag. Figure 4.3.17, 4.3.18 and 4.3.19 are the design for transformation of property call expression, navigation call expression and class elements respectively. Table 4.3.4 is the new test specification for the changes made on transforming OCL property call expression, navigation call expression and UML class elements.

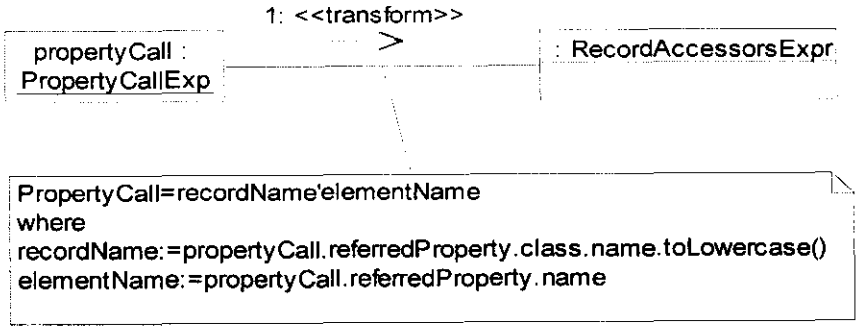


Figure 4.3.54: Property call expression to Record accessor expression

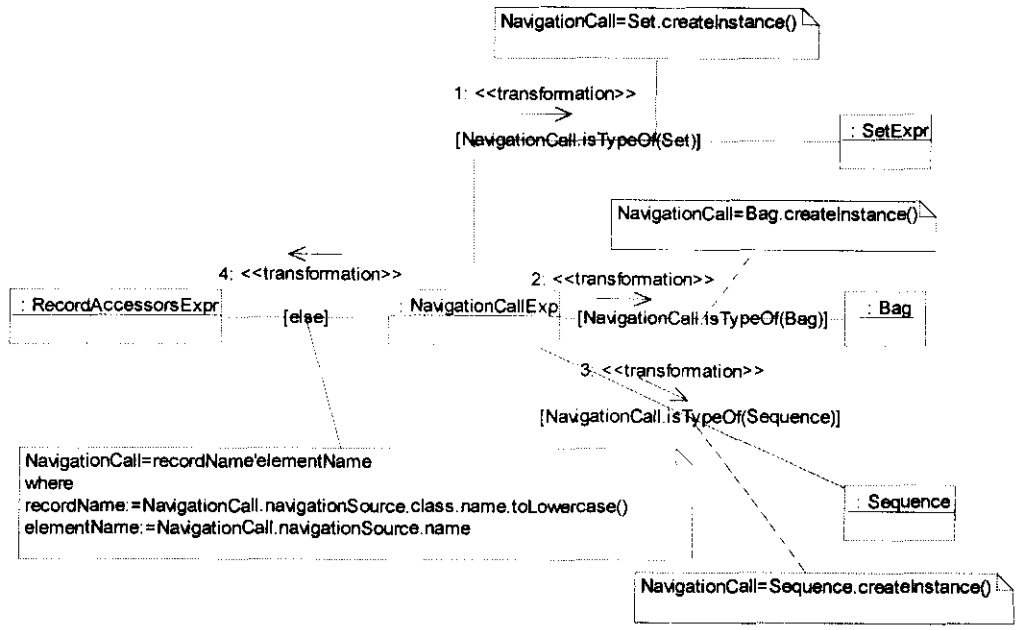


Figure 4.3.55: Navigation call expression to PVS

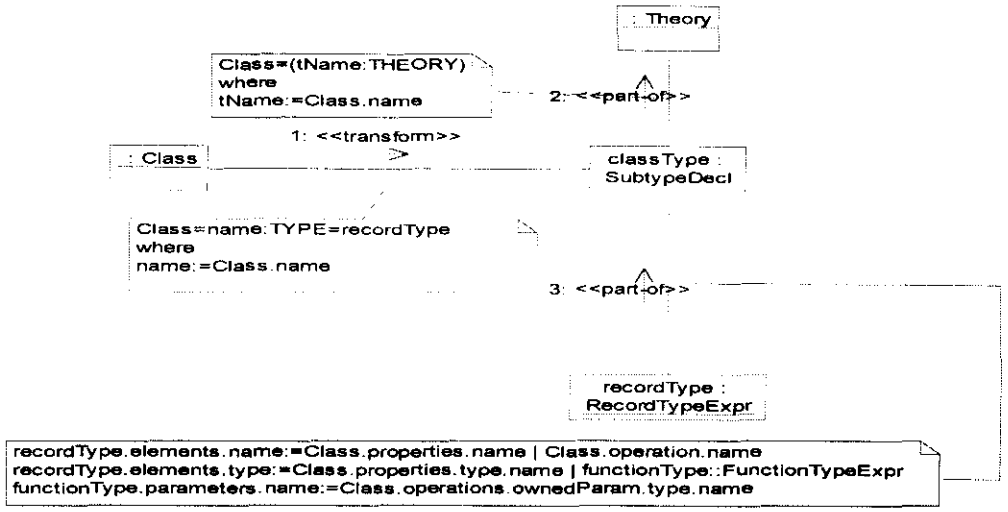


Figure 4.3.56: Class elements to PVS

One drawback of transforming all properties, operation and association end of a class to elements of a record is there is no control over the visibility of elements. In the second cycle, visibility of elements is controlled by transforming private properties and association ends to variables, and public properties to types. Transforming properties and association end to record will allow it to be used by theories that import the theory containing the record.

To control the access of elements from external theories, the transformation program for properties and association end checks the visibility of the properties/association end. If it is PUBLIC, it is transform into an element of the record and if it is PRIVATE it will be transform into a variable. Transformation on the property call and navigation call expression will check the visibility and is transformed accordingly.

Using records for classes also change the transformation of OCL initialise and derive constraint. Before this initial and derive constraint are transform into PVS assignment expression. But assignment expression cannot be used to change the value of one element of the record. To change a partial record (not all element being change) PVS override expression must be used instead.

Using override expression to change the value of record element of type constant is easy but changing the value of functions is more difficult. To make it easier in transforming constraint on operations, operations will not be transformed into element of a record but to a function type, same as the transformation for operations in the second cycle (see Figure 4.2.3). Changes on Figure 4.3.19 are shown in Figure 4.3.20. Figure 4.3.21 and 4.3.22 are the design for initialised and derive constraint respectively. Table 4.3.4 also show the new test specification for initial (FR05.2) and derive constraint (FR05.3).

Table 4.3.16: Test specification for new transformation of FR04, FR02.11, FR02.13, FR05.2 and FR05.3

Test Ref. TR34	**Req. FR04**	**Input: UML class elements**	**Output: Various PVS expression**
Test Content UML class elements shall be transformed to various PVS expressions			
Pass Criteria A theory is created with the same name as the class. A record is also created where the elements are class public properties, public association ends. Private properties and association ends is transformed into variable. An Operation is transformed into a function type that returns a type that is the same as what the operation returns and accept arguments same as the one accepted by the operations. Type of properties and association ends are maintained. A constant of type record created is also created.			
Test Ref. TR35	**Req. FR02.11**	**Input: OCL PropertyCall**	**Output: PVS expression or record accessor expression**
Test Content Property call shall be transformed into PVS expression or record accessor expression depending on visibility of the properties			
Pass Criteria If the property is public, record accessor will be created where the name of the class that owns the property in lower case and element name that is the same as the property name. Private properties are transformed into PVS expression that is the same as the name of the property.			
Test Ref. TR36	**Req. FR02.13**	**Input: OCL NavigationCall**	**Output: Various PVS expression**
Test Content Navigation call shall be transformed into a set, bag, sequence or PVS expression depending on the type generated by navigation call. The PVS expression can be a general expression or record accessor expression depending on visibility of the association end.			
Pass Criteria Navigation call will be transform into a set, sequence or bag if the navigation calls generates a set, ordered set, sequence or bag respectively. Other wise it is transform into a PVS expression or record accessor expression. If the association end is public then record it will transformed into a record accessor with the record name is the same as the class name that owned the property and the element name is the name of the property. Private association end is transformed into			

PVS expression that is the same as the name of the property. All the elements from set, ordered set, sequence or bag is copied to PVS set, sequence or bag.			
Test Ref. TR37	**Req. FR05.2**	**Input: Initial clause**	**Output: PVS override expression**
Test Content Initial clause will be transform into override expression that overrides value of record element that represent the properties			
Pass Criteria Override expression will change the value of record element that have the same name as the property that is the context of the initial clause. The new value of the record element will be an expression similar to the expression in the initial clause			
Test Ref. TR38	**Req. FR05.3**	**Input: Derive clause**	**Output: PVS override expression**
Test Content Derive clause will be transform into override expression that overrides value of record element that represent the properties			
Pass Criteria Override expression will change the value of record element that have the same name as the property that is the context of the derive clause. The new value of the record element will be an expression similar to the expression in the derive clause			

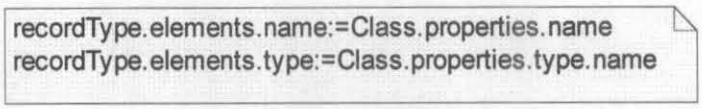


Figure 4.3.57: Changes on Figure 4.48

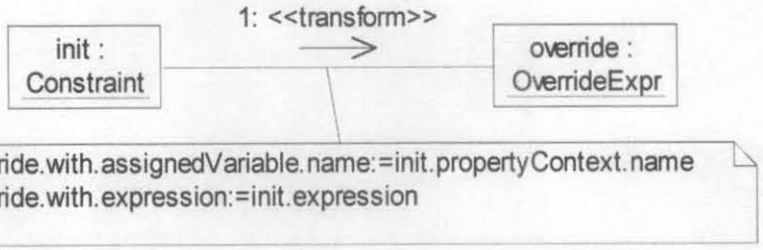


Figure 4.3.58: Transformation of OCL initial clause to PVS override expression

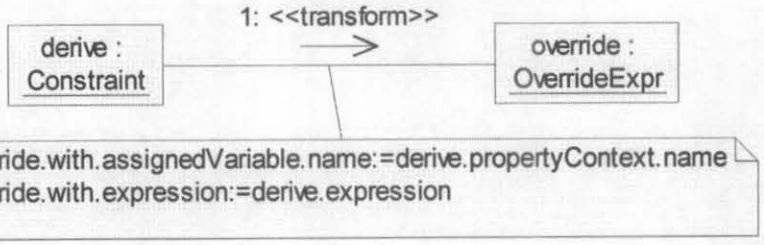


Figure 4.3.59: Transformation of OCL derive clause to PVS override expression

The use of override expression to change the value of record elements requires two constants of the same type. One constant represent the previous state of the record and another

one represent the record with the new value. The constant with the old value can be used in the transformation of property call expression with @pre. Property calls with @pre represent the previous state of the property before the operation and it can only be used in the post condition.

To consider @pre in the transformation of property calls, changes must be made to the OCL metamodel because the current metamodel does not represent @pre. The changes to the metamodel is taken from OCL metamodel in [61] where @pre is represented by a property called isMarkedPre of type Boolean in property call expression. Changes will also be made to the transformation of UML class where the transformation will create two constant of type record that is created to represent the UML class. The new design of the transformation of property call expression is shown in Figure 4.3.23.

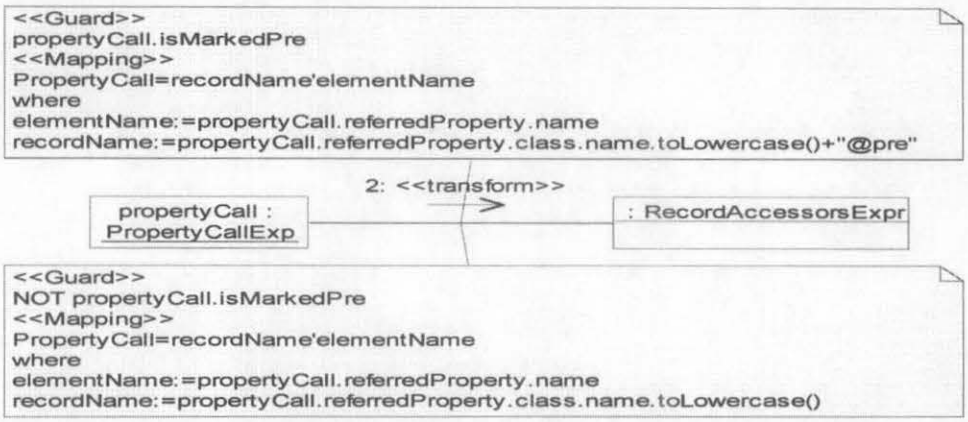


Figure 4.3.60: Transformation of property call expression

4.4 Fourth Cycle

In this cycle, OCL elements that will be transformed are OCLVoid (requirement FR01.10), OCLInvalid (requirement FR01.11), OCLAny (requirement FR01.12), Let expression (requirement FR02.6), If expression (requirement FR02.5) and standard library operations for OCLAny (requirement FR02.8). Subsection 4.4.1, 4.4.2 and 4.4.3 will discuss about the test plan, design and evaluation of the fourth cycle respectively.

4.4.1 Test Plan

Test Ref. TR23	Req. FR02.5	Input: OCL If expression	Output: PVS If expression
Test Content OCL Ifexpression shall be transformed to PVS Ifexpression			
Pass Criteria If condition in OCL shall be transform into condition for PVS If expression. Expressions in Then block shall be transform into equivalent expression in PVS Then block and expressions in Else block shall be transform into equivalent expression in PVS Else block			
Test Ref. TR24	**Req. FR02.6**	**Input: OCL Let expression**	**Output: PVS Let expression**
Test Content			

OCL Let expression shall be transformed to PVS Let expression			
Pass Criteria Local variables in PVS. The Let expression is similar to local variables in OCL's let expression. Expressions for initial value for local variable shall also be equivalent to the ones in OCL Let expression and expressions after IN keyword must also be the equivalent to the ones after 'in' keyword in OCL Let expression.			
Test Ref. TR25	**Req. FR02.8**	**Input: = and <> operator**	**Output: = and /= operator**
Test Content = and <> operator shall be transform to equivalent operator in PVS			
Pass Criteria = operator shall be transform into = operator in PVS while <> operator will be transform to /= operator in PVS. Expression on the left hand side and right hand side of both operators will also be transform to its equivalent expression in PVS.			
Test Ref. TR26	**Req. FR01.12**	**Input: AnyType**	**Output: PVS Uninterpreted Type Declaration**
Test Content OCL Any type shall be transform into uninterpreted type declaration			
Pass Criteria An uninterpreted type called any shall be created			
Test Ref. TR27	**Req. FR01.11**	**Input: InvalidType**	**Output: PVS Uninterpreted Constant Declaration**
Test Content OCL invalid type shall be transformed into uninterpreted constant declaration			
Pass Criteria Uninterpreted constant called invalid shall be created and it is of type any			
Test Ref. TR28	**Req. FR01.10**	**Input: VoidType**	**Output: PVS Uninterpreted Constant Declaration**
Test Content OCL invalid type shall be transformed into uninterpreted constant declaration			
Pass Criteria Uninterpreted constant called undefined shall be created and it is of type any			
Test Ref. TR29	**Req. FR02.8**	**Input: Call to OCLAny::isInvalid()**	**Output: PVS Function Application**
Test Content Call to isInvalid operation shall be transformed into function application expression for the function isInvalid			
Pass Criteria Function application expression for isInvalid function that accept one argument which is the instance that call OCL isInvalid operation			
Test Ref. TR30	**Req. FR02.8**	**Input: Call to OCLAny::isNew()**	**Output: PVS Function Application**
Test Content Call to isNew operation shall be transformed into function application expression for the function isNew			
Pass Criteria Function application expression for isNew function that accept one argument which is the			

instance that call OCL isNew operation			
Test Ref. TR31	**Req. FR02.8**	**Input: Call to OCLAny::isUndefined()**	**Output: PVS Function Application**
Test Content Call to isUndefined operation shall be transformed into function application expression for the function isUndefined			
Pass Criteria Function application expression for isUndefined function that accept one argument which is the instance that call OCL isUndefined operation			
Test Ref. TR32	**Req. FR02.8**	**Input: Call to OCLAny::allInstances()**	**Output: PVS Function Application**
Test Content Call to allInstances operation shall be transformed into function application expression for the function allInstances			
Pass Criteria Function application expression for allInstances function that accept one argument which is the instance that call OCL allInstances operation			
Test Ref. TR33	**Req. FR02.8**	**Input: Call to OCLAny::oclAsType()**	**Output: PVS Conversion**
Test Content Call to oclAsType operation shall be transformed into PVS Conversion			
Pass Criteria Conversion function is created for each call to oclAsType. The conversion function will accept an argument which is the type of the instance that calls oclAsType operation and returns the type that is the argument for oclAsType			

4.4.2 Design

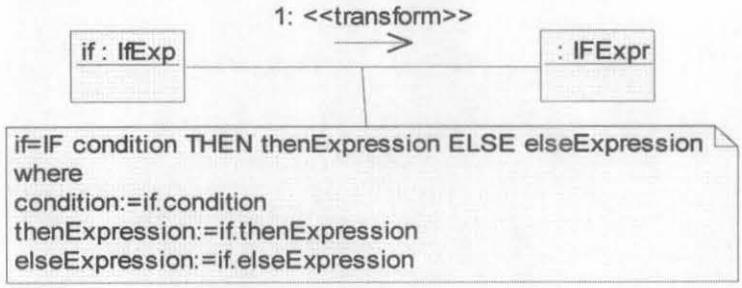


Figure 4.4.61: OCL If expression to PVS If expression

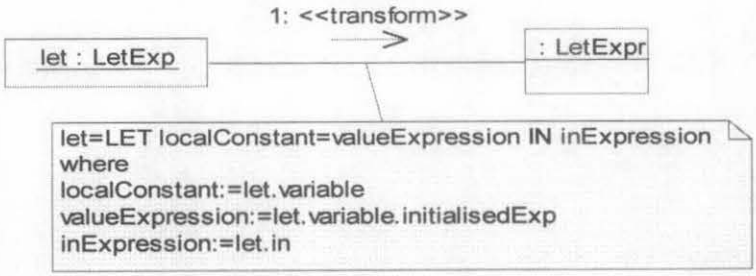


Figure 4.4.62: OCL Let expression to PVS Let expression

OCL If expression and Let expression are transformed into PVS IF expression and LET expression respectively. The transformation cause a few changes on the PVS metamodel created in the first cycle. The changes are made to PVS LET expression. LET expression is now associated with Interpreted Constant as the local variable in the LET expression. IN segment of the LET expression may contain any PVS expressions.

To transform operations for OCLAny type in the OCL standard library, the OCL metamodel used in this project need to be refined. A hierarchy of element to represent OCLAny operations from the standard library has been added. The hierarchy of elements is shown in Figure 4.4.3.

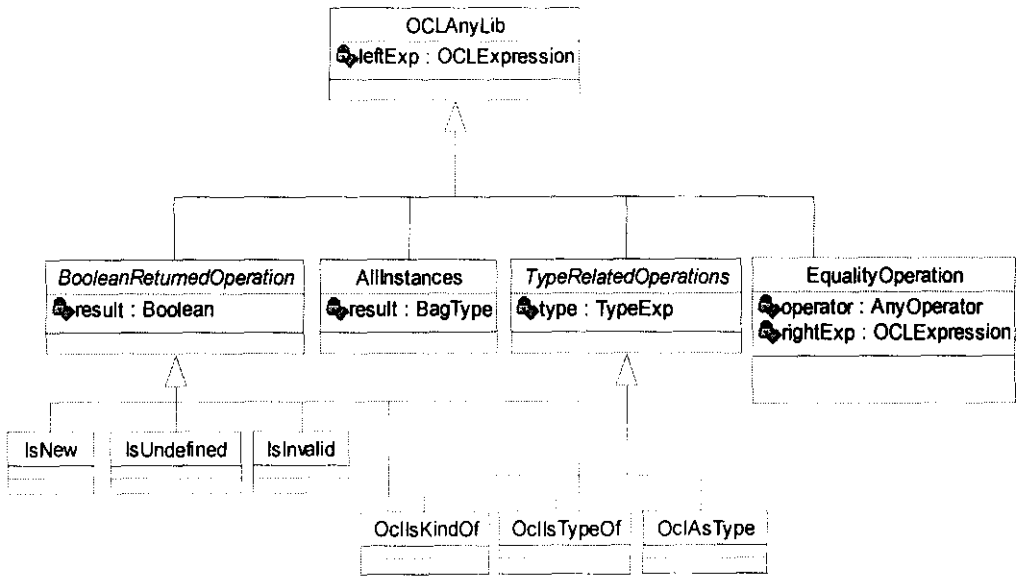


Figure 4.4.63: Hierarchy of elements for OCLAny operations in OCL standard library

From figure 4.4.3 OCL "=" and "<>" operator are transformed into PVS Boolean expression where the operators are "=" and "/=" respectively.

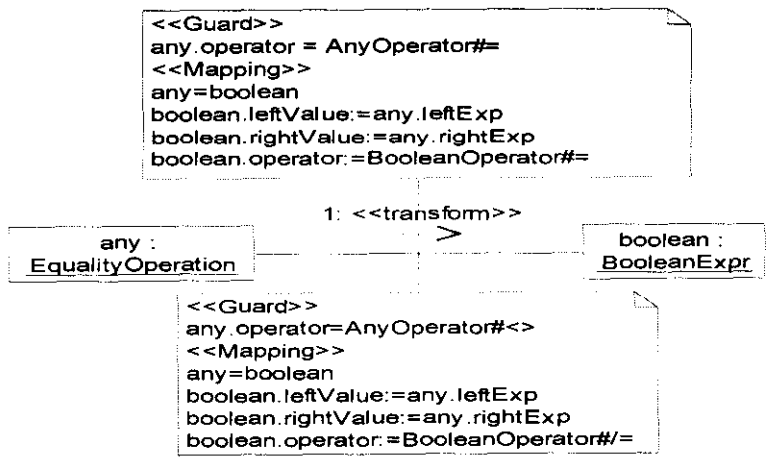


Figure 4.4.64: OCL '=' and '<>' operator to PVS '=' and '/=' operator

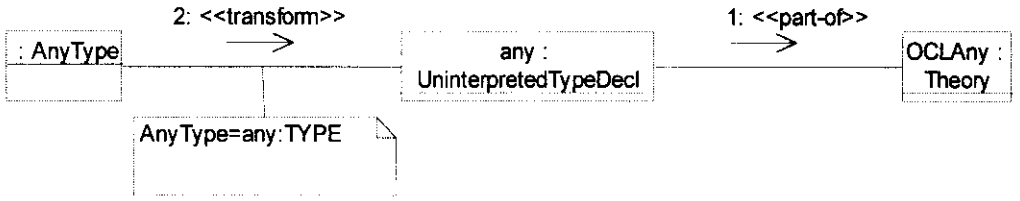


Figure 4.4.65: OCLAny to Uninterpreted Type Declaration

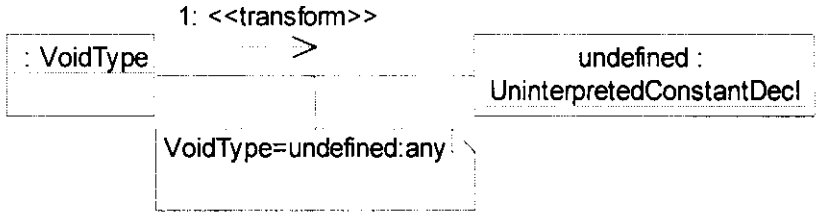


Figure 4.4.66: VoidType to Uninterpreted Constant Declaration

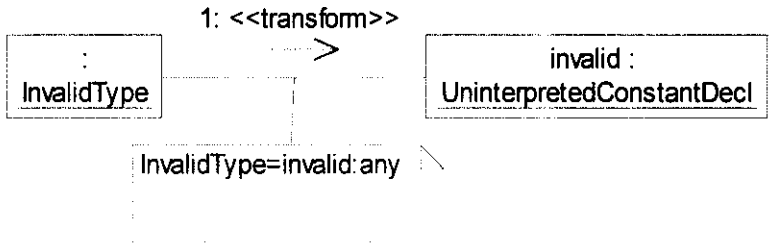


Figure 4.4.67: InvalidType to Uninterpreted Constant Declaration

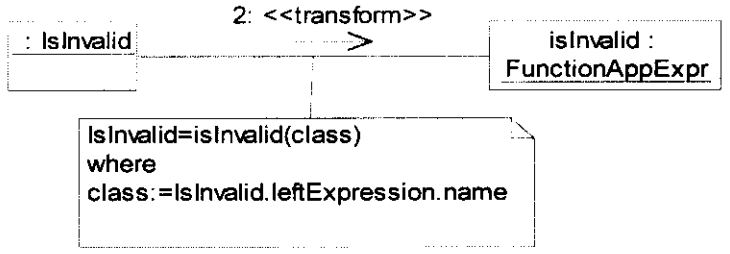


Figure 4.4.68: isInvalid operation to PVS function application

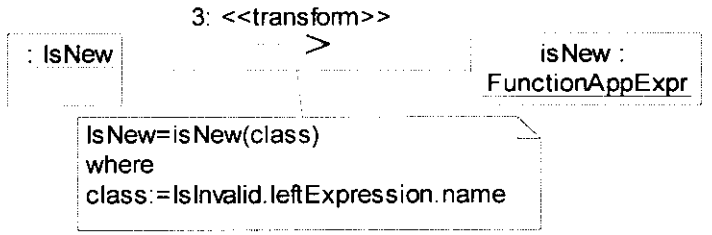


Figure 4.4.69: isNew operation to PVS function application

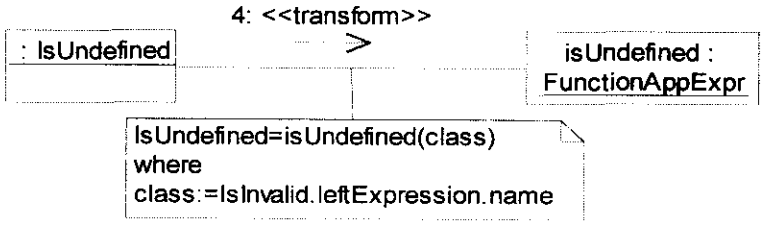


Figure 4.4.70: isUndefined operation to PVS function application

Figure 4.4.8, 4.4.9 and 4.4.10 are the design for transformation of isInvalid, isNew and isUndefined operation respectively. They are transformed into PVS function application. Three functions will also be created in OCLAny theory for isNew, isInvalid and isUndefined where these functions will return results of comparing the argument with new, invalid or undefined constant. invalid and undefined constant represent OCL VoidType and InvalidType as shown in Figure 4.4.6 and 4.4.7. new, invalid and undefined constant is of type any; that represent OCL AnyType (as shown in Figure 4.4.5).

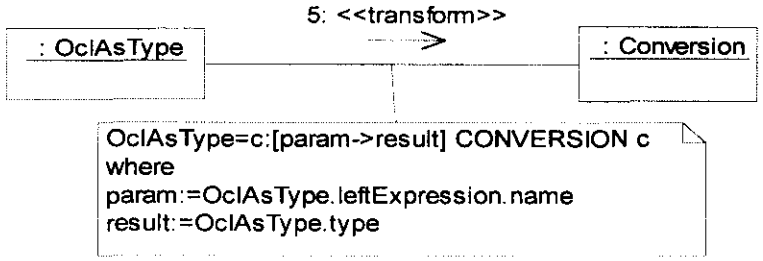


Figure 4.4.71: oclAsType operation to PVS conversion

oclAsType operation is transformed into PVS conversion. An element is added to represent PVS conversion declaration in PVSDeclaration package. Conversion is a function, created to be used by PVS type checker for casting from one type to the other. In the PVS metamodel conversion has an attribute of type UninterpretedConstantDecl to represent the function used in the conversion.

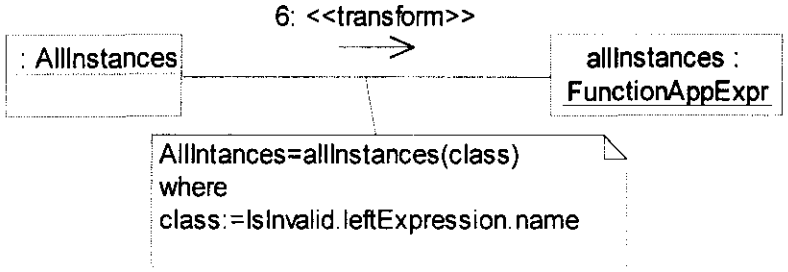


Figure 4.4.72: allInstances operation to PVS function application

To transform allInstances operation, a theory to represent a heap must be created where inside this theory a bag is created to store all instance of every class. A function called

allInstances will return this bag. Each call to OCL allInstances function will be transformed into function application of the created allInstances function.

oclIsKindOf and oclIsTypeOf operation cannot be transform into PVS. There are no PVS language construct that matches oclIsKindOf and oclIsTypeOf operations. Type checking in PVS is done automatically by the theorem prover. OCLType type is not transformed because it is not needed. OCLType represents all classifiers and data types that can be used in OCL type expression. However OCL type expression had already been transform in the second cycle.

4.4.3 Evaluation

An interesting transformation in this cycle is the transformation of OCL Invalid and Void type. OCL Void only has one instance which is undefined or null. One way of representing undefined in PVS is to create an uninterpreted constant called undefined for all type expression created. This option requires a lot of transformation and results in a PVS theory that have a lot of constant called undefined.

A more suitable solution is to create a new theory called OCLAny. In this theory a new type called *any* is created. Three constants of type any called undefined, invalid and new are created to represent OCL undefined, invalid and new. Invalid represent the only instance in OCL Invalid. All theories created for each UML class will import OCLAny theory and all types created will be a subtype of any. This solution is similar to the concept of object hierarchy in Java. In Java, all user defined classes inherits from Java Object class. Java Object class can have a null value and this means that all user defined classes can also have the value of null. For this solution, changes have to be made to the transformation of UML class, association and association end.

In OCLAny theory another type called VOID is created to represent situation where a function does not return anything. A constant of type VOID is created for it to be used in a function. VOID type is needed because a PVS function must return something even though it represents class operation that returns nothing.

The solution that we identified for OCL Void can be used in solving the problem in transforming OCL iterate expression as PVS recursive function. The problem that was mention in the third cycle can be solve by purposely adding undefined constant at the end of every sets, sequences or bags created. In the recursive function, undefined will be used to detect the end of a set, sequence or bag and this will stop the recursive call. To implement this solution, transformation of OCL navigation call expression need to be change by adding codes that add undefined constant at the end of every collection.

The fifth cycle is not implemented because of insufficient time. This project has successfully transformed where possible OCL expression to PVS. This chapter discussed the successful and unsuccessful transformation, and the changes made to both OCL metamodel and PVS metamodel used in this project.

Next Chapter: Testing the transformation and project evaluation

5. TESTING & EVALUATION

The purpose of Chapter 5 is to evaluate and discuss the achievements and findings of this project. It is divided into two main sections: the testing of the transformation program and the evaluation of the entire project.

This chapter will first focus on the testing of the transformations. The testing process will be discussed first, followed by the presentation of testing model and the result of the testing.

The second part of this chapter will talk about the evaluation of the whole project. The evaluation will cover the interesting findings that are found during the design and implementation phase. Non-functional requirements and hypotheses that are identified in Chapter 3 and Chapter 2 respectively will be revisited. Discussion will also be on how well the project meets its non-functional requirements and the status of the hypotheses.

Some of the points discussed may have been discussed in earlier chapters but we remunerate them again in this chapter to make the evaluation section more complete, to highlight the importance of the points and to make this report much easier to read by compiling all the previous discussion.

5.1 Testing

In software development, all project artefacts need to be verified. Requirements are verified through reviews or formal methods, designs are verified using model checking, static analysis, or formal methods and programs are verified using various techniques. Like any other software project artefacts, transformation program also need to be tested. Although the transformation in this project is not part of a software development project, the need for testing still applies which is to convince the user that the artefact being tested held certain properties. Another reason is seen from a research point of view. The technique (model to model transformation) used in this project needs to be tested in order for the research to be complete.

To test the transformation, the result of the transformation will be compared against a source model. The comparisons are done using Epsilon Comparison Language (ECL) programs. The testing process starts by changing the UML class model and OCL constraints for the UML class model, that conforms to the UML and OCL metamodel used in this project, to Eclipse Modelling Framework (EMF) or Metadata Repository (MDR) models. The models are going to be transformed into a PVS model that conforms to the PVS metamodel, which was created for this project, using transformation program that is created in the implementation phase. An ECL program, which is also created during the implementation phase, will compare the PVS model with UML and OCL model.

By comparing the PVS model with the source model, similarities between what is in the OCL expression and the PVS expression can be checked. Although the expressions are different, the values, variables or constants used in both expressions should be the same in order to have a semantically equivalent expression. This is what being test by comparing the source model and the result model.

The comparison is not sufficient to ensure the transformation is correct. Although the target model may create PVS specification that is semantically equal to the source model, the PVS specification may be grammatically incorrect. So, as part of testing the transformation, the PVS model will be serialised to PVS specification and run through PVS theorem prover. The result of the transformation need to be typed checked by PVS because of requirement FR03.

The testing model used in the testing is taken from a case study in [62]. The case study is about a company called Royal and Loyal (R&L) that provides loyalty programs to their clients that offers bonuses to their customers. Figure 5.1.1 is UML model for R&L case study while Listing 5.1.1 is the OCL constraint for the model. The model and OCL constraints are modified to make them more suitable for our testing. OCL constraints that were unable to be transformed will not be tested.

Figure 5.1.1 and Listing 5.1.1 only cover the transformation in Cycle 2 and 3. For Cycle 4, testing will use a different model which is shown in Figure 5.1.2 and the OCL constraints in Listing 5.1.2.

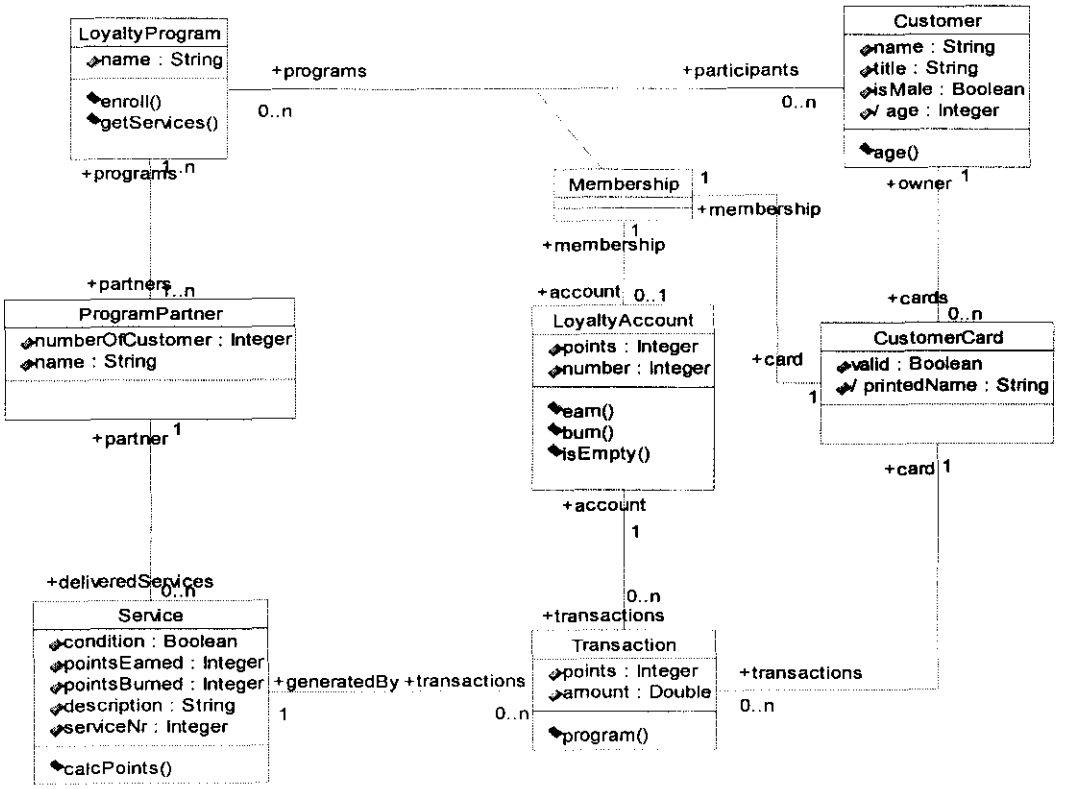


Figure 5.1.73: UML model for R&L case study [62]

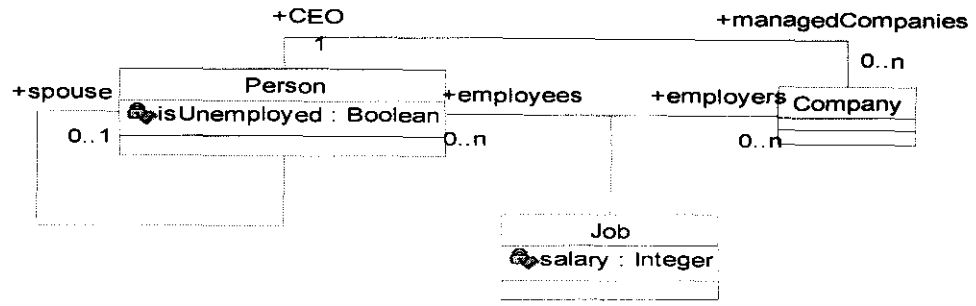


Figure 5.1.74: UML model for testing transformation in fourth cycle [62]

```
context LoyaltyAccount::points
init:0

context LoyaltyProgram::getServices():Set(Service)
body: partners.deliveredServices

context Customer
inv: self.age>=18
inv: Membership.account->select(points>0)

context LoyaltyAccount::isEmpty():Boolean
post: result=(points=0)

context Transaction::getProgram:LoyaltyProgram
post: result=self.card.Membership.programs

context CustomerCard::valid
init: true

context CustomerCard::printedName
derive: owner.name

context LoyaltyAccount::burn(i:Integer)
pre: points>0
post: points=points@pre-i

context LoyaltyProgram
inv: participants-> forAll(age()<=70)
inv: self.Membership.account->one(number<10000)

context LoyaltyAccount
inv: points>0 implies transaction->exists(t|t.points>0)
```

Listing 5.1.1: OCL constraints for R&L case study

```
context Person
inv:    let income: Integer = self.job.salary
        in
          if isUnemployed then
                income < 100
          else
                income >= 100
          elseif
```

Listing 5.1.2: OCL constraint for testing transformation in fourth cycle

The testing process has two results. The first result is the result of comparing source and target model using ECL. The second result is of comparing the PVS specification generated from the target model with the expected PVS specification. Since the generation of PVS specification from the result model is beyond the scope of this project, the PVS specification will be created manually along with the expected PVS specification. The comparison of PVS specification will be done using human observation because the difference between both specifications is explicit enough to be easily detected by human eye. Although human observation have been proved to be sometimes unreliable, time limitations have constraint our effort to research or develop tools to compare the expected and actual PVS specification.

As mentioned in Chapter 3, the results of comparing source model and target model can be categorised into four groups. The result can be either matching and conforming, matching but not conforming, not matching but belongs to domain of comparison, and not matching and not belong to domain of comparison. Table 5.1.1 is the result of the comparison.

The result that we look for is matching and conforming. Matching but not conforming means the transformation is incorrect, not matching and not belonging to transformation domain means the transformation is incomplete [16]. Not matching but belonging to the transformation domain can be interpreted differently depending on where the element belongs to. If it belongs to the target model, it means the transformation is incomplete, but if it belongs to the source model, it means the transformation is erroneous [16].

Table 5.1.17: Comparison result using ECL

Source model element	Result model element	Categorisation
Class Customer	Record type of subtype any with the name Customer	Matching and conforming
Class LoyaltyProgram	Record type of subtype any with the name LoyaltyProgram	Matching and conforming
Class ProgramPartner	Record type of subtype any with the name ProgramPartner	Matching and conforming
Class Service	Record type of subtype any with the name Service	Matching and conforming
Class Transaction	Record type of subtype any with the name Transaction	Matching and conforming
Class CustomerCard	Record type of subtype any with the name CustomerCard	Matching and conforming
Class Membership	Record type of subtype any with the same name	Matching and conforming
Class LoyaltyAccount	Record type of subtype any with the name Membership	Matching and conforming
Class Person	Record type of subtype any with the name Person	Matching and conforming
Class Company	Record type of subtype any with the name Company	Matching and conforming
Class Job	Record type of subtype any with the name Job	Matching and conforming
Properties of Customer	Element of record type with the same name and type	Matching and conforming
Properties of LoyaltyProgram	Element of record type with the same name and type	Matching and conforming

Properties of ProgramPartner	Element of record type with the same name and type	Matching and conforming
Properties of Service	Element of record type with the same name and type	Matching and conforming
Properties of Transaction	Element of record type with the same name and type	Matching and conforming
Properties of CustomerCard	Element of record type with the same name and type	Matching and conforming
Properties of LoyaltyAccount	Element of record type with the same name and type	Matching and conforming
Properties of Company	Element of record type with the same name and type	Matching and conforming
Properties of Person	Element of record type with the same name and type	Matching and conforming
Properties of Job	Variable with the name salary of type int	Matching and conforming
Operations of Customer	Constant call age of type int	Matching and conforming
Operations of LoyaltyProgram	Constant call getServices of type set and a function type call enrol	Matching and conforming
Operations of Service	Constant call calcPoints of type int	Matching and conforming
Operations of Transaction	Constant call program of type LoyaltyProgram	Matching and conforming
Operations of LoyaltyAccount	Constant call isEmpty and 2 functions call earn and burn	Matching and conforming
Initial clause for valid	Override expression that overrides valid element	Matching and conforming
Initial clause for points	Override expression that overrides points element	Matching and conforming
Body clause for getServices()	Expression	Matching and conforming
Post condition for isEmpty()	Axiom	Matching and conforming
Post condition for getProgram()	Axiom	Matching and conforming
Post condition for burn()	Axiom	Matching and conforming
Pre condition for burn()	Condition in IF expression	Matching and conforming
Invariants for LoyaltyAccount	Assumption in LoyaltyAccount theory	Matching and conforming
Invariants for LoyaltyProgram	Assumption in LoyaltyProgram theory	Matching and conforming
Invariants for Customer	Assumption in LoyaltyCustomer theory	Matching and conforming
Invariants for Person	Assumption in Person theory	Matching and conforming
All navigation call expression	Sets or bags	Matching and conforming
All property call expression	Record accessor expression	Matching and

		conforming
All operation call expression	Function application expression	Matching and conforming
Forall operation	Binding expression	Matching and conforming
Exists operation	Binding expression	Matching and conforming
One operation	Binding expression	Matching and conforming
Select operation	Filter function is Set library	Matching and conforming
All Boolean expression	Boolean expression	Matching and conforming
All Equality operation	Boolean expression	Matching and conforming
All Real operator	Numeric expression	Matching and conforming
If expression	IF expression	Matching and conforming
Let expression	LET expression	Matching and conforming

Listing 5.1.3 and 5.1.4 are the expected results that are created manually and Listing 5.1.5 and 5.1.6 are the actual result, also created manually based on the PVS model generated from the transformation. The expected and actual PVS specification are created manually because the transformation done in this project will only create a model of PVS specification. Model-to-text transformation from the PVS model to PVS specification falls outside the scope of this project.

From the PVS specification, one can see that although the result of comparison are matching and conforming for all transformation, the specification that is generated from the PVS model is not what is expected.

The difference between the actual and the expected PVS specification are 1) assumptions and records are not created in the PVS assuming block 2) OCL constraints are transformed into PVS language construct outside the theory and 3) related theories are not imported. One of the reasons why the expected and actual PVS specification is not the same is each transformation of one element of the source model to element of target model is done in separate rules. For example, the rule to transform invariants to assumption is done in Invariant2Assumption rule and derive clause is transform into override expression in Derive2Override rule. In both this rule and in some other rules, there is no link between the created PVS language construct with the theory that the construct supposed to belong to.

```
LoyaltyAccount:THEORY
BEGIN
        ASSUMING
                IMPORTING OCLAny, Transaction
                LoyaltyAccount:TYPE FROM any = [# points:int, number:int,
                        transactions:Transaction,
                membership:Membership #]
                loyaltyaccount@pre:LoyaltyAccount
                loyaltyaccount:LoyaltyAccount

                EARN:TYPE=[ int -> VOID ]
                BURN:TYPE=[ int -> VOID ]
                isEmpty:bool

                LoyaltyAccount-invariant:ASSUMPTION loyaltyaccount'points>0 IMPLIES (EXISTS
                        (loyaltyaccount'transactions):loyaltyaccount'points>0)
        ENDASSUMING

        loyaltyaccount=loyaltyaccount@pre WITH ['points:=0]
        isEmpty-post:AXIOM FORALL loyaltyaccount: isEmpty=(loyaltyaccount'points=0)
        burn:BURN=IF loyalty'account>0 THEN void ELSE void
        burn-post:AXIOM FORALL (i:int):loyaltyaccount'points=loyaltyaccount@pre'points-i
END LoyaltyAccount
LoyaltyProgram:THEORY
BEGIN
        ASSUMING
                IMPORTING OCLAny, ProgramPartner, Customer, Services
                LoyaltyProgram:TYPE FROM any = [# name:String, partners:ProgramPartner,
                        participants:Customer, Membership:Membership #]
                loyaltyprogram@pre:LoyaltyProgram
                loyaltyprogram:LoyaltyProgram

                ENROLL:TYPE = [Customer -> VOID]
                getServices:setof[Sevices]

                LoyaltyProgram-invariant:ASSUMPTION (FORALL (loyaltyprogram'participants) :
                loyaltyprogram'participants'age<=70) AND (exist1! (loyaltyprogram'Membership'account) :
                loyaltyprogram'Membership'account'number<10000)
        ENDASSUMING
END LoyaltyProgram
CustomerCard:THEORY
BEGIN
        ASSUMING
                IMPORTING OCLAny, Customer, Transaction
                CustomerCard:TYPE FROM any = [# valid:bool, printedName:String, owner:Customer,
                        transactions:Transaction #]
                customercard@pre:CustomerCard
                customercard:CustomerCard
        ENDASSUMING
        customercard=customercard@pre WITH ['valid:=true]
        customercard=customercard@pre WITH ['printedName:=customercard@pre'owener'name]
END CustomerCard
```

Listing 5.1.3: Expected PVS specification

To overcome this, some of the transformations need to be done as a batch process. Instead of creating individual rule for each transformation, one rule will be created that transform multiple elements of the source model to multiple elements of the PVS model. The elements that will be part of the batch process are class, properties, operations association ends, invariants, pre

condition, post condition, initial clause and derive clause. The transformation of class, properties, operations and association ends has already been transform as a batch process and the transformation for invariants, pre condition, post condition, initial clause and derive clause will be added to the existing rule. The shortcoming of creating such a big rule is the increase in complexity of the rule. The rule is difficult to design, build and understand.

```
Customer:THEORY
BEGIN
        ASSUMING
                IMPORTING OCLAny, LoyaltyProgram, CustomerCard
                Customer : TYPE FROM any = [# name:String, title:String, isMale:bool, age:int,
                        programs:LoyaltyProgram, cards,CustomerCard #]
                customer@pre:Customer
                customer:Customer

                age:int

                Customer-invariant : ASSUMPTION customer'age>=18
        ENDASSUMING
END Customer
Person:THEORY
BEGIN
        ASSUMING
                IMPORTING OCLAny, Job, Company
                Person:TYPE FROM any = [# isUnemployed:bool, spouse:Person, Job:Job,
                        employers:Company #]
                person@pre:Person
                person:Person

                Person-invariant:ASSUMPTION LET income:int = person'job'salary IN
                        IF person'isUnemployed THEN income<100
                        ELSE income>=100
                        ENDIF
        ENDASSUMING
 END Person
```

Listing 5.1.4: Expected PVS specification (continued)

Another reason why there are differences as mentioned above is the transformation involved two source model, UML model and OCL model. Both models are imported from different CASE tools such as Rational Rose for UML and Octopus 2.0 for OCL. Thus there is no integration between UML and OCL model. Although the relationship between OCL and UML metamodel exist in the source metamodel used in this project, it does not mean the source model is related. For example, the initial clause for *valid* property own by CustomerCard class will be transform into a override expression, but the transformation program will not know the CustomerCard in the OCL expression is the same as the CustomerrCard class in the UML model and the derive clause is for the property own by CustomerCard class. Since the transformation is done at different time and in different rule, the transformation of the derive clause will not be part of the CustomerCard theory (generated by transforming CustomerCard class).

```
Customer:THEORY
BEGIN
        Customer : TYPE FROM any = [# name:String, title:String, isMale:bool, age:int,
                programs:LoyaltyProgram, cards,CustomerCard #]
        customer@pre:Customer
        customer:Customer

        age:int
END Customer
Customer-invariant : ASSUMPTION customer'age>=18

LoyaltyProgram:THEORY
BEGIN
        LoyaltyProgram:TYPE FROM any = [# name:String, partners:ProgramPartner,
                participants:Customer, Membership:Membership #]
        loyaltyprogram@pre:LoyaltyProgram
        loyaltyprogram:LoyaltyProgram

        ENROLL:TYPE = [Customer -> VOID]
        getServices:setof[Sevices]

END LoyaltyProgram
LoyaltyProgram-invariant:ASSUMPTION (FORALL (loyaltyprogram'participants) :
loyaltyprogram'participants'age<=70) AND (exist1! (loyaltyprogram'Membership'account) :
loyaltyprogram'Membership'account'number<10000)

CustomerCard:THEORY
BEGIN
        CustomerCard:TYPE FROM any = [# valid:bool, printedName:String, owner:Customer,
                transactions:Transaction #]
        customercard@pre:CustomerCard
        customercard:CustomerCard

END CustomerCard
customercard=customercard@pre WITH ['valid:=true]
customercard=customercard@pre WITH ['printedName:=customercard@pre'owener'name]

Person:THEORY
BEGIN
        Person:TYPE FROM any = [# isUnemployed:bool, spouse:Person, Job:Job,
                employers:Company #]
        person@pre:Person
        person:Person

END Person
Person-invariant:ASSUMPTION LET income:int = person'job'salary IN
        IF person'isUnemployed THEN income<100
        ELSE income>=100
        ENDIF
```

Listing 5.1.5: Actual transformation result

```
LoyaltyAccount:THEORY
BEGIN
        LoyaltyAccount:TYPE FROM any = [# points:int, number:int,
                transactions:Transaction, membership:Membership #]
        loyaltyaccount@pre:LoyaltyAccount
        loyaltyaccount:LoyaltyAccount

        EARN:TYPE=[ int -> VOID ]
        BURN:TYPE=[ int -> VOID ]
        isEmpty:bool
END LoyaltyAccount
LoyaltyAccount-invariant:ASSUMPTION loyaltyaccount'points>0 IMPLIES (EXISTS
        (t:loyaltyaccount'transactions):t'points>0)
loyaltyaccount=loyaltyaccount@pre WITH ['points:=0]
isEmpty-post:AXIOM FORALL loyaltyaccount: isEmpty=(loyaltyaccount'points=0)
burn:BURN=IF loyalty'account>0 THEN void ELSE void
burn-post:AXIOM FORALL (i:int):loyaltyaccount'points=loyaltyaccount@pre'points-i
```

Listing 5.1.6: Actual transformation result (continued)

One solution for this problem is to merge a UML model and OCL model into a single model that conforms to the metamodel used in this project. Both models could be merged using merging language such as Epsilon Merging Language (EML) [17]. Merging the source model changes our process to prove the properties of OCL constraints that we proposed in Chapter 1 and changed in Chapter 3. Figure 5.1.3 includes merging OCL and UML model in our process.

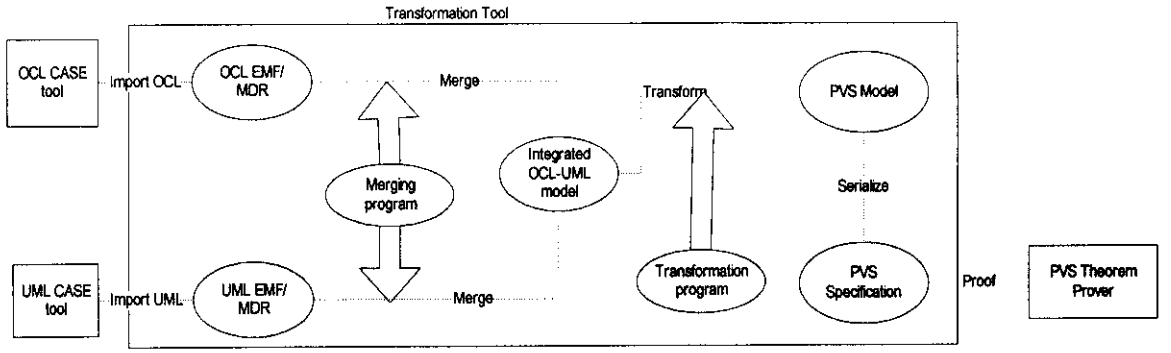


Figure 5.1.75: Process to prove the properties in OCL constraints

5.1.1 Non-Functional Requirement

In chapter 3, four non-functional requirements have been identified: correctness, traceability, simplicity and determinism. This subsection will discuss results of testing for non-functional requirements.

From the testing that has been done for functional requirements, the PVS specification that is created based on the result of the transformation is incorrect. Individual PVS expression created from the transformation are correct but the whole specification is incorrect because there is a possibility that expressions are created outside of the theory.

The incorrectness of the result can be solved using a single transformation rule that transforms multiple source elements as discussed in Section 5.1. This solution is complex and traceability from target to source element will be more difficult. Traceability is much easier if transformation is done using small transformation rule, one for each element of the source model. Since not all transformation will be done in the same transformation rule and only a few transformations are done this way, the project still meets the traceability requirement. Another way to ensure traceability is by using model merging (proposed in [18]) in order to create target model with annotated traceability information. The annotated model would be created by merging result model with traceability model.

The transformation rules created are deterministic which means the transformation will always create the same target model for the same source model. Simplicity is the most difficult characteristic to achieve. Most of the OCL expression can be transformed straightforwardly to PVS expression and the resulting expression is easy to prove. Problem in proving may arise because of complex navigation call expression that will be transformed into nested record accessor expression. For example axioms or binding expressions that have *loyaltyprogram'Membership'account'number* expression will require PVS to resolve its type. Deeply nested binding expressions are also difficult to reason about automatically and this can be generated from the transformation if the OCL expression is also deeply nested.

Both causes of difficulties mentioned above cannot be avoided unless the OCL expression is simple. Thus, it is encouraged that the OCL expression used as the source of the transformation is kept as simple as possible. OCL expression can be kept simple by carefully choosing the context of the constraint, which is one of the best practices in writing OCL constraint.

5.2 Evaluation

Section 5.2 will evaluate what the project has achieved until now. There are five topics that will be discussed: the status of the hypothesis, evaluation on the design notation for transformation, enhancement to OCL metamodel, interesting findings in transformation, the suitability of methodology and tools, and evaluation of the testing process.

5.2.1 Hypotheses

In an earlier chapter, two hypotheses have been identified: 1) There will be no problem in transforming OCL to PVS caused by different types of logic used by OCL and PVS, and 2) OCL can be represented in property-based PVS specifications.

For the first hypothesis, there is no problem with transforming OCL to PVS although OCL used First-Order logic (FOL) while PVS used Higher-Order logic (HOL). This is because logic used in PVS is sufficient to represent logic in OCL.

The second hypothesis is proved to be neither true nor false. It is more natural to transform OCL to model-based PVS specification and that is what we have done, but that does not mean property-based specification cannot be generated from PVS.

However, we have concluded that the property of the system that exists in the OCL and UML diagrams is not explicit enough for the transformation process to identify and generate a

property-based specification. Property-based specification may be generated from OCL and UML diagrams through other techniques but not using the technique used in this project.

5.2.2 Transformation design notation

For this project, a design notation was created to design the transformation. The design notation and metamodel was introduced in Chapter 3. But during the design and implementation phase, some changes have been made to the notation and metamodel. Instead of using class diagram notation, collaboration diagram notation is chosen because transformation applied not to the element of the metamodel but to the instances of the element.

The change is also made because of tool limitations. The design and metamodel is created with the help of Rational Rose. Objects are used in many UML diagrams but in Rational Rose it can only be created in UML dynamic diagrams.

However collaboration diagram notation is not the most suitable notation for our design because it lacks notation for some of the element in the metamodel. In Rational Rose links between object in collaboration diagrams cannot have arrows and stereotypes to represent the direction of the transformation and the type of link respectively. Some transformation uses the association of source model elements and target model elements and these elements should be grouped into source group and target group.

With these limitations, one has to wonder why we use Rational Rose and do not try other UML modelling tools. The reason is that, 1) Rational Rose models can be imported to Eclipse EMF project and 2) Rational Rose is widely available in the department.

For this project the notation will not be developed further as it is not the objective. The metamodel is shown in Figure 5.2.1 and explanations of the elements are in Table 5.2.1.

Table 5.2.18: Description of metamodel elements

Metamodel Elements	Description
SourceMetamodel	Metamodel for the source element
ResultMetamodel	Metamodel for the result element
SourceElement	Element that will be transform. Must be instance of element in SourceMetamodel. Name can be empty
ResultElement	Element that results from the transformation. Must be instance of element in ResultMetamodel. Name can be empty
SourceGroup	A group of source elements that related to a transformation
ResultGroup	A group of result elements that is a result of the transformation
Link	Abstract class that is the parent for RelationLink and TransformLink
TransformLink	Link that represent the transformation of instance of source metamodel to instance of result metamodel or from group of instances.
RelationLink	Abstract class that represent the relation among instances in the source group or instance in the result group
SourceRelation	Relation for instances in the source group. Stereotype is a string that show the type of relationship
ResultRelation	Relation for instances in the result group. Stereotype is a string that

	show the type of relationship
TransformStatement	Abstract class representing statements that explain the transformation
MappingStatement	Statement that explain the mapping of source to result
GuardStatement	Optional statement that represent the condition of a transformation
ActionStatement	Optional statement that represent additional task other than the transformation in the mapping statement
Condition	Condition on a link that must be true in order for a source element to be transform to a result element
Source	Parent class for SourceGroup and SourceElement
Result	Parent class for ResultGroup and ResultElement

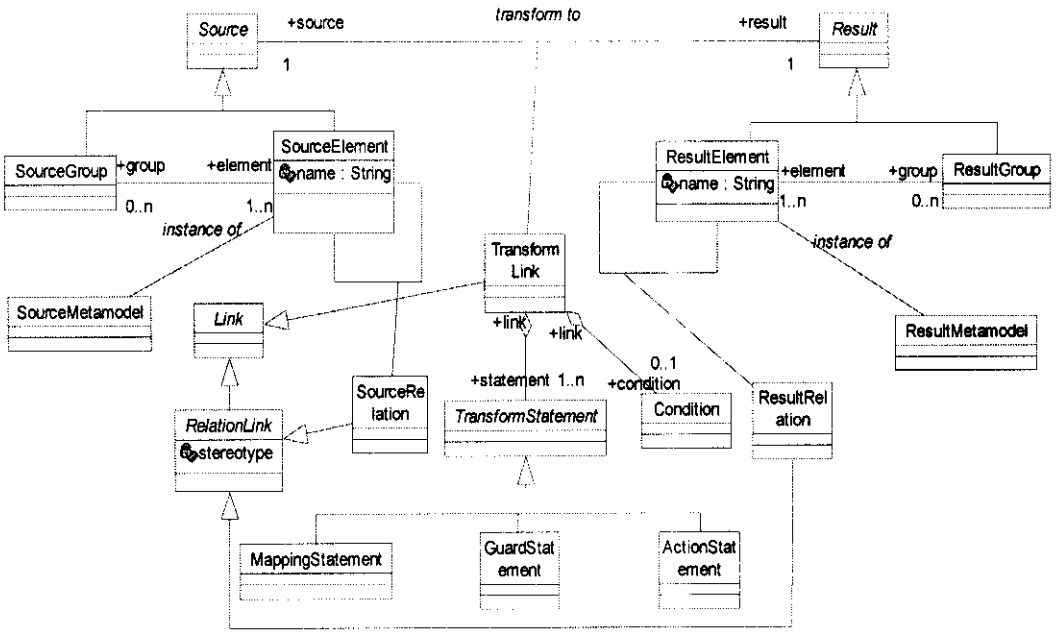


Figure 5.2.76: Design metamodel

From the experience in designing the transformations prior to writing transformation programs, a conclusion can be made that the designs are very helpful. It helps in identifying 1) the incompleteness of source and target metamodel, 2) the condition for the transformation, and 3) the mapping from source element to result element. The design will also give a hint to users on the complexity of the transformation and help the user to choose the best transformation base on the design.

5.2.3 OCL metamodel

This project chose to use the metamodel provided by OMG in order to make the project results widely applicable. Besides the metamodel provided by OMG, there are two other metamodels for OCL which has been proposed in [61], and in [62].

When implementing the transformation, we realised that the metamodel provided by OMG is not sufficient to be used in the transformation. This is not a surprise because the metamodel is intended to explain the abstract syntax of OCL and not to be used in transformation.

Elements that are lacking in the metamodel are the connection between OCL metamodel and UML metamodel. In this project, a subset of Class diagram metamodel is taken from OMG UML specification document. A relationship is created from Constraint element in UML metamodel to OCLExpression element from OCL metamodel. This relationship means a UML constraint consists of OCL expression.

In their metamodel, Richters and Gogolla [61] also make the same relationship but not directly. Constraint (from package Core) consists of a Boolean expression (from package Data Types) that is written in OCL expression (from package OCL). In [62] a simpler approach is taken where OCL expression is directly associated with elements of UML class metamodel that is the context of the OCL expression. Compare to how it is done in [61] and [62], the solution in this project is simpler and does not change the UML and OCL metamodel taken from OMG.

Other changes that are done in OCL metamodel are the addition of elements that represent standard library operations in OCL. OCL has a lot of operations for each type and it is represented as a hierarchy of elements as shown in Figure 5.2.2, 4.3.6, 4.3.7, 4.3.8, 4.3.9 and 4.4.3. From Figure 5.2.2, operations from certain types such as operations for Integer, Boolean and Real are better represented as enumerations.

Expressions that fall under Iterator expression are also being represented as sub-elements of IteratorExpression element. The hierarchy of iterator expression is shown in Figure 4.3.15. This representation of iterator expression is also the same as the representation used in [61]. Another important change that is also taken from [61] is representing @pre as an attribute of type Boolean in PropertyCallExpression.

From the changes that are made to the OCL metamodel, a conclusion can be made that an abstract syntax sometimes is not sufficient to be used as a model for transformation. Transformation requires more detail while abstract syntax only represents abstractly the language construct of a language.

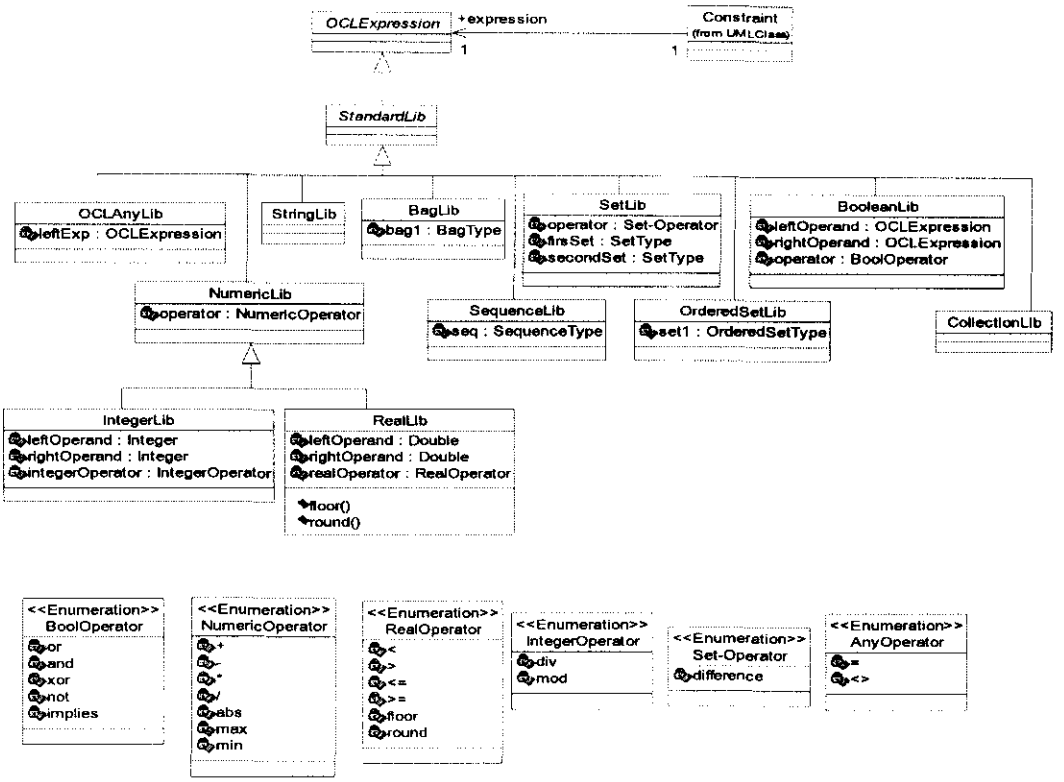


Figure 5.2.77: Metamodel for OCL standard library operation

5.2.4 Summary of transformation

This project has sufficiently transformed OCL expression to PVS. As discussed in Section 3, elements from UML class model are transformed into PVS language construct. Parts of OCL metamodel that are intentionally not transformed because of time constraints are OCL state expression (FR02.1), message expression (FR02.2), message type (FR01.14) and element type (FR01.15).

Other OCL expressions have been successfully transformed into PVS expressions except for iterate expression, iterator expressions (any and isUnique), and some standard library operations for String, Collection, Set, Sequence, Ordered Set and Bag type. Table 5.2.2 is a summary of the successful transformation. The most difficult transformation is the transformation of post conditions. It is difficult because axioms that represent the post conditions include a function application expression that calls the function created for preconditions and body clause.

Table 5.2.19: Successful transformation

UML/OCL elements	PVS elements
Class	Record
Property	Element of the class record
Operation	Function type, constant or variable
Derive clause	Override expression

Initial clause	Override expression
Body clause	Expression depending on the expression in OCL
Pre condition	Condition in IF expression
Post condition	Axiom
Property call expression	Record accessor expression
Navigation call expression	Record accessor, set, bag or sequence
Operation call expression	Function application expression
Variable expression	Expression that uses the variable
Type expression	Expression that is the name of a type
Numeric, Integer, Real literal and operation	Numeric expression
Boolean operation	Boolean expression
Void type	Constant called undefined of type any
Invalid type	Constant called invalid of type any
Any type	A theory with a new type called any
Set type	Set
Ordered set and sequence type	Sequence
Bag type	Bag
If expression	IF expression
Let expression	LET expression
Iterator expression (forall, exists and one)	Binding expression

Iterate expression is transformed into a recursion function but there exist a problem with termination condition of the recursion function. Iterate expression will iterate through all the elements in a collection. To transform this into PVS, a function is needed where it returns the size of the collection or it checks if the current element is the last element. This function will be used in terminating the recursion. Since there is no such function for Set, Sequence or Bag, transformation of iterate expression to recursion function is incomplete.

As mentioned earlier, some of the standard operation for collection and OCL any type fail to be transform because there is no similar functions in PVS. In PVS, sets, sequences and bags is a function but the type of domain and range for the function is different for set, sequence and bag. Set is a function from some type to Boolean, sequence is a function from natural number to some type and bag is a function from some type to natural number. Because of this difference, function supported by each type is different.

Different from PVS, OCL sets, sequence, ordered sets and bags are a subtype of collection type and thus it must be the same except for special characteristic of each type. For example all elements in a set must be unique and sequence is an ordered bag. Because the basic of each type is the same, some operation can be use with all types of collection. Although this feature makes it easier for OCL, it makes it more difficult to get the exact match between OCL collection types and PVS collection types. Thus transforming OCL collection to PVS collection is also difficult especially for operations that can be used for all types of collection. For this purpose, a new library for collections that is similar to the one in OCL is needed.

The new library will have four theories for collection, set, bag and sequence. Each theory will introduce a new type and have functions similar to what is in OCL standard library for collection, set, sequence and bag. Set, sequence and bag type is a subtype of collection. For ordered set, there are two options, 1) create a new theory or 2) use sequence theory because conceptually sequence and ordered set are similar.

By having new libraries that is similar to the collection type in OCL, transformation from OCL collection type to PVS collection type will be straighter and complete.

It will also be very interesting to compare our solution in representing OCL undefined value with the solution taken in [50] and [51]. [50] translates OCL undefined value in various ways:

- Restricting the domain of partial function so that it becomes a total function
- Recursive function must have termination proof and a ranking function
- Only use PVS FORALL to represent the use of OCL forAll and allInstace operation
- Assume all attributes are defined.

The solution proposed in [51] is by creating a *lifted* type for every type created and the lifted type contains the undefined value. Our solution is different from both [50] and [51]. We create a constant of type *any* to represent undefined value. *Any* is a type that represents base type for all PVS types created to represent UML classes.

Another interesting finding in this project is the representation of object-oriented (OO) concept in PVS. Unlike B or Object-Z, PVS does not directly have the concept of object, encapsulation or inheritance that is in B and Object-Z. In transforming OCL to PVS, the importance of transforming UML class model has been identified and justified and thus the OO concept will also need to be represented in PVS. The OO concept and how it is represented in PVS is shown in Table 5.2.3.

Table 5.2.20: Representation of OO concept in PVS

Object-oriented concept	Representation in PVS
Object	Constant of type record
Properties (State)	Element of a record or variable
Behaviour	Function type (behaviour with argument), constant or variable
Class	PVS record type
Encapsulation	Private properties represented as variable, public properties as element of record. Behaviour with arguments as function type, private behaviour without arguments as variable and public behaviour without arguments as constant
Inheritance	1) Subclass will be a subtype of the type that represents the super class. 2) create a record with both the properties from super class and subclass
Polymorphism	Create a constant of type function that is being override

A conclusion can be made that the transformation from OCL to PVS is sufficient to cover the important OO concept and OCL expression but it is still not complete. Parts that are not complete are mostly related to OCL standard libraries for various types (Any, Collection, Set, Sequence, Ordered Set, Bag and String), iterator expressions and iterate expression. The main problem is PVS has a strong typed language and PVS string, set, sequence and bag type although exist but they are just not similar to the one in OCL. PVS functions need to be created for iterator and iterate expressions.

5.2.5 Methodology and tools

The transformation program is developed in cycles using a modified Spiral model. In the first cycle, metamodel for PVS is created and transformation of main OCL expression and element of UML class model is done in the second cycle. Transformation in the third cycle concentrated on OCL collections, while in the fourth cycle transformations were applied to the remaining OCL types and operations. Before the implementation of each transformation cycle, the design and test specification for each transformation is created.

The advantages of using an iterative and evolutionary scheme are 1) the implementation is divided into well planned cycle and each cycle will produce workable deliverables, 2) the project is more flexible in its definition of completeness where the project can stop its implementation in any cycle and declare it complete based on the current situation and 3) the objectives of each cycle is also flexible where at the start of each cycle the objective for that cycle is refined and evaluated to make sure that the objectives can be achieved at the end of each cycle.

Luckily in this project all objectives set for all the cycles had been met and objectives of each cycle that is refined at the beginning of each cycle does not change much than the one that is identified during the requirement phase.

The project also mention about using Test Driven Development (TDD). The benefits of using TDD are the implementation is more controlled as the implementation will only try to pass the test being specified, and the test can be used to verify the design and transformation program. If the design and transformation program does not represent what is being specified in the test specification, it is considered as incorrect.

This project uses Epsilon Transformation Language (ETL) and Epsilon Comparison Language (ECL) that runs in Eclipse. Eclipse is easy to use while ETL and ECL are suitable and sufficient for the transformation of metamodel to metamodel. However a problem is encountered during the comparison of a transformation that results in calling an operation. For example, consider the transformation of the floor operation in OCL. It is transform into floor operation in NumericExpr element in PVS metamodel. ECL cannot compare the result of the transformation because it can't compare call to operations.

But this is not a critical issue because there are other ways in transforming problems like the example given above. Instead of representing PVS floor operation as an operation for NumericExpr, it can be represented as an element in the metamodel. OCL floor operation will be transform into this element and this makes it easier to compare.

5.2.6 Testing process

The testing process that is employed for the transformation in this project is different from traditional testing process that is used in testing computer software. Software is tested starting from individual modules (testing in the small) and move up to testing the whole software (testing in the large).

It starts with unit testing, which is normally done by the programmers of the unit. After all units pass their individual tests, the unit is integrated with other modules and the integration is tested. There are three most common strategies in integrating units of software which is top-

down, bottom-up and big bang. After all the units are integrated and they can function properly as one complete system, system testing is done to check the non-functional property of the software and to find abnormities cause not by functional errors. The final test will be acceptance testing where the software will be test by clients/end user at the developers' site (alpha testing) and at the clients/end users' site (beta testing).

For this project, the transformation was tested by comparing the source and target model using a comparison program. Test will also be done in checking whether the target model can generate a correct PVS specification. Compare to traditional software testing, testing transformation is more of a black box testing than a white box testing because transformations seldom have complex structures that requires white box testing.

From this project it is also realise that it is very difficult to have a unit test on each individual transformation rule and integration test for the integration of transformation rule. Unit testing is if not impossible then it is very difficult because the nature of the test data itself, which in this case is a model. It is just easier to test the transformation as a whole than testing small parts of the model. Testing of individual rules may also lead to the testing of other rules making testing rules in isolation an impossible task.

Integration testing is impossible for this project because the order of transformation is not in the control of the user but in the control of the transformation engine (Epsilon). Without this control the user can never be certain which rule will be tested first. Integration testing is also impossible because there is no structure between transformation rules like the one exists in software. For software, a hierarchy of module can be created that usually starts from the main module and the main module will call other modules. This does not exist in transformation rule and the execution of the transformation is more adhoc and depends on the model being tested.

Testing transformation is more similar to system level testing where testing is done to all the transformation rules at once. However, test to check the property of the transformation is not done quantifiably in this project. For software, quantifiably testing the property of software usually requires the used of tools such as simulators but such tools for testing transformation is still under research.

For this project the testing process is not so difficult. There are plenty of UML and OCL CASE (Computer Aided Software Engineering) tools that can generate source model for testing the transformation. The only task that requires a lot of work in the testing process is creating the comparison program. Comparison rules must be created for each transformation rules and the complexity of comparison rules depends greatly on the transformation rules. Most complex transformation need equally complex comparison program.

Next Chapter: Summary, conclusion and future work.

6 CONCLUSIONS

This project is part of an effort to provide powerful theorem proving support for proving properties about OCL constraints. To do this, the chosen technique is to transform OCL to a formal language using a model transformation language, and to run the result of the transformation in a theorem prover. For this project the PVS specification language was chosen as the formal language.

The transformation rules are written in Epsilon Transformation Language (ETL) using Eclipse as the IDE (Integrated Development Environment). The transformation is done in an iterative and evolutionary scheme fitting the Spiral methodology that was chosen at the beginning of the project. The implementation was divided into 4 cycles (iterations). The first cycle created a basic PVS metamodel that is refined iteratively during consecutive cycles. Transformation of UML models, OCL expressions, constraints and OCL basic types are transformed in the second cycle. Third cycle implements the transformation of OCL iterator expressions, iterate expression, tuple type and collection types. The final cycle is the transformation of OCL Void, Invalid and Any type. There was suppose to be a fifth cycle but due to time limitation transformation in the fifth cycle was not implemented.

A design language was created to assist in designing the transformation. The design language was created by first creating its abstract syntax and concrete syntax. The abstract syntax was created iteratively during the implementation of the transformation. The project chose to use UML collaboration diagram as the concrete syntax because of tool limitations.

During each cycle, transformation rules were tested using two methods; comparison using rules written in Epsilon Comparison Language (ECL) and syntax checking of PVS specification generated from PVS model. All the transformations passed the tests using the first method but there is a possibility of errors when testing using the second method. The errors exist because the PVS model generated from the transformation have the possibility to create PVS specification with syntax errors.

Research been done on topics related to the project, namely model transformation technologies, OCL and UML verification, and formal language. From the studies two hypothesis have been identified. The hypothesis was revisited at the end of the project and the status of the hypothesis was discussed. The project also discussed the status of functional and non-functional requirements, changes made to OCL metamodel, interesting findings from the transformation, and the testing process.

Numerous problems were encountered during the implementation phase. Most of them happened in the third cycle during the transformation of collection types. Part of transforming a type is the transformation of standard library operations for the type. Standard operations for Strings, Set, Ordered Set, Sequence, Bag, and Any type failed to be transformed (see Chapter 4 for the successful/unsuccessful transformation). The main problem is although similar type exists in PVS, it is not completely the same as the one in OCL. The main difference is OCL types, especially collections, support different operations and may have different semantics. A solution has been proposed in Chapter 5 to create a more accurate representation of OCL types in PVS.

Most of the non-functional requirements (NFR) were only partially achieved. The PVS model generated from the transformation is correct but the correctness of the PVS specification

is uncertain. Traceability is difficult for some of the transformation rule because some rules transform multiple elements in the source metamodel to multiple elements in the target metamodel. Simplicity is not guaranteed because it depends on the complexity of the OCL expression being transformed. The used of nested records to represent classes does not help in reducing type checking because it may generate PVS expression that requires type checking. Also the transformation does not use any language constructs provided by PVS to simplify type checking such as judgement or coercion. Determinism is the only NFR that the project met.

A conclusion can be made that PVS is currently not the most suitable formal language to represent OCL. Higher-order logic used in PVS is suitable to represent OCL First-order logic, but what makes PVS non-suitable for representing OCL is because of the incompatible types. Providing libraries that is more similar to types in OCL will result in a more complete representation.

6.1 Future Work

What this project achieved is only part of the work required in verifying OCL using model transformation technique. The transformation rules we have produced can be improved to have a more complete transformation and to create transformation rule that fully meet the NFRs. Two transformation rules that need to be improved are the transformation of UML and OCL constraints on operations. Transformation rule for UML need to include transformation of OCL constraints (initial clause, derive clause, pre condition, post condition, body clause and invariants).

As mentioned earlier, for a complete representation of OCL in PVS, new libraries for String, Collection, Set, Sequence, Bag and OCLAny. These new libraries will have all the operations in OCL standard library for each type. A library will also be created to represent a Heap. The Heap theory will contain a bag that contains all constants created for each type and the bag will be return by the allInstance operation.

Depending on the technique chosen to represent the new libraries in the PVS metamodel, a detailed metamodel of the library will be added to the current PVS metamodel. There are two ways to do this, one is by creating and adding the representation manually and the second choice is by creating separate metamodel for each library and write a merging program that will merge the metamodels.

A merging program will also need to be created to merge the source models. Currently the transformation has two source models, UML model and OCL model. Both models should be merged so that the constraints in OCL model can be related to their context in the UML model.

The final item of future work is the most important. Transformation rule need to be written for the transformation of PVS model (generated from model-to-model transformation) to a PVS specification. These transformation rules can be written in XML Stylesheet Language (XSL) or Model-toText language such as Epsilon Generation Language (EGL) [15] or MOFScript.

REFERENCE

[1] Pressman, R. Software Engineering: A Practitioner's Approach 6th International Edition, McGraw Hill, 2005.

[2] UMLX Subproject, http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/UMLX/index.html, viewed on 20/05/2007.

[3] Czarnecki, K. and Helson, S. Classification of Model Transformation Approaches, Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, USA, 2003.

[4] OMG. MOF QVT Final Adopted Specification, www.omg.org/docs/ptc/05-11-01.pdf, viewed on 13/05/2007.

[5] Jouault, F. and Kurtev, I. On the Architectural Alignment of ATL and QVT, SAC'06, ACM, 2006.

[6] The VIATRA 2 Model Transformation Framework , http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/doc/viatratut_October2006.pdf, viewed on 21/05/2007.

[7] Marcos et. al. AMW: A Generic Model Weaver, Proceedings of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles, 2005.

[8] Marcos, F and Jouault, F. Model Transformation and Weaving in the AMMA Platform, Pre-proceedings of the Generative and Transformational Techniques in Software Engineering (GTTSE 05), Workshop, 2005.

[9] Allilaire, F., Bezivin, J., Bruneliere, H., Jouault, F. Global Model Management in Eclipse GMT/AM3, Eclipse Technology Exchange Workshop (eTX) at ECOOP'06, 2006.

[10] Bezivin et. al. First Experiments with the ATL Model Transformation Language: Transforming XSLT to XQuery, OOPSLA'06 Workshop, 2003.

[11] Jouault, F. Loosely Coupled Traceability for ATL, ECMDA-TW'05 Proceedings, 2005.

[12] Bezivin et. al. The ATL Transformation-based Model Management Framework, http://www.sciences.univ-nantes.fr/info/recherche/Vie/RR/RR-IRIN2003-08.pdf, viewed on 12/05/2007.

[13] Kolovos, D., Paige, R., and Polack, F. Eclipse Development Tools for Epsilon, http://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-

EclipseModelingSymposium9_DevelopmentToolsforEpsilon.pdf, viewed on 14/05/2007.

[14] Kolovos, D., Paige, R., Polack, F and Rose, L. Update Transformation in the Small with the Epsilon Wizard Language, Journal of Object Technology Volume 2, ETH Zurich, 2003.

[15] Kolovos, D., Paige, R., and Polack, F. The Epsilon Object Language (EOL), LNCS: ECMDA-FA'06, Springer, 2006, page 128-142.

[16] Kolovos, D., Paige, R., and Polack, F. Model Comparison: A Foundation for Model Composition and Model Transformation Testing, GaMMa'06, ACM, 2006, page 13-19.

[17] Kolovos, D., Paige, R., and Polack, F. Merging Models with the Epsilon Merging Language (EML), http://www-users.cs.york.ac.uk/~dkolovos/publications/models06-eml.pdf, viewed on 14/05/2007.

[18] Kolovos, D., Paige, R., and Polack, F. On-Demand Merging of Traceability Links with Models, http://modelbased.net/ecmda-traceability/images/papers/2_dkolovos.traceability06.camera-ready.pdf, viewed on 14/05/2007.

[19] Balogh, A. and Varro, D. Advanced Model Transformation Language Constructs in the VIATRA2 Framework, Proceedings of SAC'06, ACM, 2006, page 1280-1287.

[20] VIATRA project. The VIATRA2 Model Transformation Framework (Project Plan for an Eclipse GMT Subproject), http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/doc/Viatra2_ProjectPlan.pdf, viewed on 21/05/2007.

[21] Nipkow, T., Paulson, L., and Wenzel, M. Isabelle/HOL: A Proof Assistant for Higher-Order Logic (LNCS2283-Tutorial), Springer, 2002.

[22] Schneider, S. The B-Method: An Introductuin, Palgrave, 2001.

[23] Diller, A. Z: An Introduction to Formal Methods, John Wiley & Sons, 1990.

[24] Spivey, J. Understanding Z: A specification language and its formal semantics, Cambridge University Press, 1988.

[25] Owre, S., Rushby. and Shankar, N. PVS: A Prototype Verification System, In Proceedings of CADE (LNAI 607), Springer, 1992, page 748-752.

[26] Owre, S. PVS Introduction, http://pvs.csl.sri.com/introduction.shtml, viewed on 30/05/2007.

[27] Schumann, J. Automated Theorem Proving in Software Engineering, Springer, 2001.

[28] Gordon, M. and Melham, T. Introduction to HOL: A theorem proving environment for higher order logic, Cambridge University Press, 1993.

[29] Isabelle's Logics, http://www.cl.cam.ac.uk/research/hvg/Isabelle/logics.html, viewed on 01/06/2007.

[30] Jacky, J. The way of Z: Practical programming with formal methods, Cambridge University Press, 1997.

[31] Potter, B., Sinclair, J. and Till, D. An Introduction to Formal Specification and Z, Prentice Hall, 1991.

[32] Woodcock, J. and Davies, J. Using Z: Specification, Refinement and Proof, Prentice Hall, 1996.

[33] Kim, S. and Carrington, D. A Formal Mapping between UML Models and Object-Z Specifications, ZB 2000: Formal Specification and Development in Z and B (LNCS 1878), Springer Verlag, 2000.

[34] LeDang, H. and Souquieres, J. Contribution for Modelling UML State-Charts in B, Third International Conference on Integrated Formal Methods (LNCS), Springer Verlag, 2002.

[35] Traore, I. An Outline of PVS Semantics for UML Statecharts, Journal of Universal Computer Science, volume 6, Springer, 2000, page 1088-1108.

[36] Traore, I. and Aredo, D. Enhancing Structured Review with Model-based Verification, IEEE Transactions On Software Engineering, volume 30, 2004.

[37] Snook, C. and Butler, M. UML-B: Formal Modelling and Design Aided by UML, ACM Transaction on Software Engineering and Methodology, Volume 5, ACM, 2006.

[38] Simmonds, C. and Bastarrica, J. A Tool for Automatic Model Consistency Checking, ASE'05, ACM, 2005, page 431-432.

[39] Sourrouille, J. and Caplat, G. Constraint Checking in UML Modelling, SEKE'02, ACM, 2002, page 217-224.

[40] Malgouyres, H. and Motet, G. A UML Model Consistency Verification Approach Based on Meta-modelling Formalization, SAC'06, ACM, 2006, page 1804-1809.

[41] Kotb, Y. and Katayama, T. Consistency Checking of UML Model Diagrams Using XML Semantics Approach, WWW 2005, ACM, 2005, page 982 and 983.

[42] Truong, N. and Souquieres, J. Verification of Behavioral Element of UML Models Using B, SAC'05, ACM, 2005, page 1546-1552.

[43] Toval, A., Saez, J., and Maestre, F. Automated Property Verification in UML Models, http://users.ecs.soton.ac.uk/~mal/avocs03/proceedings/02.pdf, viewed on 05/06/2007.

[44] Mostefaoui, F. and Vachon, J. Verification of Aspect-UML Models using Alloy, AOM'07 Workshop, ACM, 2007, page 41-48.

[45] The Alloy Analyzer, http://alloy.mit.edu/index.php, viewed on 11/06/2007.

[46] Eshuis, R. and Wieringa, R. Verification Support for Workflow Design with UML Activity Graphs, ICSE'02, ACM, 2002, page 166-176.

[47] Beato, E., Barrio-Solorzano, M., Cuesta, C. UML Automatic Verification Tool (TABU), SAVCBS'04, ACM, 2004.

[48] Schinz, I., Toben, T., Mrugalla, C. and Westphal, B. The Rhapsody UML Verification Environment, http://www-omega.imag.fr/doc/d1000312_1/WP22-312-V1-ruve2004.pdf, viewed on 05/06/2007.

[49] Gogolla, M., Richters, M. and Bohling, J. Tool Support for Validating UML and OCL Models through Automatic Snapshot Generation, Proceedings of SAICSIT, 2003, page 248-257.

[50] Kyas et al. Formalizing UML Models and OCL Constraints in PVS, www.wisdom.weizmann.ac.il/~kugler/SFEDL04.pdf, viewed on 23/04/2007.

[51] Brucker, A. and Wolff, B. A Proposal for a Formal OCL Semantics in Isabelle/HOL, TPHOLs'02, LNCS 2410, Springer, 2002, page 99-114.

[52] Roe, D., Broda, K. and Russo, A. Mapping UML Models Incorporating OCL Constraints into Object-Z, http://www-ala.doc.ic.ac.uk/research/technicalreports/2003/DTR03-9.pdf, viewed on 05/06/2007.

[53] Marcano, R. and Levy, N. Transformation Rules of OCL Constraints into B Formal Expression, http://www4.in.tum.de/~csduml02/31.pdf, viewed on 18/04/2007.

[54] Marcano, R. and Levy, N. Using B Formal Specification for Analysis and Verification of UML/OCL Models, se2c.uni.lu/tiki/se2c-bib_download.php?id=877, viewed on 18/04/2007.

[55] Ledang, H. and Souquires, J. Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expression into B, APSEC'02, IEEE, 2002.

[56] Distefano, D., Katoen, J. and Rensink, A. Towards Model Checking OCL, http://www.dcs.qmul.ac.uk/~ddino/papers/puml2k.pdf, viewed on 05/05/2007.

[57] OMG. Object Constraint Language: OMG Available Specification Version 2.0 (formal/06-05-01).

[58] Lamari. Towards an Automated Test Generation for the Verification of Model Transformations, SAC'07, ACM, 2007, page 998-1005.

[59] Baudry, B., Fleurey, F. and Steel, J. Validation in Model-Driven Engineering: Testing Model Transformations, http://www.fleurey.com/weblog/attachments/issre04_submission.pdf, viewed on 13/06/2007.

[60] Lin, Y., Zhang, J. and Gray, J. A Testing Framework for Model Transformation, http://www.gray-area.org/Pubs/transformation-testing.pdf, viewed on 13/06/2007.

[61] Richters, M. and Gogolla, M. A Metamodel for OCL, www.db.informatik.uni-bremen.de/publications/Richters_1999_UML.ps.gz, viewed on 14/05/2007.

[62] Warmer, J. and Kleppe, A. The Object Constraint Language: Getting Your Models Ready for MDA, 2nd Edition, Pearson Education, 2003.

[63] Boehm, B. A Spiral Model of Software Development and Enhancement, http://www.sce.carleton.ca/faculty/ajila/4106-5006/Spiral%20Model%20Boehm.pdf, viewed on 22/06/2007.

[64] Owre, S., Shankar, N., Rushby, J. and Stringer-Calvert, D. PVS Language Reference, SRI International, http://pvs.csl.sri.com/doc/pvs-language-reference.pdf, viewed on 21/02/2007.

[65] Bezivin, J. From Object Composition to Model Transformation with the MDA, Proceedings of TOOLS-39, IEEE, 2001.

[66] Object Management Group (OMG). MDA Guide Version 1.0.1 (omg/2003-06-01), OMG, 2003.

[67] Jussila et.al. Model Checking Dynamic and Hierarchical UML State Machines, http://modeva.itee.uq.edu.au/accepted_papers/paper_4_8.pdf, viewed on 06/05/2007.

[68] OMG. Unified Modelling Language: Superstructure Version 2.1.1 (formal/06-05-01).

Appendix A: Description of design features use in categorising transformation language

Design Features		Description
Transformation Rules	LHS/RHS	LHS/RHS of rules. Can represented using combination of variables, patterns or logic.
	LHS/RHS Syntactic Separation	RHS and LHS may or may not be separated
	Bidirectional	Rule may be executed in both direction
	Parameterization	Rules can have parameters to control configuration and tuning
	Intermediate Structures	Some transformation approaches have intermediate structure (no direct transformation between source and target)
Rule Application Scoping		Rules can be applied to certain parts of the model
Relationship between Source & Target		Some language will only allow new model to be created as a result of transformation while others allow in-place transformation or update existing target
Rule-Application Strategy		Transformation rules can be applied using deterministic, non-deterministic or interactive strategy.
Rule Scheduling	Form	Scheduling can be in the form of explicit or implicit. Explicit form can be external or internal where external there is a clear separation of rules and scheduling logic. Internal allow rules to call other rules. Implicit form does not allow human intervention
	Rule Selection	Rule selection can be either explicit condition (guards), non-deterministic or interactive. Conflict resolution algorithm may also be provided
	Rule Iteration	Include recursion, looping or fixpoint
	Phasing	Transformation process may be organised in several phases and scheduling of rule will base on the phases

Rule Organisation	Modularity Mechanism	Allow rules to be group into packages or modules
	Reuse Mechanism	Rules can be reuse using inheritance or logical composition (rules calling other rules)
	Organisational Structure	Rules can be source-oriented, target-oriented or independent. Source means rules are organise base on source model, target base on target model or independent of source or target model.
Tracing	Storage Location	Traceability information can be stored in a separate model, source model or target model
	Control	Traceability can be done manually or automatic
Directionality		Transformation may be unidirectional or bidirectional. Bidirectional can be achieve by providing bidirectional rules or using complementary pairs of unidirectional rules.

Appendix B: UML/OCL translation to B in [54] and [37]

UML/OCL	Marcano & Levy translation [54]
Class	Abstract machine
Class instance	Abstract variable of type set
Attribute	Abstract variable typed as total function. the total function is from class instance to attribute typed
Associations	A variable of type relations between associated classes
Property call expression	Image of total function that represent the attribute
Navigation call expression	Image of binary relation that represent the association
Invariants	Predicates added in INVARIANT section of the abstract machine that represents the context of the invariant. The invariant range of over all instance of the context
Preconditions	Predicates in PRE section of B operation
Post conditions	Predicates in THEN section of B operation

Operations on OCL basic types and collections are map into equivalent B operator or expressions.

UML	Butler & Snook translation [37]
Package	Abstract machine
Class	Variable that is part of a set
Instance	Set
Attribute	Variable of type function from instance to attribute type
Association	Similar to attribute except that the range is the supplier end of the function
Association Multiplicities	See figure 1
Inheritance	Variable that is part of super class
States	Enumerated set of type class
Transition	Operation transit from source state to target state

Association Representations in B for Different Multiplicities		
A *and* B *are the current instances sets of class* A *and* B *respectively.* f *is a function representing the association (i.e. the role name of the association with respect to the source class.* A↣ disjoint (f) *is defined in* B *as:* $\forall i,j \cdot i \in dom(f) \wedge j \in dom(f) \wedge i \neq j \Rightarrow f(i) \cap f(j) = \varnothing$		
UML association multiplicity	Informal description of B representation	B invariant
$0..^{1} \rightarrow 0..1$	partial function to B	$f \in A \rightarrow B$
$0..^{1} \rightarrow 1..1$	total function to B	$f \in A \rightarrow B$
$0..^{*} \rightarrow 0..^{*}$	total function to subsets of B	$f \in A \rightarrow \mathbb{P}(B)$
$0..^{*} \rightarrow 1..^{*}$	total function to non-empty subsets of B	$f \in A \rightarrow \mathbb{P}_1(B)$
$0..1 \rightarrow 0..1$	partial injection to B	$f \in A \rightarrowtail B$
$0..1 \rightarrow 1..1$	total injection to B	$f \in A \rightarrowtail B$
$0..1 \rightarrow 0..^{*}$	total function to subsets of B that don't intersect	$f \in A \rightarrow \mathbb{P}(B) \wedge$ disjoint(f)
$0..1 \rightarrow 1..^{*}$	total function to non-empty subsets of B that don't intersect	$f \in A \rightarrow \mathbb{P}_1(B) \wedge$ disjoint(f)
$1..^{1} \rightarrow 0..1$	partial surjection to B	$f \in A \twoheadrightarrow B$
$1..^{1} \rightarrow 1..1$	total surjection to B	$f \in A \rightarrow B$
$1..^{1} \rightarrow 0..^{*}$	total function to subsets of B that cover B	$f \in A \rightarrow \mathbb{P}(B) \wedge$ union(ran(f))= B
$1..^{1} \rightarrow 1..^{*}$	total function to non-empty subsets of B that cover B	$f \in A \rightarrow \mathbb{P}_1(B) \wedge$ union(ran(f))= B
$1..1 \rightarrow 0..1$	partial bijection to B	$f \in A \rightarrowtail B$
$1..1 \rightarrow 1..1$	total bijection to B	$f \in A \rightarrowtail B$
$1..1 \rightarrow 0..^{*}$	total function to subsets of B that cover B without intersecting	$f \in A \rightarrow \mathbb{P}(B) \wedge$ union(ran(f))= B $\wedge$ disjoint(f)
$1..1 \rightarrow 1..^{*}$	total function to non-empty subsets of B that cover B without intersecting	$f \in A \rightarrow \mathbb{P}_1(B) \wedge$ union(ran(f))= B $\wedge$ disjoint(f)

Figure 1: Translation of association multiplicities [37]

Appendix C: UML/OCL translation to Object-Z in [52]

UML class	Object-Z class with the same name
Attributes	Variable of the same name declared in the state schema.
Constant attributes	Variable of the same name declared in a axiom
Attribute with multiplicities more than 1	A sequence of the same type as the attribute declared in the state schema
Operation	Operation schema with a parameter adorned with '?' and return adorned with '!'
Visibility	Public attributes/operations are included in the class visibility list
Association	A variable declared in the state schema with the same name as the association end or name of the class (if no association end). Association ends with cardinality more than one will create a sequence
Aggregation	Similar to mapping of association with the compound class construct containing a variable of type power set of part class
Composition	A variable declared in state schema with Object-Z notation that denotes unshared containment
Association class	Object-Z class with the same name and variables declared in state schema to represent association ends. Variable of type association class will be added to class in the association
Inheritance	Used Object-Z inheritance notation
Invariants	Predicates in state schema
Pre and post conditions	Predicates in operation schema. All variables in post conditions will be adorned with prime symbol to denote value after change except for variable with @pre.
Result keyword	Replace with return value name
Collection->size()	#Collection
Collection->count(object)	#{c:Collection | c=object}
Collection->includes(object)	object ε Collection
Collection->includesAll(parameter)	parameter Collection
Collection->isEmpty()	Collection=ø
Collection->notEmpry()	Collection<>ø
Collection->sum(feature)	∑c:Collection.c.feature
Collection->select(e|expression)	{e|expression}
Collection->reject(e|expression)	{e|¬expression}
Collection->collect(e|expression)	{e:collection|expression}

Collection->forAll(e|expression)	e:Collection.expression
Collection->exists(e|expression)	e:Collection.expression
OCL initial clause	Class initial schema

OCL basic types (integer, real, and Boolean) are map into its equivalent types in Object-Z and String are map into sequence of characters. Basic type literals may be used freely in Object-Z syntax.

Concrete collection types (set, sequence and bag) are map into its equivalent form in Object-Z. Operations specific to set (union, intersection, minus, symmetricDifference, including, and excluding) and specific operations to sequence (first, last, at, append, and prepend) are map to semantically similar operators or expressions in Object-Z.

Appendix D: OCL Expressions

OCL Expression	Expression Type	Requirement	Cycle
OCLAny::=	Standard Library	FR02.8	4
OCLAny::<>	Standard Library	FR02.8	4
OCLAny::oclIsKindOf()	Standard Library	FR02.8	4
OCLAny::oclIsTypeOf()	Standard Library	FR02.8	4
OCLAny::oclAsType()	Standard Library	FR02.8	4
OCLAny::isNew()	Standard Library	FR02.8	4
OCLAny::isUndefined()	Standard Library	FR02.8	4
OCLAny::isInvalid()	Standard Library	FR02.8	4
OCLAny::allInstances()	Standard Library	FR02.8	4
Number::+	Standard Library	FR02.8	2
Number::-	Standard Library	FR02.8	2
Number::*	Standard Library	FR02.8	2
Number::/	Standard Library	FR02.8	2
Number::abs	Standard Library	FR02.8	2
Number::max	Standard Library	FR02.8	2
Number::min	Standard Library	FR02.8	2
Integer::div	Standard Library	FR02.8	2
Integer::mod	Standard Library	FR02.8	2
Real::floor()	Standard Library	FR02.8	2
Real::round()	Standard Library	FR02.8	2
Real::<	Standard Library	FR02.8	2
Real::>	Standard Library	FR02.8	2
Real::<=	Standard Library	FR02.8	2
Real::>=	Standard Library	FR02.8	2
Boolean::or	Standard Library	FR02.8	2
Boolean::xor	Standard Library	FR02.8	2
Boolean::and	Standard Library	FR02.8	2
Boolean::not	Standard Library	FR02.8	2
Boolean::implies	Standard Library	FR02.8	2
String::size()	Standard Library	FR02.8	2
String::concat()	Standard Library	FR02.8	2
String::substring()	Standard Library	FR02.8	2
String::toInteger()	Standard Library	FR02.8	2
String::toReal()	Standard Library	FR02.8	2
Collection::size()	Standard Library	FR02.8	3
Collection::count()	Standard Library	FR02.8	3
Collection::isEmpty()	Standard Library	FR02.8	3
Collection::notEmpty()	Standard Library	FR02.8	3
Collection::includes()	Standard Library	FR02.8	3
Collection::excludes()	Standard Library	FR02.8	3
Collection::includesAll()	Standard Library	FR02.8	3

Collection::excludesAll()	Standard Library	FR02.8	3
Collection:any()	Iterator Expression	FR02.9	3
Collection::exist()	Iterator Expression	FR02.9	3
Collection::forAll()	Iterator Expression	FR02.9	3
Collection::isUnique()	Iterator Expression	FR02.9	3
Collection::one()	Iterator Expression	FR02.9	3
Collection::iterate()	Iterate Expression	FR02.10	3
Collection::count()	Standard Library	FR02.8	3
Collection::sum()	Standard Library	FR02.8	3
Collection::product()	Standard Library	FR02.8	3
Set::=	Standard Library	FR02.8	3
Set::asSequence()	Standard Library	FR02.8	3
Set::asBag()	Standard Library	FR02.8	3
Set::asOrderedSet()	Standard Library	FR02.8	3
Set::select()	Iterator Expression	FR02.9	3
Set::reject()	Iterator Expression	FR02.9	3
Set::collect()	Iterator Expression	FR02.9	3
Set::collectNested()	Iterator Expression	FR02.9	3
Set::sortedBy()	Iterator Expression	FR02.9	3
Set::-	Standard Library	FR02.8	3
Set::intersection()	Standard Library	FR02.8	3
Set::symmetricDifference()	Standard Library	FR02.8	3
Set::union()	Standard Library	FR02.8	3
Set::excluding()	Standard Library	FR02.8	3
Set::including()	Standard Library	FR02.8	3
Set::count()	Standard Library	FR02.8	3
Set::flatten()	Standard Library	FR02.8	3
Bag::=	Standard Library	FR02.8	3
Bag::asOrderedSet	Standard Library	FR02.8	3
Bag::asSet()	Standard Library	FR02.8	3
Bag::asSequence()	Standard Library	FR02.8	3
Bag::select()	Iterator Expression	FR02.9	3
Bag::reject()	Iterator Expression	FR02.9	3
Bag::collect()	Iterator Expression	FR02.9	3
Set::collectNested()	Iterator Expression	FR02.9	3
Set::sortedBy()	Iterator Expression	FR02.9	3
Bag::intersection()	Standard Library	FR02.8	3
Bag::union()	Standard Library	FR02.8	3
Bag::excluding()	Standard Library	FR02.8	3
Bag::including()	Standard Library	FR02.8	3
Bag::flatten()	Standard Library	FR02.8	3
Bag::count()	Standard Library	FR02.8	3
Sequence::=	Standard Library	FR02.8	3
Sequence::asBag()	Standard Library	FR02.8	3
Sequence::asSet()	Standard Library	FR02.8	3

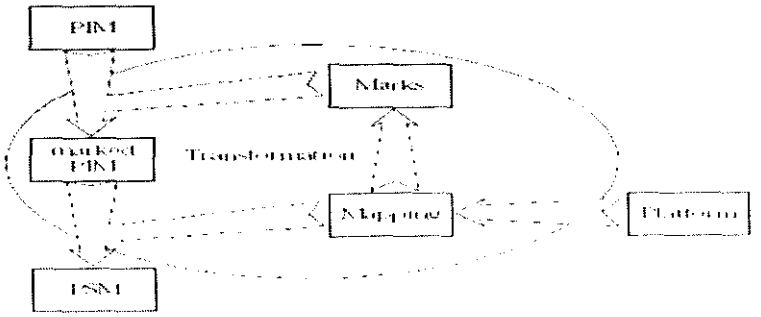
Sequence::asOrderedSet()	Standard Library	FR02.8	3
Sequence::select()	Iterator Expression	FR02.9	3
Sequence::reject()	Iterator Expression	FR02.9	3
Sequence::collect()	Iterator Expression	FR02.9	3
Set::collectNested()	Iterator Expression	FR02.9	3
Set::sortedBy()	Iterator Expression	FR02.9	3
Sequence::union()	Standard Library	FR02.8	3
Sequence::excluding()	Standard Library	FR02.8	3
Sequence::including()	Standard Library	FR02.8	3
Sequence::append()	Standard Library	FR02.8	3
Sequence::prepend()	Standard Library	FR02.8	3
Sequence::at()	Standard Library	FR02.8	3
Sequence::first()	Standard Library	FR02.8	3
Sequence::last()	Standard Library	FR02.8	3
Sequence::subsequence()	Standard Library	FR02.8	3
Sequence::count()	Standard Library	FR02.8	3
Sequence::flatten()	Standard Library	FR02.8	3
Sequence::indexOf()	Standard Library	FR02.8	3
Sequence::insertAt()	Standard Library	FR02.8	3
OrderedSet::append()	Standard Library	FR02.8	3
OrderedSet::prepend()	Standard Library	FR02.8	3
OrderedSet::insertAt()	Standard Library	FR02.8	3
OrderedSet::subOrderedSet()	Standard Library	FR02.8	3
OrderedSet::at()	Standard Library	FR02.8	3
OrderedSet::indexOf()	Standard Library	FR02.8	3
OrderedSet::first()	Standard Library	FR02.8	3
OrderedSet::last()	Standard Library	FR02.8	3
self	Variable Expression	FR02.4	2
result	Variable Expression	FR02.4	2

Appendix E: Model Transformation Approaches

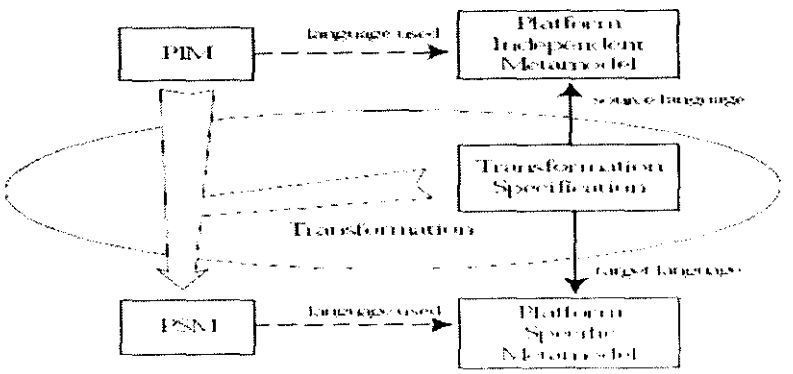
Table 21: Definition of elements if model transformation approaches

Elements	Definition
Platform Independent Model (PIM)	Model from platform independent point of view. Contains no platform specific information, thus allow the model to be use with different platforms
Platform Model	Model of the platform chosen to be applied to PIM. Contains parts and services that make up the platform.
Platform Specific Model (PSM)	A model that contains information that is in PIM and how it is used in a particular platform. PSM is a model from platform specific viewpoint.
Marks	Annotations on a PIM model that indicates how an element should be transform.
Mapping	Specification of the transformation that is written in a transformation language.
Patterns/Templates	Parameterized models that specify particular kind of transformation.

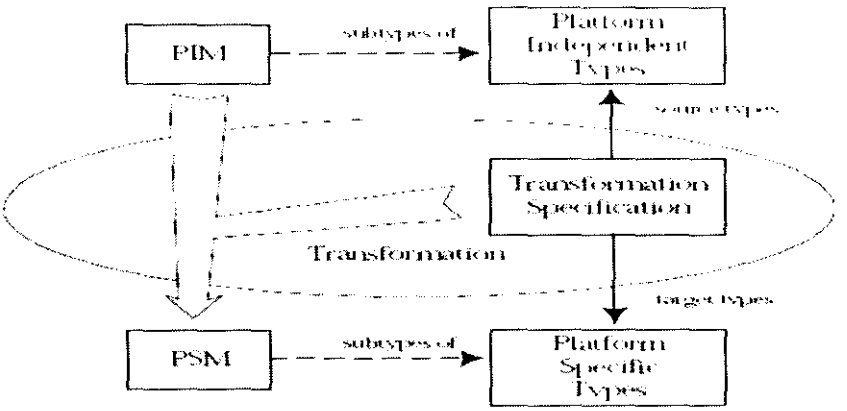
Marking



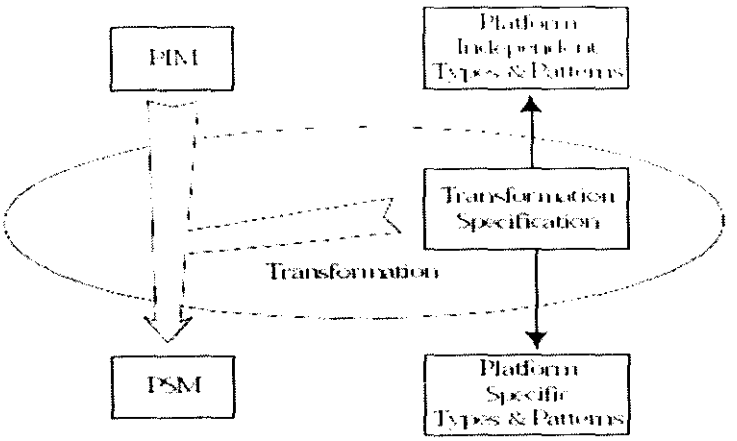
Metamodel Transformation

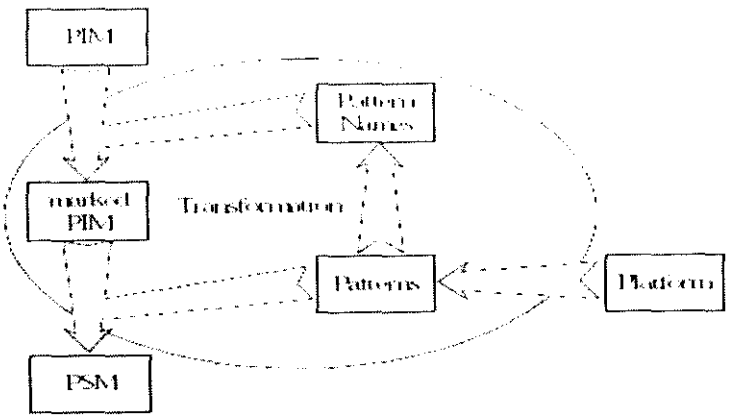


Model Transformation



Pattern Application





Model Merging