Verifying OCL pre/post condition using Cyclone

YueLou Ding*,†, Hao Wu*

Computer Science Department, Maynooth University, Ireland

Abstract

In this paper, we present an idea about verifying OCL pre/post condition. This idea works by mapping pre/post conditions to a graph specification that can be verified by a tool called Cyclone. The purpose of this paper is to show a developing idea rather than to present a more complete approach. We illustrate this idea by using an example, discuss our work-in-progress and outline challenges to be tackled in the future.

Keywords

OCL, Pre/Postcondition, Verification, Cyclone

1. Introduction

The Unified Modeling Language (UML) is a widely adopted standard for modeling complex systems due to its intuitive nature [1]. However, being described in natural language but not in formal language, it lacks rigor, which may cause inaccuracy and misconception [2]. The Object Constraint Language (OCL) complements UML by providing a formal textual notation to specify constraints or rules over UML models [3]. Pre/post conditions in OCL are crucial as they define the conditions that specific elements in a UML model must satisfy before and after an operation execution [4]. Despite their significance, OCL is a declarative language grounded in first-order logic. This characteristic can pose quite a challenge when reasoning about the correctness of a model [5]. Recently, a number of approaches have been proposed to tackle this challenge [6, 7, 8, 9]. They primarily focus on translating OCL pre/post conditions into different formal languages and verifying these conditions using different tools [10, 11, 12, 13].

Several works have proposed mappings from OCL pre/post conditions specified on UML classes into different formalisms, including RSL [6], PVS [7], K semantics [8], RM2PT[9], and FoCaLiZe [10]. These approaches utilize automated provers to check the correctness of OCL constraints. For example, A. Cunha et al. present a bidirectional model transformation between Alloy and UML class diagrams annotated with OCL, offering greater availability and usability [11]. T. Nguyen et al. introduce TC4MT, a high-level visual declarative language for specifying model transformation requirements based on a contract-based approach [12]. Przigoda et al. propose a technique to assist users to transform OCL pre/post conditions to bit vectors, and use satisfiability (SAT) solvers for automatic checking [13]. However, these tools have three main drawbacks (1) They lack the capability to visualise the trace or verification results. This might not be very helpful when users try to understand why and how the verification results do not match their specified OCL operations, in particular, when a model has complex operations. (2) They are typically not well maintained after a while. This imposes significant challenges on other researchers who would like to replicate or have access to their tools. (3) The mappings from OCL contracts to a particular formalism may seem daunting for regular users who do not have enough formal background.

In this paper, we present an idea about verifying OCL pre/post conditions. This works by mapping them to a graph specification that can be efficiently verified by a tool called Cyclone [14]. Our mapping

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10-13, 2025

*Corresponding author.

†YueLou Ding is supported by China Scholarship Council and the Department of Further and Higher Education, Research, Innovation and Science.

yuelou.ding.2025@mumail.ie (Y. Ding); haowu@cs.nuim.ie (H. Wu)
https://yueloud.github.io/ (Y. Ding); https://classicwuhao.github.io/ (H. Wu)
0009-0009-9835-1485 (Y. Ding); 0000-0001-5010-4746 (H. Wu)

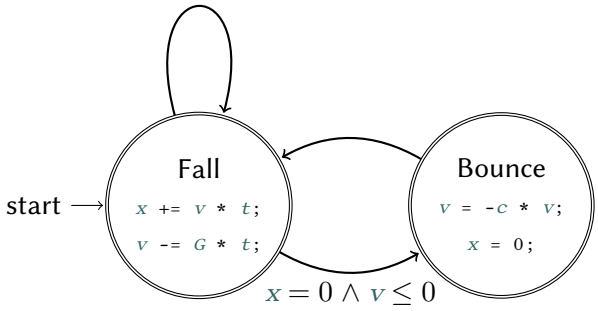

Figure 1: A transition graph for a bouncing ball.

gives us three advantages (1) The graph representation can be easily visualised when performing detailed analysis. (2) The tool Cyclone is well-maintained and can be accessed through the online playground without installing any additional packages locally. (3) The mapping is very straightforward and does not require sophisticated knowledge about a formalism. Though this idea is still under development, we believe that it has the potential to be extended to cover a much wider range of UML models due to its simplicity and clarity.

2. Background

Our idea is to represent OCL pre/post conditions into a graph representation (namely, a transition diagram), then use a tool called Cyclone to help us to automatically verify them [14, 15]. Cyclone is a tool that allows users to represent their verification tasks into a graph. It is written in Java and consists of more than 100k lines of code including build scripts, test cases, configurations, and other related projects. It has about 180 grammar rules for the Cyclone specification language and uses ANTLR to generate lexers and parsers. The type checker ensures the type safety of a specification, and produces relevant error messages if there are any semantic errors in a specification.

It uses a *state matrix* as an intermediate representation (IR) to capture all necessary information needed from the front-end of the Cyclone compiler. Once this IR is produced, we use a novel algorithm to generate a set of graph conditions with respect to the bounds chosen by users. These graph conditions can then be efficiently verified by an off-the-shelf SMT solver[1]. Finally, the trace generator produces a *trace* or *counter-example* for the specification (depending on the mode it is running as).

Currently, Cyclone supports six different data types: int, bool, real, bv (bit-vector), enum and record. The first four data types are primitive data types, a set of pre-defined constants can be declared as enum and record data type to represent complex objects.

We now use a bouncing example to illustrate some of the basic features of Cyclone. Figure 1 plots a transition graph for bouncing ball model. Listing 1 shows the corresponding Cyclone specification. The two states [2](`Fall` and `Bounce`) in Figure 1 are directly mapped to the nodes and each transition is mapped to an edge. The calculation of the position of the ball is shown in each node. For example, the `Bounce` node describes that when the ball hits ground (x=0), it loses some energy and changes direction to bounce back. For this simple model, we use Cyclone to verify whether the position of the ball always stays positive. In other words, the ball is either in the air or on the ground. This property is described using a graph invariant [3] (Line 22 in Listing 1) in the specification. To verify this specification, Cyclone uses a new algorithm that effectively reduces the transition graph along with its invariants into a set of formulas that can be solved by an SMT solver [16].

[1]Currently, Cyclone uses Z3 [16] as its default solver. Work is in progress to also support the CVC5 solver.
[2]The computation could terminate at any of the two nodes.
[3]When a specification contains at least one graph invariant, Cyclone switches to the counter-example discovery mode.

Listing 1: A Cyclone specification modeling a transition graph of a bouncing ball hybrid system depicted in Figure 1. The variable block (Line 2–6) contains all necessary variables for computing the ball's position x, which can never be below 0 (Line 22). The node and edge blocks (Line 8–20) model different states of the ball. The goal block (Line 24–26) instructs Cyclone to explore a counter-example within a set of given bounds. Note that t is the discretisation unit for time.

```
1  graph Bouncing_Ball {
2      real x where x >= 0;            // position of the ball
3      real v;                         // velocity of the ball
4      real t where t >= 0;            // time sequence
5      const real G = 9.81;            // gravitational force
6      real c where c >= 0 && c <= 1; // constant (energy loss)
7
8      normal start final node Fall {
9          x += v * t;
10         v -= G * t;
11     }
12
13     normal final node Bounce {
14         v = -c * v;
15         x = 0;
16     }
17
18     edge { Fall -> Fall }
19     edge { Fall -> Bounce, where x == 0 && v <= 0; } // the ball starts to bounce back when it reaches ground
           and its velocity becomes negative.
20     edge { Bounce -> Fall }
21
22     invariant inv { x >= 0; } // position of the ball is never < 0
23
24     goal { // defines different bounds for Cyclone to explore.
25         check for 2,3,4,5
26     }
27 }
```

3. Mapping OCL Pre/Post conditons to Cyclone

This section demonstrates how one can map OCL pre/post conditions to Cyclone. For illustration purpose, we use a stack as an example to demonstrate such mapping. A *stack* is a Last-In-First-Out (LIFO) structure that is commonly used in many algorithms[17]. It has two main operations: $push$ and pop. The $push$ operation simply pushes an element on top of a stack and pop removes the element from top of a stack[4]. We depict this simple structure using a UML class diagram shown in Figure 2. This class diagram also defines a list of OCL pre/post conditions over its operation. For example, to prevent a stack from being overflow, before executing a push operation, we must ensure that a stack is currently not full. Such a condition is captured by an OCL pre condition. Every time a $push$ or pop operation is successfully executed, the $size$ of the stack is increased or decreased by 1 but never goes below 0 or beyond its pre-defined $capacity$.

Since one can issue a sequence of $push$ and pop operations to a stack, we can simply use a transition diagram to capture such a sequence. This transition diagram is depicted in Figure 3. Since a sequence of operations could stop (terminate) at any of the $push$ and pop operations, both are marked termination nodes in Figure 3.

To observe how a sequence of $push$ and pop operations affects a stack, we define a *configuration* (C) of a stack as follows:

$$C = \langle N, E, S \rangle$$

where N specifies a unique object identifier, E denotes a list of elements inside a stack and S represents current size of a stack. Hence, we now can show the configurations of a stack every time a $push$ or pop operation is applied. For example, the following configuration shows the 2 pushes and 1 pop operation.

[4]Since *pop* can remove all identical elements in the stack, we stipulate that all elements in the stack should be unique.

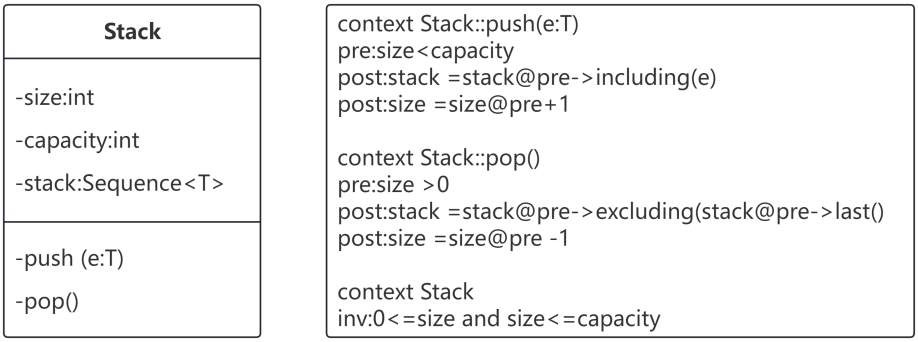

Figure 2: A UML class diagram with OCL pre/postconditions for stack operations.

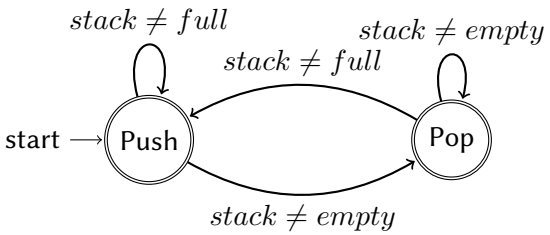

Figure 3: A transition diagram that captures a sequence of $push$ and pop operations.

$$\langle O_{stack}, [], 0\rangle \xrightarrow{push(a)} \langle O_{stack}, [a], 1\rangle \xrightarrow{push(b)} \langle O_{stack}, [ba], 2\rangle \xrightarrow{pop()} \langle O_{stack}, [a], 1\rangle$$

Assuming that the capacity of O_{stack} is 5, the configurations $\langle O_{stack}, [aa], 1\rangle$ and $\langle O_{stack}, [bbbbba], 6\rangle$ are both invalid due to incorrect size and exceeded capacity, respectively. It is obvious that the invariant for our configuration is: $0 \leq S \leq capacity$. As long as this invariant holds, our stack will never be underflow or overflow.

Thus, to be able to verify no such invalid configuration will occur in our $stack$ example, we map our transition diagram along with the OCL pre/post condition to our Cyclone specification. The mapping is straightforward and the rules are as follows:

1. Every operation ($push$ and pop) is mapped to a *node*.
2. Every precondition of an operation is mapped to an *edge* (conditional transition).
3. Every postcondition of an operation is mapped to a set of detailed computational instructions.
4. A class invariant is mapped to a *graph invariant*.

By following these rules, the complete Cyclone specification mapped from Figure 2 and Figure 3 is shown in List 2. Now, we can query Cyclone to check whether an invalid configuration could occur after a certain number of $push$ and pop operations. In this case, we ask Cyclone to perform bounded checking upto 11[5] (Line 28 in Listing 2) possible operations (approximately 2^{11} all possible operations). We choose 11 here for illustration purpose only and users has a freedom to choose an upper bound.

To encode a collection of objects that can be pushed onto a stack, we use a bit vector (bv) to represent a stack. This means that whenever there is an object to be pushed onto a stack, we shift right 1 bit and add 1 to our bit vector (Line 9 in Listing 3). When a stack pops off an object, we simply remove the last bit and shift left bit vector by 1 bit (Line 14 in Listing 3). In the meanwhile, we keep track of the size of a stack. Thus, the following shows a valid configuration (a capacity of 5) for our stack encoded using a bit vector.

$$\langle O_{stack}, [00000], 0\rangle \xrightarrow{push()} \langle O_{stack}, [00001], 1\rangle \xrightarrow{push()} \langle O_{stack}, [00011], 2\rangle \xrightarrow{pop()} \langle O_{stack}, [00001], 1\rangle$$

[5]Please note that a single transition consists of 2 nodes. Thus, a path length of 10 has 11 nodes.

The graph invariant here is shown in Line 23 in Listing 3. This invariant is considered as *safety* property for our stack. That is, no matter how many $push$ and pop operations are executed (following the transition diagram in Figure 3), this property should always hold. In Cyclone, whenever there is a user-defined invariant, it switches to counter-example finding mode. In this case, Cyclone does not find any counter-example that breaks this invariant within 2^{11} possible $push$ and pop operations. This example can be accessed from the link here and checked through our online playground.

Listing 2: A Cyclone specification modelling a transition graph of a stack depicted in Figure 3. The variable block (Line 2–6) contains all necessary variables for the stack, the size of which can never be below 0 or beyond its capacity (Line 23). The node and edge blocks (Line 8–21) model different states of the stack. The goal block (Line 25–29) instructs Cyclone to explore a counter-example within a set of given bounds. Note that $size$ denotes the current size of the stack.

```
1  graph Stack{
2      const int capacity=10; /* capacity of stack */
3      bv[capacity] stack;    /* our stack */
4      const bv[capacity] EMPTY=0b0;   /* situation of the stack being empty */
5      const bv[capacity] FULL=~EMPTY; /* situation of the stack being full */
6      int size=0; /* size refers to the current number of elements in the stack */
7
8      start final node Push{
9          stack = (stack << 0b1)+0b1;
10         size++;
11     }   /* when it comes to node Push, an element is pushed into the stack */
12
13     final node Pop{
14         stack = (stack - 0b1)>>0b1;
15         size--;
16     }   /* when it comes to node Pop, an element is popped out of the stack */
17
18     edge { Push -> Push where stack !=FULL;}
19     edge { Pop -> Push where stack !=FULL;} /* if Push is the target node, the transition can happen only when
           stack is not full */
20     edge { Push -> Pop where stack !=EMPTY;}
21     edge { Pop -> Pop where stack !=EMPTY;} /* if Pop is the target node, the transition can happen only when
           stack is not empty */
22
23     invariant safety { size>=0 && size<=capacity;}/* prove our stack can never overflow or underflow */
24
25     goal{
26         assert (initial(stack)==EMPTY); /* stack is initially empty. */
27
28         check upto 10 /* bounded checking upto 11 push(es)&pop(s) regardless of the order of a call sequence */
29     }
30 }
```

4. Work in Progress

Based on this idea, we also map a queue[6] structure from OCL to Cyclone and verify it. Currently, we are collecting OCL pre/post conditions from literature and hope to translate them into Cyclone for verification. More importantly, our main objective is to propose a new and powerful approach to map UML state machine to Cyclone for automated verification. We have already been investigating the literature on different approaches to verification of UML state machine [18]. Interestingly, a large number of techniques and approaches have been proposed, but none of them work very well in terms of availability, usability, and scalability. This leaves a big gap between the verification and modeling communities. The challenges here are (1) Most of the techniques work well for toy examples and do not scale well to large and complex models. (2) Features like concurrency (fork/join) and sub state machines are often missed by tools. (3) Mappings are not always straightforward and often require

[6]The queue example can be accessed here

a significant amount of knowledge about a particular formalism. Hence, to tackle these challenges, we are investigating a systematic approach that is built on our idea presented in this paper, and hope to close/minimize the gap between verification and UML modeling by harnessing the simplicity and power of new verification tool such as Cyclone.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] O. OMG, Unified Modeling Language (OMG UML) version 2.5.1, Object Management Group (2017).

[2] A. Nugroho, M. R. Chaudron, A survey into the rigor of UML use and its perceived impact on quality and productivity, in: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008, pp. 90–99.

[3] D. OMG, OMG Object Constraint Language (OCL), version 2.3. 1, 2012.

[4] J. Cabot, M. Gogolla, Object constraint language (OCL): a definitive guide, in: International school on formal methods for the design of computer, communication and software systems, Springer, 2012, pp. 58–90.

[5] A. Brucker, J. Doser, B. Wolff, Semantic issues of OCL: Past, present, and future, Electronic Communications of the EASST 5 (2006).

[6] N. Debnath, A. Funes, A. Dasso, G. Montejano, D. Riesco, R. Uzal, Integrating OCL expressions into RSL specifications, in: 2007 IEEE International Conference on Electro/Information Technology, 2007, pp. 158–162. doi:10.1109/EIT.2007.4374543.

[7] L. A. Rahim, Transforming OCL to PVS: Using theorem proving support for analysing model constraints (2007).

[8] A. Arusoaie, D. Lucanu, V. Rusu, Towards a K semantics for OCL, Electronic Notes in Theoretical Computer Science 304 (2014) 81–96.

[9] Y. Yang, X. Li, Z. Liu, W. Ke, RM2PT: a tool for automated prototype generation from requirements model, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE, 2019, pp. 59–62.

[10] M. Abbas, Using FoCaLiZe to check OCL constraints on UML classes, in: International conference on information technology for organization development, IT4OD 2014, conference proceedings, 2014, pp. 31–38.

[11] A. Cunha, A. Garis, D. Riesco, Translating between Alloy specifications and UML class diagrams annotated with OCL, Software & Systems Modeling 14 (2015) 5–25.

[12] D.-H. Dang, T.-H. Nguyen, A contract-based specification method for model transformations, VNU Journal of Science: Computer Science and Communication Engineering 39 (2023).

[13] N. Przigoda, M. Soeken, R. Wille, R. Drechsler, Verifying the structure and behavior in UML/OCL models using satisfiability solvers, IET Cyber-Physical Systems: Theory & Applications 1 (2016) 49–59.

[14] H. Wu, T. Flinkow, D. Méry, Cyclone: A new tool for verifying/testing graph-based structures: Tool paper, in: International Conference on Tests and Proofs, Springer, 2024, pp. 107–124.

[15] H. Wu, Z. Cheng, Verifying event-b hybrid models using cyclone, in: International Conference on Rigorous State-Based Methods, Springer, 2023, pp. 179–184.

[16] L. De Moura, N. Bjørner, Z3: An efficient SMT solver, in: International conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.

[17] B. Bratton, The stack, The Log 35 (2015) 128–59.

[18] É. André, S. Liu, Y. Liu, C. Choppy, J. Sun, J. S. Dong, Formalizing UML state machines for automated verification–a survey, ACM Computing Surveys 55 (2023) 1–47.