Send Orders for Reprints to reprints@benthamscience.ae

RESEARCH ARTICLE

A Comparative Study on Transformation of UML/OCL to Other Specifications

Jagadeeswaran Thangaraj* and Senthilkumaran Ulaganathan

School of Information Technology & Engineering, VIT University, Vellore, TamilNadu, India

ARTICLE HISTORY

Received:
Revised:
Accepted:

DOI:

Abstract:

Background: Static verification is a sound programming methodology that permits automated reasoning about the correctness of an implementation with respect to its formal specification before its execution. Unified Modelling Language is most commonly used modelling language which describes the client's requirement. Object Constraint Language is formal language which allows users to express textual constraints regarding the UML model. Therefore UML/OCL express formal specification and helps the developers to implement the code according to the client's requirement through software design.

Objective: This paper aims to compare the existing approaches generating Java, C++, C# code or JML, Spec# specifications from UML/OCL.

Method: Nowadays, software system is developed via automatic code generation from software design to implementation when using formal specification and static analysis. In this paper, the study considers trans-formation from design to implementation and vice versa using model transformation, code generation or other techniques.

Results: The related tools, which generate codes, do not support verification at implementation phase. On the other hand, the specification generation tools do not generate all the required properties which need for verification at implementation phase.

Conclusion: If the generated system supports the verification with all required properties, code developer needs less e orts to produce correct software system. Therefore, this study recommends introducing a new framework which to be act as an interface between design and implementation to generate verified software systems.

Keywords: Model Transformation, UML, OCL, Spec#, JML, Formal Specifications, Static Verification.

1. INTRODUCTION

The software development cycle [1][2] consists of different phases which include Requirements, Design, Implementation and Testing as shown in Fig. 1. Soft-ware requirements specify the needs of clients that are to be achieved by the system. Requirements analysis comprises those tasks that take into determining the needs or conditions to achieve for a new or altered system. Software design is the activity following requirements analysis. It is the process of creating conceptual design or frame which indents to achieve the goals of a system using set of components and subject to constraints. Implementation is the original process which makes the runnable system by generating executable codes or programming techniques. Implementing a system requires expertise programmers to develop a complex software system using complex algorithms, formal logics and different programming techniques with the knowledge of application domain. Therefore, implementation is the major and larger process in software development cycle. Software testing is the process of investigating the quality of software system with information provided by clients.

Software development is measured by a 'well defined' process which delivers software that behaves as expected and produces correct results in order to reduce complexity and save energy. Many software development approaches have been introduced to deal with development for producing correct systems, improving its quality and reducing its complexity, time and cost of development. This paper discusses the comparison about various works done in generating verified software systems.

*School of Information Technology & Engineering, VIT University, Vellore, Tamilnadu, India; E-mail: jagadeest@gmail.com

2213-2759/19 $58.00+.00 **© 2019 Bentham Science Publishers**

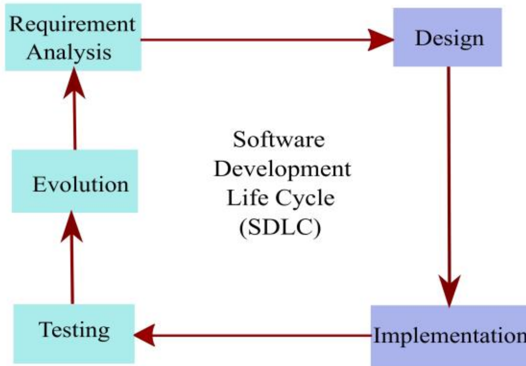

Fig. 1. Software Development Life Cycle

2. BACKGROUND & MOTIVATION

2.1. Formal Specification

Formal specifications are used to specify requirements when developing soft-ware. These specifications form a contract between the customer/client and the developers, i.e., the customer specifies the needs of the system and the developer implements the system according to the customer's specification. The for-mal specifications specify number of contracts in design of software development. These contracts are influenced by Design by contract (DbC) mechanism [3]. DbC is a programming methodology where the behavior of program components is described as a contract between the provider and the clients of the component [4]. The client explains the provider what he or she expects from the system. The provider implements the system to achieve the goal of client's requirements and he is free to choose the system as long as it meets requirement of the client. Design by Contract has become a popular methodology for object-oriented languages.

2.2. Contract Specification

Contract is the agreement between the client and the provider. These contracts are specified using side effect free expressions (Boolean) or assertions. These contracts are classified as follows [4].

- Method contracts
- Class level specifications
- Non-Null or Nullable expressions
- Expressions
- Specification for verification

Method contracts: Method contracts express the conditions over the method invocation using some natural language or other formalism. For example, what should be the result after or before the method invocation. The precondition specifies the constraint which must be satisfied before the execution of the method. The postcondition specifies the constraint which must hold after execution of the method. As a constructor comes live after its object invocation, It should not specify any precondition over the object when specifying the constructor. It can have post conditions over constructor specification. Visibility specifications are used to access modifiers for the specification purposes.

Class level specifications: Class level specifications express the condition which stands for the entire life of the class object. An invariant is a constraint of a type over an element of the model. This expression must be true for all instances of that type at all times. Class invariant should be true for entire object's life. Some condition should be true when the invocation of that object. For example, the initial value of some properties of an object are expressed using these specifications. Some specifications express the condition over the object's history values. During inheritance of objects, some properties must be hold. The inheritance specifications are used to express those conditions.

Non-Null or Nullable expressions: These expressions specify the condition whether the properties can refer non-null or nullable references.

Expressions: Some specifications express constraints over statically unknown length of properties by using quantifying expressions. Some specifications express constraints over the generalized values of properties. Also, constraints over the pre-state values are expressed using these expressions.

Specification for verification: Some condition expresses over the body which may change the result of the method, which is called framing problem. To avoid this problem, framing is used when using software verification process. Some constraints express the condition using loop variables. Frame conditions limit the parts of the program state that the method is allowed to modify. Therefore, it may change the state of that variable. Loop invariants are used to avoid this problem which avoid the state change over loop variables. Some specifications are used to avoid the state change of particular block.

2.3 Static verification

A requirement specification defines what a system must do and the design specification defines how to implement the system. The design specification plays an important role in the software development to produce correct results. These specifications are expressed in a formal specification language. If a formal specification is used in a software design, it can be verified using some mathematical techniques called formal verification techniques [5]: theorem proving, model checking and equivalence checking.

Static verification is a process of checking that implementation meets requirements by inspecting the code before it runs i.e., that checks the correctness of an implementation with respect to its formal specification before its execution. Therefore, it is the set of processes that analyses the program to ensure defined coding practices are being followed, without executing the application itself.

2.4 Ownership & Types

Object-oriented programming (OOP) is a programming methodology for implementing software systems. In an object-oriented program, classes contain description about data fields and methods. An object is an instance of a class

which has state and methods which can change that state. Any object can refer to other objects. Aliasing occurs when one object is reachable through multiple paths, i.e., more than one reference is referred by the same object. Owners have privileged access [6] to a given object or objects in the context. That means, the owners have exclusive write access to the object they own. Therefore, the ownership determines the owner who enjoys privilege access to the object. When accessing an object, that object's owner will not permit changing the values. This aliasing is a technique which allows a programmer to share the single data source with different objects [7]. Ownership helps to control aliasing and assists in structuring object relationships in a program [8][9]. By using this ownership representation, an owner object can access the reference objects. Ownership types help the programmer track information about object aliasing. Ownership types are introduced to deal with the object aliasing [10]. Ownership type systems are classified into two types in the sense of encapsulation: *Universe* or *Dynamic*.

A universe type is a static system that enforces the ownership topology and modifier owner encapsulation [11]. The universe type system allows objects to migrate from one owner to another to facilitate modification of objects [12]. Software verification is a process that ensures that a software system meets its specification and that it fulfils its intended purpose. The universe type system allows modifying owner references for the purpose of verification. In this system, references are used to read attributes and to call pure methods. Pure methods are methods which do not cause side effects.

2.5 Motivation

Nowadays, software is developed via automatic code generation from software designs to implementation when using formal specification and static analysis [13]. The modelling approaches are used to describe the client's specification and translates to implementation. The correct software maintains the consistency of a programs data throughout its verification. If it fails to maintain the consistent details, the system may fail and lead to a number of errors during program development. Therefore, transformation of these specifications plays major role in verification of design and implementation. So that, it helps in reducing programmer's complexity and saves their energy in the sense of time and e ort of implementation. This paper presents,

1. Formal Languages to represent the specification at design and implementation phases with verification support
2. Methodologies in transforming specification between design and implementation
3. Related works done so far for this purpose and Comparison of these related works

3. FORMAL LANGUAGES

There are number of formal languages available to verify the correct software systems. This section explains few formal specification languages and their features which support to specify the contracts over software development.

3.1 UML & OCL

The Unified Modeling Language (UML) is most commonly used modelling languages [14] that includes a graphical representation to create an abstract model of a system, referred to as a UML model [15]. The UML is a language for specifying artefacts of software system at Design phase. The UML model describes object-oriented programming components clearly. In the UML model, most of the programming details can be described by graphical structures, i.e., class diagrams, state diagrams and object diagrams.

A class diagram depicts the details of a class of the model in an object-oriented system. Classes have attributes, operations and relationships with other classes. The relationship restrictions with other classes can be described by associations which are called UML constraints. Association multiplicities de ne the connection relation of classes to each other, i.e., a class is related to other classes through its role name and number of instances. However, the UML model needs some more facilities in order to add textual constraints on it directly. The textual constraints contain the restriction of a system's state. Those specify what the system must have in its implementation.

The Object Constraint Language (OCL) is a formal language which is part of the UML standard. It solves this problem by allowing users to express textual constraints regarding the UML model [16]. As OCL expressions are side-effect free, they can query or constrain the state of the system but not modify it. Be-cause OCL is a declarative language, means that OCL does not include imperative constructs like assignments. Therefore, the evaluation of an OCL expression does not change the state of a model [17]. The OCL constraints simply state what should be true, but not how this should be achieved.

OCL is used to express constraints: invariants, preconditions and postconditions. These constraints do not affect the state of the system. Normally the OCL is used to specify the property that must not change through the execution of the design of the system, i.e., software execution. Some constraints must be true before and after the invocation of methods.

3.2 Spec#

The Spec# programming system provides static verifier for C# programs. Spec# is a formal language, which extends C# with constructs for writing specification [18]. It supports design by contract properties over C# to allow programmers to express their restrictions or constraints in the implementation. The constraints are the same as OCL constraints: invariants, preconditions and postconditions [18]. The Spec# system consists of: the Spec# programming language, the Spec# compiler and the Spec# static program verifier (Boogie)[19][20]. Spec# supports object ownership at the implementation phase to permit static verification. It supports holding the reference of component objects as well other object references; for example, during an inheritance, a subclass object may have reference to superclass objects.

Spec# supports distinguishing the other objects in the current class con-text. A dynamic ownership system checks the universe type encapsulation with ownership transfer during program execution. In a dynamic ownership system, the

ownership of an object may temporarily be broken. The reference may be passed, and the owner's states may be changed during program verification. This *ownership transfer* is used for verification purposes and is done using representative exposure. This is otherwise called *expose blocks* which is a frame used in the Spec# programming language. Inside this *expose block*, the owner is mutable. Therefore an owner object can modify the values of a referenced object inside the *expose blocks*. These modifications do not affect the actual results because the modifications are completed using query statements. Query statements are statements which gives the results with side effect free. Here the *'this'* object acts as the owner of the referenced object. The dynamic ownership is implemented in the Spec# language for the purpose of the verification of C# programs [21].

Dynamic ownership systems enable ownership transfer in the *expose blocks* during program execution. This dynamic ownership is supported by three major constructs: Object topology, ownership types and representation exposure. In Spec#, an object can refer to other objects for the internal definition of its data. The *[Rep]* keyword is used to annotate such attributes. Therefore the *'this'* object is declared as the owner of *Rep* referenced objects. Generally, an invariant is a constraint of a type over an element of the model, i.e., expressed by the OCL expression [16]. Object invariant is a constraint of the object during its instance. Object invariants must be true all the time for an object instance. During execution of a Spec# program, it is necessary to break some object invariants for the purpose of verification [22]. Therefore Spec# introduces a block statement called expose block. Invariants are temporarily broken by exposing an object using the expose construct. The object invariants may be broken within an *expose block* [18] i.e., the object invariant cannot be proved as a logically true inside the *expose block*. In the expose block, an owner is mutable. Therefore the current owner is the owner of the referenced object. At the end of an expose block, the object invariant must hold. This ownership transfer supports the program verification in the dynamic ownership system.

3.3 JML

The Java Modelling Language (JML) is a formal specification language for Java which specifies behavior of Java classes i.e., it records design and implementation decisions. JML adds assertions as annotation to specify invariants, preconditions and postconditions. This system also uses the design-by-contract approach to annotate programs with assertions so that tools, such as compilers and theorem provers, can generate the proof obligations required to verify that a program satisfies its specification.

This universe ownership type system is implemented in JML compiler for the purpose of the verification of Java programs. Therefore, this system checks the ownership properties statically at compile time [23]. In JML, the owner of an object is determined during object creation. JML distinguishes references into three major types.

- **peer** references
- **rep** references
- **readonly** references

peer references are references between objects in the same context. The **rep** references are references from an object X to an object directly owned by X. The **readonly** references are references between objects in arbitrary contexts[24]. **readonly** reference does not specify an ownership context. The readonly reference may refer to objects with arbitrary owners. They denote a reference that is read-only and might point to objects in any context. Therefore, references specified with the *readonly* modifier cannot be used to modify the referenced object. In JML, the modifier any can be used as a semantically equivalent alternative. The modifier ownership type does not allow modifying the reference objects of arbitrary type.

3.4 Formal Software Verification

Static verification is a process of checking that implementation meets requirements by inspecting the code before it runs i.e., that checks the correctness of an implementation with respect to its formal specification before its execution. The Spec# programming system provides static verifier for C# programs. Spec# is a formal language, which extends C# with constructs for writing specification [18]. It supports *design by contract* properties over C# to allow programmers to ex-press their restrictions or constraints in the implementation. The constraints are the same as OCL constraints: invariants, preconditions and postconditions [18]. The Spec# system consists of: the Spec# programming language, the Spec# compiler and the Spec# static program verifier (Boogie). Spec# supports object ownership at the implementation phase to permit static verification. It supports holding the reference of component objects as well other object references; for example, during an inheritance, a subclass object may have reference to super class objects. Spec# supports distinguishing the other objects in the current class context.

Similarly, JML annotations specifies client's constraints over Java programs. There are numerous tools available to verify Java programs with respect to JML such as: Runtime Assertion Checking (*jmlc*), Unit testing tool for Java (*jmlunit*), Extended Static Checking (*ESC/Java*), Program Verification with LOOP, Static Verification with *JACK* and *KeY* tool. ESC/Java tool [25] checks simple assertions and some common errors in Java programs as static checking. KeY[26] is one of popular verification tool of JML. This tool helps to formally verify the correctness of Java programs.

4. MODEL DRIVEN ENGINEERING

New technologies have been introduced to reduce the complexities of software development. Model Driven Engineering (MDE) is one of the most popular mechanisms of software development. MDE refers to the systematic use of models in the entire software engineering life cycle [13][27]. There are two popular methodologies available in practice for generating Design specification to Implementation.

- Model Transformation
- Design Patterns

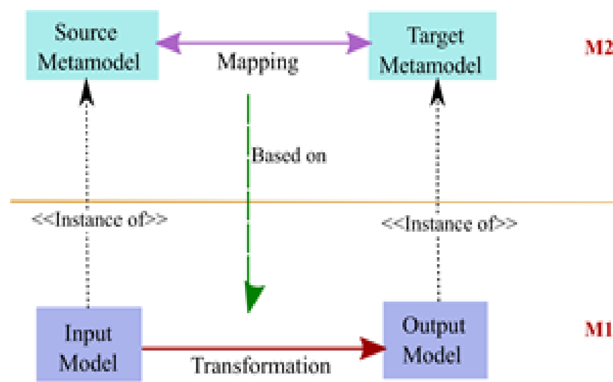

Fig. 2. Model Transformation

4.1 Model

A model gives a full description of a system. In order to represent an overview of a system to develop, a model for the system is used in the design phase. These models are used in model transformation approaches. For example, a UML class diagram is one such model. Any kind of model can be transformed by Model transformations, but they must fulfil the specification requirement.

4.2 Metamodel

In MDE, metamodels play a key role. A metamodel defines the abstract syntax of models and the interrelationships between model elements. Metamodeling is a technique for constructing properties of the model in a certain domain. The metamodel defines the general structure and characteristics of real world phenomena. Therefore, a model must conform to this protocol of the metamodel similarly to how a grammar conforms for a programming language.

4.3 Model Transformation

Model Transformation is a mechanism for transforming one model into another model, based on some transformation rules. The OMG introduced two modelling layers M1 and M2 as shown in Fig. 2. M1 represents the models and M2 represents metamodels. Each model in M1 is an instance of a Metamodel in the M2 layer. A transformation engine reads one source model conforming to a source metamodel and writes a target model conforming to a target metamodel. According to its characteristics, the model transformation is classified into two types: *Horizontal* or *Vertical* transformation. *Horizontal transformation* transforms the models based on semantic equivalence. *Vertical transformation* transforms model into another abstraction level model.

With respect to target models, model transformation is classified into two types: *Model To Model* (M2M) and *Model To Text* (M2T). M2M transformation creates its target as a model which conforms to the target meta-model. M2T transformation creates strings for a given input model. For example, UML2C# [28] is a M2T translation and UML2SQL [29] is a M2M translation. UML2C# translates UML models into C# code and the UML2SQL generates the SQL model from the UML model. QVT and ATL are popular model translation languages.

4.4 Design Patterns

A design pattern is a reusable method to support software construction to a given problem which recurrently occurred in the context of software design. It provides templates for how to solve a problem that can be used in many different situations. The design patterns become a standard in model transformation. These patterns are grouped into following categories: creational patterns, structural patterns, behavioral patterns and concurrency patterns [30]. Design pattern gives an intermediate structured view of program paradigm between specification and implementation. In most of works, patterns act as auxiliary metamodel, auxiliary correspondence model, entity splitting/merging, transformation chain and structured preservation.

5.SCOPE OF STUDY

This paper compares some related works done in transformation from the soft-ware design to implementation in order to support software verification. In literature, this paper considers transformation from UML or UML/OCL to other specifications and vice versa. These systems translate using model transformation (model to model) or code generation (model to text) or other techniques.

5.1 Criteria for Evaluation

This paper evaluates the related works using some analysis parameters and those are as follows.

Direction: This parameter defines transformation direction. Values, this paper used, are unidirectional and bidirectional. Unidirectional technique transforms from specification to implementation only and bidirectional also transforms vice versa.

Source Language: In this survey, this paper considers source languages into account. Since, primary focus of this paper is on UML/OCL, this paper checks that different constructs of UML/OCL like UML, OCL or UML/OCL. Here UML specifies that UML is only source language, whereas, OCL specifies that OCL is only source language. UML/OCL specifies both specifications as source language. In this paper, OCL is used to mention OCL constraints, OCL expressions or OCL specifications regardless the authors usage to avoid confusion as they specify same concept with different terminology.

Target Language: In this survey, this paper considers target languages into account. This parameter specifies the output of the tools, that is, target specifications such as Java, C#, C++ codes or JML, Spec# specification languages. This is the important criteria to evaluate the verification support.

Methodology: This parameter defines transformation methodology of the work such as model transformation, code generation or design patterns.

Transformation: This parameter defines whether the transformation is manual or automatic.

Verification support: Some works support verification of specification or implementation or both. This paper uses parameters as X or ✓ and levels.

Collection Support: Target coverage parameter describes the coverage of collection operations in target specification. Since, primary focus of this paper is on target specification/code. This paper checks the support to different collection operations in the target specification using standard collection, user specified library. X denotes the no support for collection operations.

Scalability: This parameter evaluates the potential and capability of the system. This paper uses three parameters: High, Med and Low. *'High'* specifies the system with high potential and verified using a number of case studies in their paper. '*Med'* denotes less case studies and *'Low'* denotes no case studies respectively.

Ownership addition: This parameter describes the coverage of ownership type constraints at specification level. This paper uses two parameters: X and ✓. ✓ denotes the addition of ownership type constraints and X denotes the no coverage.

Historic variables: Historic variables denote the production of appropriate historic variable in the generated specification (e.g. denotes old or @*pre*). This paper uses X or ✓.

6.RELATED WORKS

Many researchers are working on the translation of UML/OCL specifications into other code/specifications for different purposes. Following section discusses some related works in transformation of UML/OCL.

6.1 Pattern based Translation of OCL to JML

Ali Hamie developed a system [31] which translates OCL into another Design by Contract like language JML based on the concept of constraint patterns. This system translates constraints in OCL into an equivalent JML expression by mapping the patterns of constraints of both languages. In this approach, the translation is done by mapping basic types, collection operators, iterating expressions and other OCL constructs into equivalent JML constructs.

Limitations: This approach does not provide any correctness of the transformation but does the automating translation. It just transforms similar constructs from both languages. This approach only concentrates the assertions generations, and not about the program code. Also, this does not concentrate on ownership. Addition of the ownership concepts could be implemented in this work to ensure object aliasing concepts.

6.2 Bidirectional Translation between OCL and JML

This work is done by *Shimba et al* [13]. In this tool, one can translate UML class diagram with OCL into Java skeleton code with JML annotations. They developed this tool by using *Xtext* plugin in the Eclipse framework. *Xtext* uses the syntax of the model for transformation. Models are defined in an editor and transformed into target text by the transformation rules of corresponding properties.

Limitations: Authors verify their tool by checking some known examples in both languages. There is no formal verification of the transformation. Also, there is no support for reasoning about ownership and framing conditions.

6.3 OCL2Spec#: Adding ownership constraints to OCL

OCL2Spec# system [34] introduces ownership constraints to OCL to make this information available during the design specification, in order their implementations easier to develop and less prone to error. This system allows to add ownership constraints to OCL specification to describe the program's specification at the specification level and Spec# to develop the code at the implementation level. The *OCL2Spec#* generates the Spec# program with the standard collection library: List.

Limitations: This approach mainly concentrated about ownership type addition at specification level. It is evaluated by the correctness of ownership type translation from OCL to Spec#.

6.4 Incremental OCL to C# Code Generation

Tamás Vajk et al. developed a tool framework which generates C# code from OCL [35] by incremental iterations. This framework generates C# code from OCL constraints using incremental code generation. This incremental process iterates the constraints to minimal code in C#.

Limitations: This tool generates C# code from OCL constraints directly. It does not allow the verification of the code during the implementation phase. Also, it does not generate specifications and not support verification about code generation.

6.5 Generating Assertion code from OCL

Rodion Moiseev et al. worked on a tool which generates assertion code from OCL [36] based on the similarities of the languages: OCL, Java, Python and Haskell. In their approach, they translate constraints in OCL into an equivalent assertion code in the target language code to check the correctness of the program. They implemented this transformation using the similarity of the programming languages which can be modelled via abstract syntax Tree (AST). They wrote the transformation rules based on these AST similarities between the languages. Their input is a model of OCL AST. This tool generates the target language's AST and then transforms equivalent executable code by applying set of rewriting rules. They evaluated their tool by generating assertion code in four different languages: Java, Python, Haskell and O'Haskell.

Limitations: They uncovered the functionalities of history expressions, i.e., previous value of attributes (@*pre* value). They did not check the consistency of the transformation i.e., accurate transformation from OCL to generated code. And their transformation did not support the hierarchy properties.

6.6 UML-RSDS

Recent days, *Kevin Lano* is working towards on a project, *UML-RSDS* tool, which is a subset of UML 2.0 and OCL 2.0 [37][38]. *UML-RSDS* generates implementation code in Java, C#, C++ and JSP/Servlets from UML/OCL [39]. This tool is used in part of software development in design and implementation. It ensures the consistency between these languages using model transformation and design pattern concepts.

Tools/ Properties	**OCL2 Spec# [34]**	**Hamie's [31]**	**Shimba's [13]**	**Rodion's [36]**	**UML-RMDS [37]**	**Tamás's [35]**	**XRTSDIC [40]**
Direction	Unidirection al	Unidirectional	Bidirectional	Unidirection al	Unidirectional	Unidirection al	Unidirection al
Source Language	UML/OCL	OCL	OCL	OCL AST	UML/OCL	OCL	UML
Target Language	Spec# skeletons	JML expressions	Java skeletons	AST (Java, Haskell, O'Haskell)	Java, C#, C++, JSP/Servlets	C#	Java, C#, C++
Methodology	Code generation	Design Patterns	Model Transformation	Code generation	Pattern based Code generation	Incremental Code generation	Code generation
Transformation	Manual	Automatic	Automatic	Manual	Automatic	Automatic	Automatic
Verification support	Both	Code Implementation phase	X	X	X	X	X
Collection support	Standard C# collection	JML Standard collection	X	X	OCL Set & Sequence	Standard C# collection	X
Scalability	High[34]	Med[32]	Med[33]	Low[36]	High[38][39]	Med[35]	Med[40]
Ownership addition	Software Design phase	Code Implementation phase	X	X	X	X	X
Historic variable	✓	✓	X	X	X	X	X

Table 1. Comparison of related works in transformation of UML/OCL

Limitations: This tool generates implementation code in programming languages from UML/OCL. It does not generate the formal specifications like JML. Therefore, it does not support the verification at implementation phase.

6.7 XRTSDIC: model transformation from PIM to PSM

Ramesh has developed a framework, named *XRTSDIC*, which supports the model transformation of UML to Java, C# and C++ [40] using the concept of `Dialects'. This tool is used in part of software development in design and implementation. It ensures the consistency between these languages using consistency checker algorithm.

Limitations: This framework automatically generates the source code from UML model. This does not support verification either in design or implementation phases. But it uses the consistency checker to make sure the correct code generation from a given UML model.

7.DISCUSSION

Table 1. shows the comparison of the approaches generating code/specifications from UML/OCL design. These approaches transform UML/OCL specifications to the target specifications such as Java, C#, C++ code or JML, Spec# specification languages in order to generate verified software systems. The main source language is OCL specification. *OCL2Spec#* and *UML-RMDS* use UML along with OCL specification as source language. *XRTSDIC* use only UML as source language. All these tools generate equivalent code in different programming languages such as Java, C# and C++ or target specifications in JML or Spec# specification languages. These tools generate code directly using code generation, intermediate model transformation and design patterns. *Rodion's tool* [36] generates the OCL AST into an equivalent AST in three other languages: Java, Haskell and O'Haskell. It needs a second translation engine to generate the code. All approaches transform unidirectionally except Shimba's bidirectional tool. As discussed in section 2.4, the ownership types play important role in verification. Therefore, it is important to produce right ownership types in the implementation as done by *Hamie's* and *OCL2Spec#* tools. But *OCL2Spec#* allows adding ownership types to OCL at the design phase of software development. OCL specifications use the standard OCL collection library to specify the collection operations in the constraints. Therefore, target specifications must have the corresponding operations to specify them correctly when generating specifications to support the verification. OCL2Spec# and *Hamie's* tool use the corresponding C# or JML standard library in this scenario. Most of these tools have verified using different case studies as mentioned in their papers. Also, these tools transform the historical variable which must support in the verification specifications. *OCL2Spec#* and Rodin's tool transform manually, where, others are automatic transformation tools. As a conclusion, the related works transform Java, C#, C++ codes or JML, Spec# Specifications. The related tools, which generate codes, do not support verification at implementation phase. Therefore, it must go under testing process to verify the final system. On the other hand, the specification generation tools, *Hamie's* and *OCL2Spec#,* do not generate all the properties which need for verification at implementation phase. If the generated system supports the verification with all required properties, code developer needs less e orts to produce correct software system. Also, it saves cost and time by holding all the required properties for verification and avoiding long testing processes. Therefore, there is a need of new work which supports the transformation of the target

specification with support of verification at implementation phase.

8. CONCLUSION OF STUDY

This paper has explained the comparison of approaches towards the transformation from UML/OCL to implementation to generate verified software system. It gives a clear idea how the tools operate, what properties support or not support, strengths and weaknesses as discussed in the previous section. As a conclusion, class artefacts, such as UML and OCL specifications, are a major component of object-oriented design and development. These artefacts contain static class structure with specification constraints. Yet there are many barriers to describing and using them correctly across different models in a model-oriented environment. Finally, it needs to investigate the class artefacts at the intra- and inter-model level and investigate the portability of these artefacts between different phases. Some work analyzed frame conditions in UML/OCL [41]. Therefore, if a system has a suitable methodology which supports re-usability of class artefacts, then it will be much efficient system in the transformation. Therefore, this study recommends introducing a new framework which to be act as an interface between design and implementation. The new framework should help in producing verified system in the target language according to the design specification and to support the re-usability of the specification. Also, it should be able to produce specifications in different languages enabling the implementation by expertise in that language. Therefore, the new to be proposed framework will be based on the sustained development of different formalism to support interoperability between these specifications.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers whose valuable comments and suggestions helped to improve and clarify this paper.

REFERENCES

1. Vanshika Rastogi: Software Development Life Cycle Models - Comparison, Consequences, In International Journal of Computer Science and Information Technologies (IJCSIT), Vol.6 (1),168-172 (2015).
2. P. Barthelmess and K. M. Anderson: A View of Software Development Environments Based on Activity Theory, In: Computer Supported Cooperative Work, Kluwer Academic Publishers, 1-2 (2002)
3. Gary T. Leavens, Yoonsik Cheon: Design By Contract with JML, ACM - http://www.eecs.ucf.edu/leavens/JML/jmldbc.pdf (2006)
4. Huisman, M., Ahrendt, W., Bruns, D., Hentschel, M.: Formal specification with JML. (Karlsruhe Reports in Informatics: No. 2014-10). Karlsruhe: Department of Informatics, Karlsruhe Institute of Technology (2014).
5. Hasan Amjad: Combining model checking and theorem proving, Computer Laboratory, Department of Computer Science, University of Cambridge (2004)
6. Peter Müller: Reasoning about Object Structures Using Ownership, Verified Software: Theories, Tools, Experiments: Springer Berlin Heidelberg, 93-104 (2008)
7. Tobias Wrigstad: Ownership-Based Alias Management, Computer and Systems Sciences, KTH Information and Communication Technology, Stockholm, Sweden (2006)
8. Mycroft, Alan and Voigt, Janina: Notions of Aliasing and Ownership, Aliasing in Object-Oriented Programming. Types, Analysis and Verification: Springer Berlin Heidelberg, 59-83 (2013)
9. John Hogg and Doug Lea, Alan Wills, Dennis de Champeaux and Richard Holt: The Geneva Convention on the Treatment of Object Aliasing, Aliasing in Object-Oriented Programming. LNCS: Springer 7-14, (2013)
10. Dave Clarke, Johan Astlund, Tobias Wrigstad: Ownership Types: A Survey, Aliasing in Object Oriented Programming. LNCS: Springer, 15-58 (2013)
11. Werner Dietl, Peter Müller: Object Ownership in Program Verification, Aliasing in Object-Oriented Programming. Types, Analysis and Verification: Springer, 289-318 (2013)
12. Annetta Schaad: Inferring Universe Annotations on the Presence of Ownership Transfer, Software Component Technology Group, Department of Computer Science, ETH Zurich (2007)
13. Hiroaki Shimba, Kentrao Hanada, Kozo Okano and Shinji Kusumoto: Bidirectional Translation between OCL and JML for Round-Trip Engineering, 20th Asia-Pacific Software Engineering Conference (APSEC), Bangkok, pp. 49-54. DOI: 10.1109/APSEC.2013.111 (2013)
14. Beatriz Perez, Ivan Porres: An Overall Framework for Reasoning About UML/OCL Models Based on Constraint Logic Programming and MDA, In International Journal On Advances in Software: ICSEA (2013)
15. OMG: Unified Modeling Language (UML): Version 2.4.1. Object Management Group, http://www.omg.org/spec/UML/2.4.1, (2011)
16. OMG: Object Constraint Language(OCL): Version 2.3.1. Object Management Group, http://www.omg.org/spec/OCL/2.3.1 (2012)
17. Jordi Cabot, Martin Gogolla: Object Constraint Language (OCL): A Definitive Guide, In SFM 2012: Springer 58 - 98 (2012)
18. Mike Barnett, Rustan Leino, Wolfram Schulte: The Spec# programming system: An overview, In Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS), 46-69 Springer (2004)
19. K. Rustan M. Leino, Peter Müller: Using the Spec# language, methodology, and tools to write bug-free programs, LASER Summer School 2007/2008: Springer-Verlag, (2008)

20. Mike Barnett, Rustan Leino, Wolfram Schulte and Peter Muller: Specification and Verification: The Spec# Experience, Object Oriented Programming (2009)
21. K. Rustan M. Leino and Peter Müller: Object Invariants in Dynamic Contexts, ECOOP Object-Oriented Programming, 491-515 (2004)
22. Peter Müller: Modular Specification and Verification of Object-Oriented programs, PhD thesis, FernUniversitat Hagen, Germany (2001)
23. Werner Dietl, Peter Müller: Universes: Lightweight Ownership for JML. In Journal of Object Technology: ETH Zurich, 5-32 (2005)
24. Dave Cunningham, Werner Dietl, Sophia Drossopoulouand, Adrian Francalanza, Peter Muller and AJ Summers: Universe Types for Topology and Encapsulation. In Journal of Formal Methods for Components and Objects: Springer-Verlag, 72-112 (2008)
25. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI2002), pages 234-245, (2002).
26. Bernhard Beckert, Reiner Hahnle, Martin Hentschel and Peter H. Schmitt: Formal Verification with KeY, Deductive Software Verification - The KeY Book: From Theory to Practice, Springer, LNCS 10001, 541-570, (2016).
27. Florian Heidenreich, Christian Wende, Birgit Demuth: A Framework for Generating Query Language Code from OCL Invariants, In proceedings of the Workshop Ocl4All: Modelling Systems with OCL with MoDELS (EASST 2007)
28. INRIA: ATL Transformation Example: UML 2 Java, Eclipse Framework, http://www.eclipse.org/atl/ atlTransformations/UML2Java/ (2005)
29. Leszek Siwik, Krzysztof Lewandowski, Adam Wos, Rafal Drezewski and Marek Kisiel-Dorohinicki: UML2SQL A Tool for Model-Driven Development of Data Access Layer, In Journal of Studies in Computational Intelligence: Springer Berlin Heidelberg, 227-246 (2010)
30. K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, M. Sharbaf: A survey of model transformation design pattern usage, ICMT (2017)
31. A. Hamie, Using Patterns to Map OCL Constraints to JML Specifications, In Model-Driven Engineering and Software Development 2015, Springer International Publishing - Cham, (CCIS, volume 506) ISBN:978-3-319-25156-1, 35{48 (2015)
32. A. Hamie, Pattern-based mapping of OCL specifications to JML contracts, 2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Lisbon, pp. 193-200. (Jan 2015)
33. Kentaro Hanada and Hiroaki Shimba and Kozo Okano and Shinji Kusumoto: Implementation of a Prototype Bi-directional Translation Tool between OCL and JML, (2014)
34. Jagadeeswaran Thangaraj: Adding Ownership Constraints to OCL to automatically generate Spec# skeletons, MSc Thesis, Department of Computer Science, Maynooth University, Ireland (August 2015)
35. Tamás Vajk and Gergely Mezei: Incremental OCL to C# Code Generation, In IEEE International Joint Conferences on Computational Cybernetics and Technical Informatics (ICCC-CONTI 2010): IEEE, (2010)
36. Rodion Moiseev, Shinpei Hayashi, Motoshi Saeki: Generating Assertion Code from OCL: A Transformational Approach Based on Similarities of Implementation Languages, In Journal of Model Driven Engineering Languages and Systems: Springer, vol:5795, 650-664 (2009)
37. K. Lano: Agile Model-based Development using UML-RSDS, CRC Press (2017).
38. K. Lano: Families to Persons Case with UML-RSDS, In 10th Transformation Tool Contest, STAF (2017).
39. K. Lano: The UML-RSDS Manual, https://nms.kcl.ac.uk/kevin.lano/uml2web/uml rsds.pdf (2018).
40. G. Ramesh, XRTSDIC: Model Transformation from PIM to PSM, International Journal of Engineering and Technology (UAE), Vol 7 No. 1.8, pp. 92-98, DOI: 10.14419/ijetv7i1.8.9980 (2018).
41. Philipp Niemann, Nils Przigoda, Robert Wille and Rolf Drechsler: Analyzing Frame Conditions in UML/OCL Models - Consistency Equivalence and Independence. 139-151. In 6th International Conference on Model-Driven Engineering and Software Development, 10.5220/0006602301390151. (2018)