Transformation of TOCL Temporal Properties into OCL

Mustafa Al Lail
mustafa.allail@tamiu.edu
Texas A&M International University
Laredo, Texas, USA

Antonio Rosales*
Hector Cardenas*
antoniorosales@dusty.tamiu.edu
hector_cardenas@dusty.tamiu.edu
Texas A&M International University
Laredo, Texas, USA

Lars Hamann
lars.hamann@haw-hamburg.de
Hamburg Univ. of Applied Sciences
Hamburg, Germany

Alfredo Perez
alfredoperez@unomaha.edu
University of Nebraska at Omaha
Omaha, Nebraska, USA

ABSTRACT

Specifying and verifying the temporal properties of UML-based systems can be challenging. Although there exist some extensions of OCL to support the specification of temporal properties in UML-based notations, most of the approaches depend on using non-UML formal formalisms such as LTL, CTL, and CTL* while transforming the under-development UML models into non-UML model checking frameworks for verification. This approach introduces complexities and relies on techniques and tools that are not within the UML spectrum. In this paper, we show how TOCL (one OCL extension for temporal properties specification) can be transformed into OCL for verification purposes. Towards this end, we created a formal EBNF grammar for TOCL, based on which a parser and a MOF metamodel were generated for the language. Additionally, to facilitate the analysis of the TOCL properties, we formally defined transformation rules from TOCL metamodel to OCL metamodel using QVT. Finally, we validated the implementations of the transformation rules using USE.

CCS CONCEPTS

• **Software and its engineering → Unified Modeling Language (UML)**; **System modeling languages**.

KEYWORDS

Temporal propeties,TOCL, OCL,Transformation, UML

ACM Reference Format:
Mustafa Al Lail, Antonio Rosales, Hector Cardenas, Lars Hamann, and Alfredo Perez. 2022. Transformation of TOCL Temporal Properties into OCL. In *Submission to OCL Workshop'22, October 23-25, 2022, Montreal, CA.* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

*Both authors contributed equally to this research.

Submission to OCL WS'22, October 23–25, 2022, Montreal, CA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

For Model-driven Engineering (MDE) approaches to succeed, software designers must integrate the development process with practical techniques to improve the quality of models. If a model has unresolved design faults, they are propagated to the code where they can be more difficult to uncover and more expensive to remove. One approach to uncovering design errors is to formally specify and analyze the temporal properties that a system has to satisfy. Temporal properties are useful in capturing a broad range of relevant system properties and requirements [18]. Software designers can use temporal logic formalisms, (e.g., Linear Temporal Logic (LTL) [18] and Computation Tree Logic (CTL) [8]) to formally specify properties.

Although significant research appears in specifying and analyzing properties, there is not an effective and efficient UML-based framework that specifies and analyzes temporal properties. Most of the current approaches depend on using non-UML formal formalisms such as LTL, CTL, and CTL* for specifying properties while transforming the under-development UML models into non-UML model checking frameworks for verification. This approach introduces complexities and relies on techniques and tools that are not within the MDE spectrum.

Figure 1 depicts the analysis approach that represents the context of the research described in this paper [2, 3]. The approach

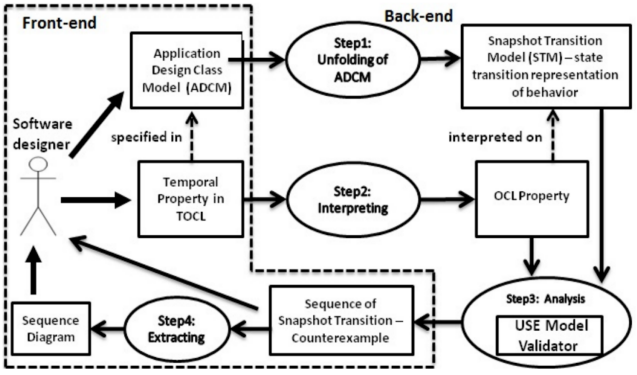

Figure 1: An Overview of the Analysis Approach.

exclusively uses UML-based notations, technique, and tools to analyze temporal properties. On the front-end, a UML designer creates a UML class model for a software application and specifies some temporal properties in TOCL (Figure 1 left). The designer then uses the approach to analyze the system's behavior for violation of the properties. The first step (Figure 1 top) is to 'unfold' the behavior of a class diagram by transforming it into a *Snapshot Transition Model* (STM) [20]. An *STM* is a class model that statically represents the behavior of a system in terms of states and transitions. As an *STM* is an ordinary class diagram, it can not only model different scenarios of system execution but also can be constrained by normal OCL expressions. The second step of the process translates a TOCL temporal property to an OCL expression interpreted on an *STM* (Figure 1 middle). The TOCL properties are translated into OCL query expressions and constraints, which traverse the state-transition chains, searching for any state that violates a specified property. Given (i) a class model representing the behavior and (ii) an OCL expression constraining this behavior, we can do the analysis using a UML static-analysis tool, which is the third step of the approach (Figure 1 bottom-right). The approach uses the USE Model Validator [16] to perform the analysis. If the analysis finds a state where the system violates a given property, the process produces a counterexample demonstrating that the system does not uphold the respective property. We can then return this counterexample to the designer as a sequence diagram or as a sequence of state-transition (Figure 1 bottom). Given this counterexample, the designer can examine the situation and revise and improve their design.

The research presented in this paper aims to revise and enhance the transformation from TOCL to OCL (step 2 in Figure 1 of the approach) of the original work presented in [2, 3]. In particular, the original transformation had the following drawbacks. First, the transformation was not formally defined, but rather was based on a collection of templates (Dwyers' et al. patterns [11]) that can be used to specify temporal properties in TOCL [3]. Even though the property specification patterns are expressive to specify most temporal properties, there are many new types of properties that can not be specified. The second drawback of the original work is that it has not been implemented.

In this paper, we aim to address these drawbacks. Towards this aim, we formally define and implement general transformation between TOCL and OCL by completing the following activities:

(1) We formally define an EBNF grammar for TOCL, based on which we created a parser for TOCL.
(2) We extended the metamodel of OCL with the required element in TOCL to generate a metamodel for TOCL.
(3) We formally defined a set of transformation rules between both languages in QVTO.
(4) We implemented the transformation rules.

This paper is structured as follows. After giving and overview of the analysis approach in this introduction, Section 2 details the main contributions of this paper. In particular, we discuss how we defined and created an EBNF grammar and created a parser for TOCL, generated the TOCL metamodel, defined the transformation rules, and implemented them. The validation of the transformation rules is provided in Section 3. The related work is discussed and compared to our approach in Section 4 Section 5 concludes with an overview of the progress made and future work yet to be done.

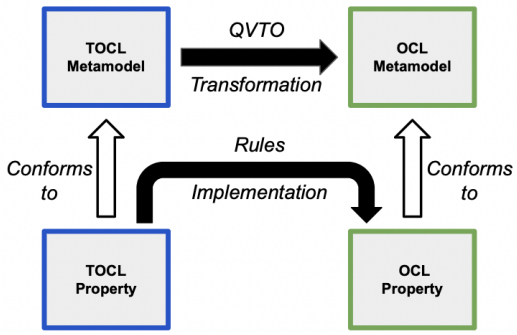

Figure 2: TOCL to OCL Transformation Methodology.

2 TOCL TO OCL TRANSFORMATION

Figure 2 depicts the transformation methodology. The definition of the transformation requires a source and target language. The figure shows that we defined the transformation between the TOCL metamodel (MM), the source language, and the OCL MM, the target language. We formally specified the transformation rules in QVTO. To implement the transformation, we needed an EBNF grammar for TOCL. However, the initial syntax for TOCL was not defined using EBNF format [21]. To facilitate parser generation, we, therefore, created an EBNF grammar for TOCL following the specifications of the original syntax and considering the attribute-grammar static semantics of TOCL. We then used the ANTLR4 parser generator to create a TOCL parser from this grammar. To create the TOCL MM, we used the created EBNF syntax of TOCL and a text-to-model transformation algorithm described in [5] to generate the language's corresponding MM. Finally, we implemented these rules using the generated TOCL parser. In this section, we discuss how we generated the TOCL MM, defined the QVTO transformation rules, and provide implementations of these rules.

2.1 Creating TOCL EBNF Grammar and Parser

We created the EBNF grammar for TOCL by extending the OCL 2.4 grammar (i.e., concrete syntax) and fixing some existing ambiguities. In this process, we also created a new parser for OCL 2.4. The grammar that was used to generate the resulting OCL parser heavily draws from the OCL grammar used by the UML-based Specification Environment (USE). Recall that the analysis approach uses USE in the back-end; therefore, when creating our OCL and TOCL parsers, we emulate its parsing methodology while fixing the ambiguities and defining the grammar in the ANTLR4 format. The finalized ANTLR4 OCL 2.4 grammar was then used as a base for the TOCL grammar.

The original syntax of TOCL in [21] was defined using mathematical notations as opposed to EBNF. We adapted the original syntax by interpreting the mathematical notation while taking into account the intended semantics of the expressions. We also applied an attribute grammar in the form of Java actions to implement the boolean type constraint. The impact of this attribute grammar is

limited as an accompanying object model is not available until the analysis stage.

Listing 1 shows a partial EBNF grammar for TOCL that is used to generate the TOCL parser. In general, each TOCL operator has a corresponding production rule associated with it. Exceptions to this are the until, since, and before operators. This is because these operators are always preceded by either an always or sometime expression, and are therefore treated as specialized alternatives of those expressions. We also defined the @next operator which has been added as an alternative in the operation call production. It is worth noting that we use binaryOperationExp in our production rules because it is the topmost expression in our grammar with alternatives that allow for any OCL expression to be used as an upheld expression for a TOCL operator expression; provided this expression evaluates to a boolean value. Additionally, all the grammar rules are annotated with their corresponding production in the original TOCL syntax.

```
toclExpression[Environment env]: nextExp[$env]
                | alwaysExp[$env]
                | sometimeExp[$env]
                | previousExp[$env]
                | alwaysPastExp[$env]
                | sometimePastExp[$env]
                | nextOperationCallExp[$env]
                ;

nextExp[Environment env]: NEXT e=binaryOperationExp[$env]
{
    if ($e.ctx != null && $e.ast != null) {
        if (!$e.ast.getType().equals("Boolean")) {
            System.out.println("Warning: 'Next' operator
     applied to non-boolean expression");
        }
    }
};

alwaysExp[Environment env]:ALWAYS e1=binaryOperationExp
[$env] (op=(SINCE | UNTIL) e2=binaryOperationExp[$env])?
{
    if ($e1.ctx != null && $e1.ast != null) {
        if (!$e1.ast.getType().equals("Boolean")) {
            System.out.println("Warning: 'Always'
     operator applied to non-boolean expression");
        }
        if ($e2.ctx != null && $e2.ast != null) {
            if (!$e2.ast.getType().equals("Boolean")) {
                System.out.println("Warning: " + $op.text
      + " operator applied to non-boolean expression");
            }
        }
    }
};
```

Listing 1: Partial EBNF grammar for TOCL

Based on the created EBN, we created a parser for TOCL using ANTLR4. As we discuss later, we utilize the ANTLR4 parser to facilitate the implementation of the transformation rules between TOCL and OCL.

2.2 Creating TOCL Metamodel

TOCL includes a set of temporal operators that are based on Linear Temporal Logic (LTL). As TOCL is an extension on OCL, the TOCL MM adds temporal logic constructs to the OCL MM. There exist several algorithms to create an abstract syntax model (i.e., Metamodel) from the concrete syntax of a language (i.e, EBNF). Many of the algorithms aim to achieve a bidirectional transformation between instances of the abstract and concrete syntax. They do this, for example, by employing naming conventions that retain information about the order of properties in the concrete syntax [4]. As our aim is only to transform TOCL to OCL, there is no need for these extra features. We decided to follow a fairly straightforward approach proposed by Anastasakis [5], In the following, we present the steps of the algorithm and apply them to create the TOCL MM.

- **Step 1. The non-terminal symbols of a production rule are represented as MOF classes.** To apply this step, we create a meta class for each of the 12 TOCL operator expressions, as well as one class for the @next alternative for an operation call. Non-terminal symbols for the operator expressions all correspond to the binaryOperation production. We created this production in the EBNF to remove left-recursive ambiguity in the OCL syntax, and to clearly define the precedence between OCL operators. However, the original definition of the TOCL syntax states that TOCL operators can be used on any expression so long as it is of type Boolean. Therefore, we set the abstract form of binaryOperation to be an OclExpression, which has already been defined in the OCL MM, and add an OCL invariant that states OCL expressions associated with TOCL expressions may only be boolean expressions.
- **Step 2. If a terminal symbol is an identifier, it will be represented as a MOF class and an id attribute will be added to the class.** No identifier symbols are present in the EBNF syntax, therefore no additional MOF classes are created in this step.
- **Step 3. The optional terminal symbols of a rule are transformed to boolean attributes of the MOF class to which the non-terminal related to the terminal was transformed.** There are no optional terminals in the TOCL syntax. However, because there is already an isPre attribute defined in the OCL metaclass FeatureCallExp that the TOCLOperationCallExp class inherits from, we also define isNext as a Boolean attribute for TOCLOperationCallExp. In this way we represent both alternatives for the toclOpCallExp production rule.
- **Step 4. The left hand side of a production rule with alternatives is transformed to an abstract MOF class. A number of concrete classes that extend the abstract class are then introduced, to represent each of the alternative choices.** For this rule, a TOCLExpression class is introduced as the parent class for all the alternative classes created in Step 1 of the algorithm
- **Step 5. If the right hand side [of a production rule] does not have any alternative rules or only terminals, an association is generated between the class representing the left hand side of the rule and the class(es) representing the elements of the right hand side.** As discussed in Step 1, the abstract form of the binaryExpression symbol is the OperationCallExp. Here, we create a relationship between the operator expression classes and

this class. The rule states that: The association ends of the classes representing the left hand side of the rule, are non-navigable, while the association ends of the classes on the right hand side are navigable. Multiplicities of the association ends depend on the multiplicity of the symbols in the grammar. Therefore, the type of the relationship established between classes is a composition relationship. The final stipulation of the sule is that: If the multiplicity of an association end is zero to many or one to many, it is always represented by an ordered association end. Because the relationships between the OperationCallExp class and the TOCL operator classes are strictly one to one, we do not take further action.

- **Step 6. If all the right hand side parts of a rule are alternatives and each alternative only has terminals as its elements, the class on the left hand side of the rule is depicted as an enumeration class and each alternative terminal is represented as an enumeration literal.**
  No such rule exists in the TOCL grammar, therefore no enumerations are created.
- **Step 7.Elements of the concrete syntax (e.g. braces, parentheses) are not transformed to any element of the MOF abstract syntax.** We do not transform any of the elements of the concrete syntax.

A diagram of the resulting TOCL MM is shown in Figure 3.

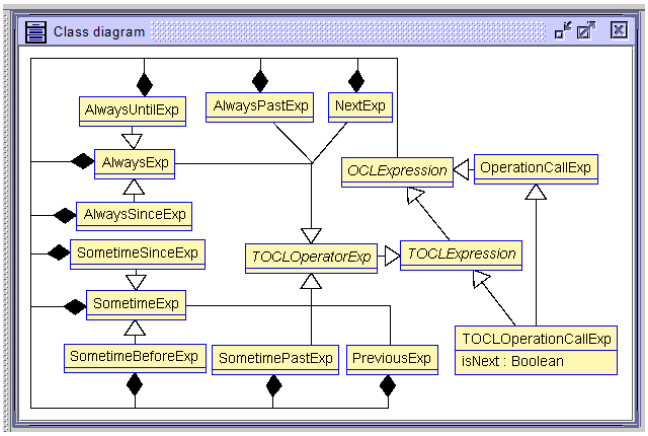

Figure 3: The TOCL MOF metamodel.

2.3 QVTO Transformation Rules

Once we have both the TOCL MM and the OCL MM, we can employ QVTO to formally define the transformation rules between the two languages. As OCL expressions are usually define on class diagram, we include the UML class diagram MM as it is necessary to access UML properties of constructs in OCL MM, i.e, their type. For a similar reason, we include the STM MM (not shown or described in this paper) because the translations of TOCL operator expressions into OCL use constructs defined within a STM.

As an example we explore the QVTO transformation rule for a next operator, as shown in Listing 2.

```
mapping NextExp::NextExp2OCLExpression() :
     OperationCallExp {
   var getSnapshotCall = new OperationCallExp(getSnapshot
     , selfExp.clone(), null);
   var getNextCall := new OperationCallExp(getNext,
     getSnapshotCall, null);
   source := getNextCall;
   referredOperation := snapSat.oclAsType(EObject);
   argument := self.upheldExp.oclAsType(OCLExpression);
}
```

Listing 2: QVTO rule for the TOCL next operator.

Based on the rule above, to create an OCL translation for the next expression, we first instantiate a self expression that represents the self call at the beginning of the expression. Next, we create an OperationCallExp instance corresponding to the getCurrentSnapshot() call. We also set the source of this expression to be the self expression we defined previously. Then, we create another instance of OperationCallExp that represents the call to the getNext() operation, and set its source to be the object representing the call to getCurrentSnapshot(). Finally, we instantiate an OperationCallExp which represents the sat(P) operation call. We set its source to be the previously defined representation of the getNext() call, then return the resulting object.

As can be seen in Listing 2, there exist some OCL query operations such as getCurrentSnapshot() and getNext(). There operations are defined in the context of the STM diagram that the OCL expressions are transformed and evaluated on. The STM MM also includes constraints and operations that facilitate the specification and analysis of temporal properties. The operations (called traversal operations) allow for the traversal of system states. For example, the getNext() operation returns the next state, and the getPost() operation returns all succeeding system states. The constraints ensure the creation of valid scenarios representing system behavior. For instance, we define a constraint that prevents cyclic behavior. There are many other traversal operations and constraints that are defined in the context of the STM that we do not include here.

Another example of a QVTO rule that defines the transformation for the always operator is presented in Listing 3.

```
mapping AlwaysExp::AlwaysExp2OCLExpression() :
     IteratorExp {
    var getSnapshotCall := new OperationCallExp(
     getSnapshot, selfExp.clone(), null);
    var CS := new OCL::Variable(getSnapshotCall);
    var let1 := new LetExp(CS, result);
    var getPostCS := new OperationCallExp(getPost, new
     VariableExp(CS), null);
    var includingCS := new OperationCallExp(including,
     getPostCS, OrderedSet{new VariableExp(CS)});
    var sVar := new OCL::Variable(null);
    result.oclAsType(NamedElement).name := "forAll";
    iterator := sVar;
    source := includingCS;
    body := self.upheldExp.oclAsType(OCLExpression);
}
```

Listing 3: QVTO rule for the TOCL always operator.

We formally define QVTO transformation rules to all 12 TOCL operators in a similar manner.

No.	TOCL Operator	OCL Translation
1	next P	self.getCurrentSnapshot().getNext().sat(P)
2	always P	let CS:Snapshot = self.getCurrentSnapshot() in CS.getPost() ->including(CS)->forAll(s | s.sat(P))
3	always P since Q	let CS:Snapshot = self.getCurrentSnapshot() in let LSQ = CS.getPre()->select(s | s.sat(Q))->asOrderedSet()->first() in if (LSQ.isDefined()) then (CS.getPre()->including(CS) -> LSQ.getPre())->including(LSQ)->forAll(s | s.sat(P)) else CS.getPre()->forAll(s | s.sat(P)) endif
4	always P until Q	let CS:Snapshot = self.getCurrentSnapshot() in let FSQ = CS.getPost()->select(s | s.sat(Q))->asOrderedSet()->first() in if (FSQ.isDefined()) then (FSQ.getPre()-CS.getPre())->forAll(s | s.sat(P) else CS.getPost()->including(CS)->forAll(s | s.sat(P)) endif
5	sometime P	let CS:Snapshot = self.getCurrentSnapshot() in CS.getPost() ->including(CS)->exists(s | s.sat(P))
6	sometime P since Q	let CS:Snapshot = self.getCurrentSnapshot() in let LSQ = CS.getPre()->select(s | s.sat(Q))->asOrderedSet()->first() in if (LSQ.isDefined()) then (CS.getPre()->including(CS) -> LSQ.getPre())->including(LSQ)->exists(s | s.sat(P)) else CS.getPre()->exists(s | s.sat(P)) endif
7	sometime P before Q	let CS:Snapshot = self.getCurrentSnapshot() in let FSQ = CS.getPost()->select(s | s.sat(Q))->asOrderedSet()->first() in if (FSQ.isDefined()) then (FSQ.getPre()-CS.getPre())->exists(s | s.sat(P)) else CS.getPost()->including(CS)->exists(s | s.sat(P)) endif
8	previous P	let CSPrev:Snapshot = self.getCurrentSnapshot().getPrevious() in CSPrev.isDefined() implies CSPrev.sat(P)
9	alwaysPast P	self.getCurrentSnapshot().getPre()->forAll(s | s.sat(P))
10	sometimePast P	self.getCurrentSnapshot().getPre()->exists(s | s.sat(P))
11	P.ω@next(A_1,...,A_n)	let NT = P.getCurrentSnapshot().nextT in NT.oclIsTypeOf(ω) and (let NT = NT.oclAsType(ω) in NT.transitionContext = P and NT.$parameter_1$ = A_1 and … and NT.$parameter_n$ = A_n
12	P.ω@pre(A_1,...,A_n)	let PT = P.getCurrentSnapshot().beforeT in PT.oclIsTypeOf(ω) and (let PT = PT.oclAsType(ω) in PT.transitionContext = P.getPrevious() and PT.$parameter_1$ = A_1[.getPrevious()] and … and PT.$parameter_n$ = A_n[.getPrevious()]

Figure 4: Translation of TOCL operators to OCL.

2.4 Implementation of TOCL to OCL Transformation

We implemented the transformation as a Java code built on the TOCL parser we created using ANTLR4 listeners. The listeners traverse the parse tree of a TOCL expression and produce the corresponding OCL expression.

Two reasons encouraged us to do this. First, it is more efficient to use the existing functionality of the generated parser than it would be to create external additional code that must communicate with the parser to perform the transformation. Second, based on our experience, the integration of the transformation code into an analysis tool will be significantly more straightforward. As such, the transformation from TOCL to OCL is implemented overriding generated listener methods of the ANTLR4 TOCL parser. To use these rules we walk the parse tree generated by the TOCL parser.

To transform TOCL expressions, we defined translations for TOCL operators to OCL 4. To create these translations, we utilized some important query operations. Method getCurrentSnapshot() gets the snapshot associated with an object in the "current state" or the state where the expression is being evaluated. The getNext() and getPrevious() operations when applied to a snapshot get the snapshot in the next state and previous state, respectively. When applied to an object, getNext() and getPrevious() get the corresponding object in the next and previous state, respectively. Similarly, operations getPre() and getPost() get the collection of all snapshots before and after a given snapshot in a sequence of states. Additionally, the sat(expr) operation evaluates whether a snapshot satisfies a given boolean expression expr. For example, the next P operator gets the current snapshot of an object, then gets the next snapshot, and finally checks if that next snapshot satisfies expression P. If it is satisfied, the expression evaluates to true.

Note the use of the *sat*() operation in the implementations of the operator in the OCL expressions. The operation can be implemented as a 'built-in' operation in USE and interpreted in Java in the backend. A drawback of this approach is that the *sat*() operation will not be reusable by other OCL tools. A better approach is to provide an OCL operation that can be imported and used by any OCL tool.

After defining these translations, we created rules for them using listeners. These rules take the OCL translation of each TOCL operator and simply replace the appropriate parts with the children of the corresponding parse tree node. For instance, when translating an always expression, we replace P in '.sat(P)' with the child at index 1, which corresponds to the expression to be evaluated. We do this using the getOCL(ParseTree ctx) operation, which gets the text associated with node ctx. In addition, we also keep track of the original TOCL version of the expression in the variable origTOCL. After the translation, we push both the translated OCL expression and the original TOCL expression into a stack, which will be popped at the end of traversing the entire parse tree. Figure 5 shows an example of the implementation of the the TOCL always expression using an ANTLR4 listener.

```
String origTocl = tokens.getText(ctx);
oclTranslation = "let CS:Snapshot = self.getCurrentSnapshot()
     in CS.getPost()->including(CS)->forAll(s | s.sat("+getOCL(ctx.getChild(1))+"))";
stack.push(oclTranslation);
stack.push(origTocl);
```

Figure 5: ANTLR4 listener implementation of always rule.

Once the listener has visited every node within a tree, it finally visits the root where it creates the string that is the result of the OCL translation. Since TOCL is an extension of OCL, many of the constructs are the same and should be directly mapped in a translation. Thus, first, we store the token stream in the variable tokens. The operation getText(ctx) concatenates the lexemes of the tokens that descend from the node represented by ctx. Next, we pop every translated OCL expression and the corresponding original TOCL expression from the aforementioned stack. Then, we replace every instance of the original TOCL expression with its equivalent OCL translation using the replace(CharSequence target, CharSequence replacement) operation. Finally, after replacing the translated TOCL expressions, we use the setOCL(ParseTree ctx, String s) operation to set this finished translated OCL to the root of the parse tree.

3 VALIDATION OF THE TRANSFORMATION RULES

For our TOCL to OCL translator to be useful, the translations of the TOCL operators must work as intended. This section demonstrates that the generated OCL expressions, from TOCL transformation, are valid constraints in their corresponding STM class diagrams.

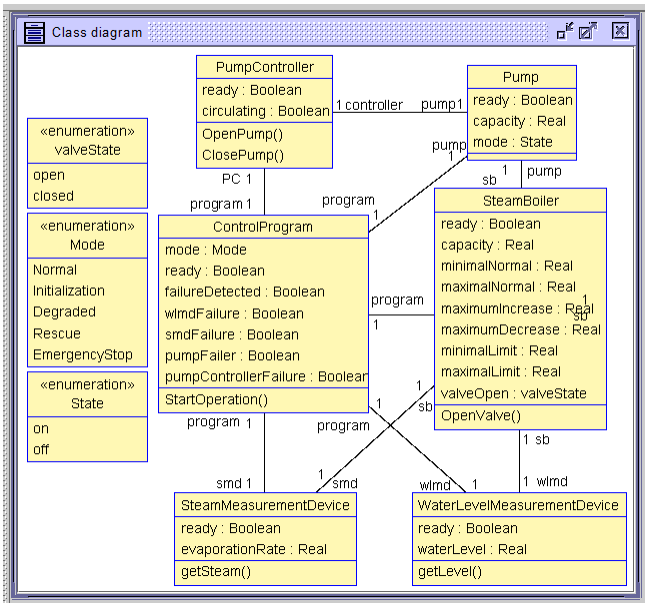

Figure 6: The class diagram for SBCS.

This section also puts the TOCL to OCL transformation back into the context of the analysis approach in Figure 1. To show case the validity of our transformation, we applied the translation on the Steam Boiler Control System (SBCS), shown in Figure 6. The SBCS specification problem Abrial et al. [1] has been used extensively to assess the effectiveness of many software specification and verification approaches.

By applying the algorithm to produce the STM of the class diagram depicted in Figure 6) (e.g., Step 1 of the analysis approach in Figure 1), we obtain the diagram in Figure 7.

We manually created an instance of the STM of the SBCS in Figure 7 to show the correctness of our implementation of the TOCL to OCL transformation rules and the validity of the obtained OCL expressions. Note that this figure shows one possible execution of the system. Using this figure, we evaluated all the TOCL operators translation. The informal validation of the transformation is done as follows. First, we create a TOCL temporal property and manually evaluate it based on the figure and indicated the expected evaluation result. In the second steps, we automatically generate the corresponding OCL expression using the implementation of our transformation. Finally, we use USE to evaluate the generated OCL expression on the provided scenario in Figure 8. We show that the expected evaluation of the expression matches the result produced by USE. We follow this procedure for the evaluation of all TOCL operators.

Next is a unary operator of the format next P that evaluates whether the expression P is true in the next state. The expressions in figure 9 are evaluated on the object diagram in fig 8 and produce the expected result by USE.

Always is a unary operator of the format always P that evaluates whether the expression P is true in all future states including the present one. In a similar way, the expressions in figure 10 are

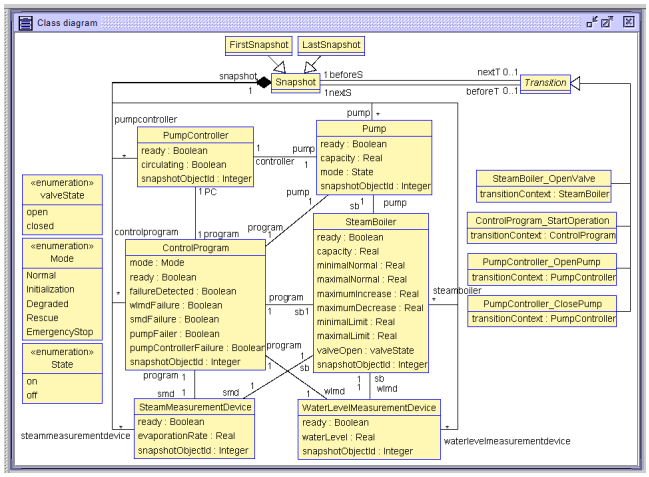

Figure 7: The STM for the SBCS class diagram in Fig. 6

evaluated on the object diagram in fig 8 and produce the expected result by USE.

Always … since is a binary operator of the format always P since Q that evaluates whether the expression P is true in all past states, including the current state, that occur after a state that satisfies expression Q. If no past state satisfies Q, then P should be true in all past states. In a similar way, the expressions in figure 11 are evaluated on the object diagram in fig 8 and produce the expected result by USE. The first expression is an example of when Q is defined while the second one is an example of when Q is not defined.

The validation of the correctness of the remaining operators in Figure 4 is performed in a similar manner.

4 RELATED WORK

Enriching OCL with temporal logic is a research objective that has been tackled by many researchers who have produced a variety of approaches. Some approaches, such as the one presented in [19], extend OCL with temporal operators from logic systems such as LTL. Others propose a pattern-based specification approach that is geared towards providing an accessible way for designers to specify system properties without being highly familiar with temporal logic systems [10, 15, 17] Previous works have also proposed a semantic foundation for temporal OCL extensions without defining a concrete syntax for specific temporal expressions. For instance, the approach in [7] proposes an extension of OCL based on mu-calculus that provides semantics that could be used in conjunction with an appropriate syntax for temporal expressions. Researchers in [12] present an OCL extension based on CCTL semantics and define a UML profile for the specification of temporal constraints on statechart diagrams. In this section, we compare these approaches to our work.

Temporal properties in OCL are covered in a number of publications. LTL operators for OCL were defined in [21], but an implementation was left out. The work presented in [12] focuses on the integration of time bounds in connection with temporal constructs and state machines which "enables modelers to specify behavioral

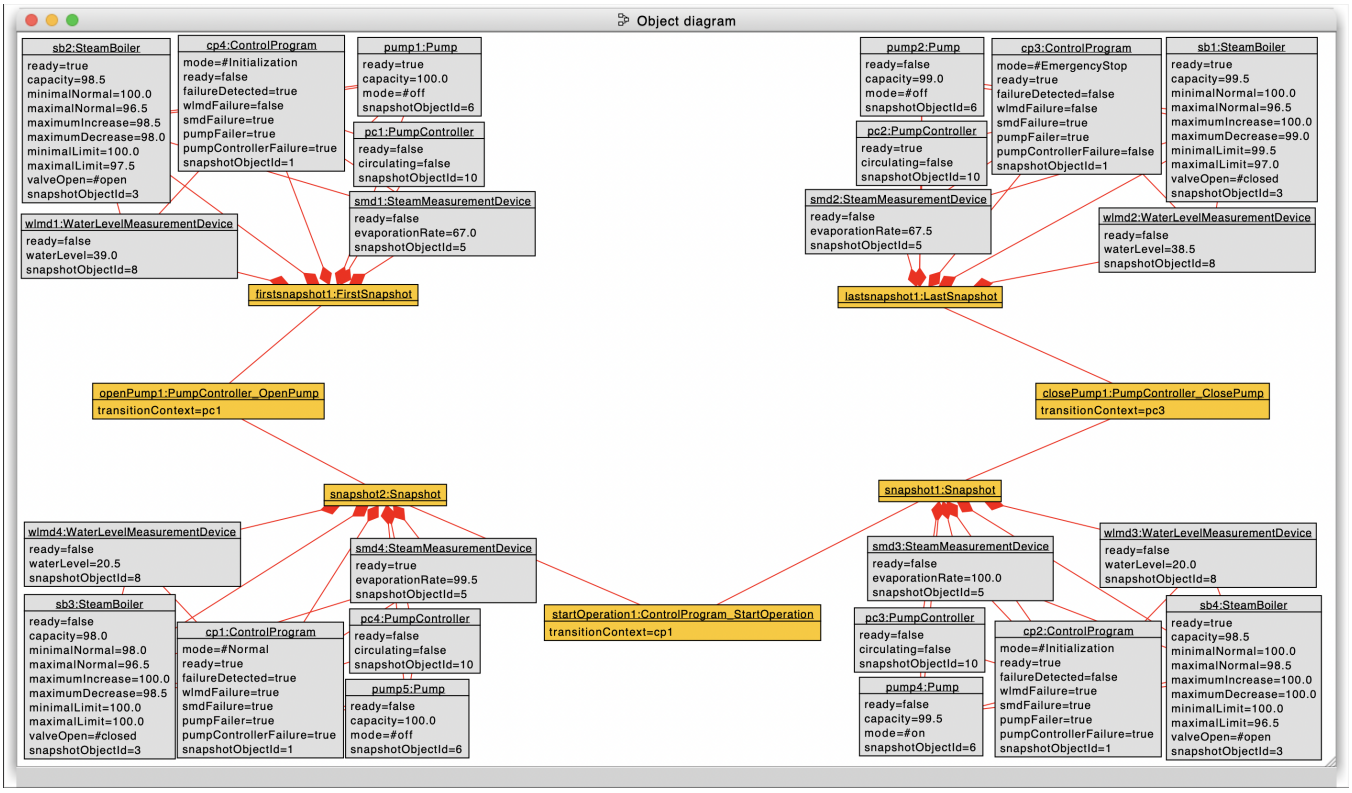

Figure 8: Scenario: An instance of the STM SBCS in Fig. 7

Expression in TOCL on SBCS Class Diagram	Expression in OCL on STM of the SBCS	Expected Evaluation Result
next sp2.capacity = 98.0	sp2.getNext().capacity = 98.0	True
next sp2.capacity = 100.0	sp2.getNext().capacity = 98.0	False

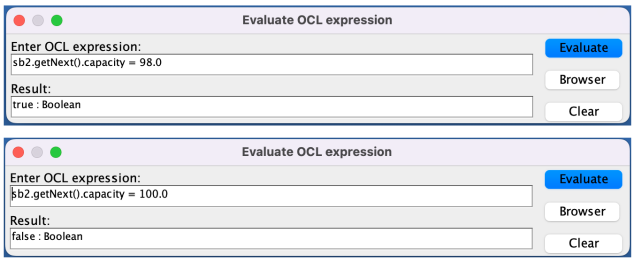

Figure 9: Evaluation of next expressions.

Expression in TOCL on SBCS Class Diagram	Expression in OCL on STM of the SBCS	Expected Evaluation Result
always pc2.circulating = false	let CS:Snapshot = pumpcontroller3.getCurrentSnapshot() in CS.getPost()->including(CS)->forAll(s | pc2.atSnapshots(s).circulating= false)	True
always pump1.mode = #off	let CS:Snapshot = pump1.getCurrentSnapshot() in CS.getPost()->including(CS)->forAll(s | pump1.atSnapshots(s).mode=#off)	False

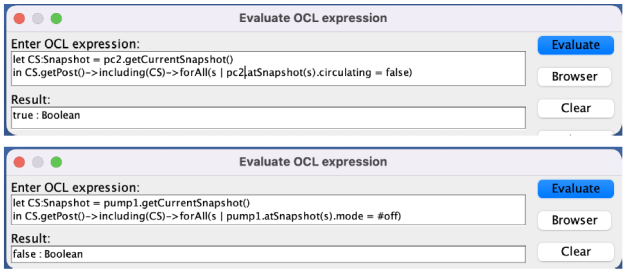

Figure 10: Evaluation of always expressions.

state-oriented real-time constraints". An approach specifying temporal properties without the need to know LTL or CTL but also a detailed comparison of different temporal extensions to OCL is described in [15]. The authors show an implementation on top of Elipse MDT OCL. In [19] OCL is extended in in simlar way to [21] to define Linear Temporal OCL (LT-OCL) formulas over states.

An approach that also uses translation to plain UML and OCL is the filmstrip model [13, 14]. Similar to our snapshot transition model, UML models enriched with TOCL expressions are translated into a so-called filmstrip model that is verified by the USE model finder. The translation is also realized as a USE plugin to hide the details of the underlying filmstrip model from the modeler.

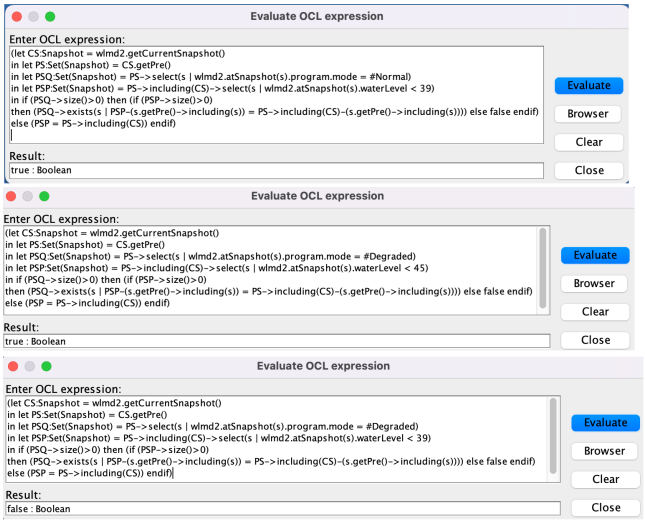

Figure 11: Evaluation of always … since expressions.

Compared to [14] the work presented here supports more temporal operators.

Work that covers verification of temporal properties using UML and OCL extensions, in contrast to the aforementioned one and our approach, is described in [6, 9, 17]. In [6] a tool chain is presented that the authors named the MADES approach. It combines several well-known technologies, like a subset of the UML, the MARTE profile and a verification tool to be able to verify embedded systems. [9] add a temporal extension to OCL based on process states. These temporal constraints are afterwards translated to Petri nets to be verified. ProMoBox[17] is a framework to support verifying temporal properties in the context of domain specific modeling. The authors state that their generic language can be integrated into domain specific modeling languages to support temporal properties.

5 CONCLUSION

When software designers generate code from their software models, any errors follow into the result, except it will be more expensive to fix. The analysis approach that was discussed aims to specify and verify temporal properties in UML-base notations, techniques, and tools.

The contributions laid out in this paper address the second step of our class diagram analysis method. We created TOCL and OCL grammars that include all the rules that define their syntax. Using ANTLR, we generated lexers and parsers for these grammars that could create parse trees out of expressions. Additionally, we created a TOCL metamodel by extending the OCL metamodel with TOCL constructs and defined rules that translate TOCL metamodel components to OCL. Finally, we created a listener-based TOCL to OCL translator that uses the parse trees created by the TOCL parser to create the translation.

Creating a way of translating TOCL to OCL will allow us to analyze UML class diagrams based on the lifetime of a system by allowing software developers to define temporal properties more easily using TOCL. Furthermore, the creation of the TOCL parser and TOCL metamodel will help advance the Model-Driven Engineering field by encouraging and providing the use of temporal logic in systems.

Now that we have created a way to transform UML class diagrams to Snapshot Transition Models, to facilitate their analysis, and to translate TOCL expressions to OCL, we plan to focus on the analysis of class diagrams in our future work. The UML-based Specification Environment has a model validator we plan to use to conduct our analyses. Additionally, we will investigate how to optimize this analysis, such as by manipulating the parameters of the analysis. Afterward, we will work on creating a sequence diagram that displays errors found in the analysis to users. Eventually, we would like to package our work into a tool for software designers to use to improve their models.

ACKNOWLEDGMENTS

This work was partially supported by NSF under grant award 1950416.

REFERENCES

[1] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack (Eds.). 1996. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995)*. Lecture Notes in Computer Science, Vol. 1165. Springer. https://doi.org/10.1007/BFb0027227

[2] Mustafa Al-Lail, Ramadan Abdunabi, Robert B. France, and Indrakshi Ray. 2013. An Approach to Analyzing Temporal Properties in UML Class Models. In *Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2013, co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, October 1st, 2013 (CEUR Workshop Proceedings, Vol. 1069)*, Frédéric Boulanger, Michalis Famelis, and Daniel Ratiu (Eds.). CEUR-WS.org, 77–86. http://ceur-ws.org/Vol-1069/11-paper.pdf

[3] Mustafa Al-Lail, Ramadan Abdunabi, Robert B. France, and Indrakshi Ray. 2013. Rigorous Analysis of Temporal Access Control Properties in Mobile Systems. In *2013 18th International Conference on Engineering of Complex Computer Systems, Singapore, July 17-19, 2013*. IEEE Computer Society, 246–251. https://doi.org/10.1109/ICECCS.2013.43

[4] Marcus Alanen, Ivan Porres, Turku Centre, and Computer Science. 2003. *A Relation Between Context-Free Grammars and Meta Object Facility Metamodels*. Technical Report.

[5] Kyriakos Anastasakis. 2009. *A Model Driven Approach for the Automated Analysis of UML Class Diagrams*. Ph. D. Dissertation. School of Computer Science.

[6] Luciano Baresi, Gundula Blohm, Dimitrios S. Kolovos, Nicholas Drivalos Matragkas, Alfredo Motta, Richard F. Paige, Alek Radjenovic, and Matteo Rossi. 2015. Formal verification and validation of embedded systems: the UML-based MADES approach. *Softw. Syst. Model.* 14, 1 (2015), 343–363. https://doi.org/10.1007/s10270-013-0330-z

[7] Julian C. Bradfield, Juliana Küster Filipe, and Perdita Stevens. 2002. Enriching OCL Using Observational Mu-Calculus. In *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2306)*, Ralf-Detlef Kutsche and Herbert Weber (Eds.). Springer, 203–217. https://doi.org/10.1007/3-540-45923-5_14

[8] Edmund M. Clarke and E. Allen Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981 (Lecture Notes in Computer Science, Vol. 131)*, Dexter Kozen (Ed.). Springer, 52–71. https://doi.org/10.1007/BFb0025774

[9] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, Xavier Thirioux, and François Vernadat. 2007. A Property-Driven Approach to Formal Verification of Process Models. In *Enterprise Information Systems, 9th International Conference, ICEIS 2007, Funchal, Madeira, Portugal, June 12-16, 2007, Revised Selected Papers (Lecture Notes in Business Information Processing, Vol. 12)*, Joaquim Filipe, José Cordeiro, and Jorge Cardoso (Eds.). Springer, 286–300. https://doi.org/10.1007/978-3-540-88710-2_23

[10] Frédéric Dadeau, Elizabeta Fourneret, and Abir Bouchelaghem. 2019. Temporal property patterns for model-based testing from UML/OCL. *Softw. Syst. Model.* 18, 2 (2019), 865–888. https://doi.org/10.1007/s10270-017-0635-4

[11] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, Barry W. Boehm, David Garlan, and Jeff Kramer (Eds.). ACM, 411–420. https://doi.org/10.1145/302405.302672

[12] Stephan Flake and Wolfgang Müller. 2003. Formal semantics of static and temporal state-oriented OCL constraints. *Softw. Syst. Model.* 2, 3 (2003), 164–186. https://doi.org/10.1007/s10270-003-0026-x

[13] Martin Gogolla, Lars Hamann, Frank Hilken, Mirco Kuhlmann, and Robert B. France. 2014. From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In *Modellierung 2014, 19.-21. März 2014, Wien, Österreich (LNI, Vol. P-225)*, Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer (Eds.). GI, 273–288. https://dl.gi.de/20.500.12116/17056

[14] Frank Hilken and Martin Gogolla. 2016. Verifying Linear Temporal Logic Properties in UML/OCL Class Diagrams Using Filmstripping. In *2016 Euromicro Conference on Digital System Design, DSD 2016, Limassol, Cyprus, August 31 - September 2, 2016*, Paris Kitsos (Ed.). IEEE Computer Society, 708–713. https://doi.org/10.1109/DSD.2016.42

[15] Bilal Kanso and Safouan Taha. 2012. Temporal Constraint Support for OCL. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7745)*, Krzysztof Czarnecki and Görel Hedin (Eds.). Springer, 83–103. https://doi.org/10.1007/978-3-642-36089-3_6

[16] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. 2011. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *Objects, Models, Components, Patterns - 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6705)*, Judith Bishop and Antonio Vallecillo (Eds.). Springer, 290–306. https://doi.org/10.1007/978-3-642-21952-8_21

[17] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. 2014. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8706)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer, 1–20. https://doi.org/10.1007/978-3-319-11245-9_1

[18] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32

[19] Michael Soden and Hajo Eichler. 2009. Temporal Extensions of OCL Revisited. In *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5562)*, Richard F. Paige, Alan Hartman, and Arend Rensink (Eds.). Springer, 190–205. https://doi.org/10.1007/978-3-642-02674-4_14

[20] Lijun Yu, Robert B. France, and Indrakshi Ray. 2008. Scenario-Based Static Analysis of UML Class Models. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5301)*, Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter (Eds.). Springer, 234–248. https://doi.org/10.1007/978-3-540-87875-9_17

[21] Paul Ziemann and Martin Gogolla. 2003. OCL Extended with Temporal Logic. In *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers (Lecture Notes in Computer Science, Vol. 2890)*, Manfred Broy and Alexandre V. Zamulin (Eds.). Springer, 351–357. https://doi.org/10.1007/978-3-540-39866-0_35