VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Dinh Minh Hai

A SUPPORT TOOL TO SPECIFY AND VERIFY TEMPORAL PROPERTIES IN OCL

BACHELOR'S THESIS
Major: Computer Science

HA NOI – 2025

VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Dinh Minh Hai

A SUPPORT TOOL TO SPECIFY AND VERIFY TEMPORAL PROPERTIES IN OCL

BACHELOR'S THESIS

Major: Computer Science

Supervisor: Assoc. Prof. Dang Duc Hanh

HA NOI – 2025

ABSTRACT

Abstract: In Model-Driven Engineering (MDE), models serve as central artifacts for abstracting and designing software systems. Modern software systems often need to express and verify behaviors that involve temporal constraints and event-driven conditions. The Unified Modeling Language (UML) and the Object Constraint Language (OCL) are widely used in MDE to model systems and specify constraints. While OCL is effective for defining structural and simple behavioral properties, it lacks the ability to express temporal constraints and event-based behaviors. This limitation makes it challenging to specify and verify dynamic aspects of systems. This thesis proposes an extension of OCL with temporal and event-based constructs to enhance its ability to express and verify behavioral properties. We implement this extension as a plugin, called TemporalOCL, for the UML-based Specification Environment (USE) tool.

Keywords: *Model-Driven Engineering, Object Constraints Language, Temporal Properties, Model Checking*

DECLARATION

I hereby declare that I composed this thesis, ***"A Support Tool to Specify and Verify Temporal Properties in OCL"***, under the supervision of Assoc. Prof. Dang Duc Hanh. This work reflects my own effort and serious commitment to research. I have incorporated and adapted select open-source code and modeling resources to align with the research objectives, and all external materials used have been properly cited. I take full responsibility for the content and integrity of this thesis.

Ha Noi, 07th April 2025

Student

Dinh Minh Hai

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Assoc. Prof. Dang Duc Hanh, for his invaluable guidance and unwavering support throughout the research and writing of this thesis. His expertise and dedication have been instrumental in shaping this work.

I am also grateful to the alumni and current members of the research group for their insightful discussions and constructive feedback, which greatly enriched my research.

Furthermore, I extend my thanks to the faculty members of the University of Engineering and Technology for their passionate teaching and for equipping me with the essential knowledge and skills that form the foundation of this thesis.

Lastly, I offer my gratitude to my family for their constant care, support, and encouragement. Their belief in me provided the motivation and stability I needed to pursue and complete this thesis.

Although I have endeavored to conduct this research to the highest standard, I recognize that limitations in my knowledge and experience may have led to unintentional shortcomings. I sincerely welcome comments and suggestions from professors and peers to enhance this work further.

To all who have supported me on this journey, I am profoundly grateful.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATION AND TERMS

Abbreviation	**Full Form**
MDE	Model Driven Engineering
UML	Unified Modeling Language
OCL	Object Constraint Language
USE	UML-based Specification Environment
DEX	Decentralized Exchange
SPL	Solana Program Library
SDK	Software development kit
DOM	Document Object Model

INTRODUCTION

Modern software development faces significant challenges as systems grow increasingly complex. Traditional development approaches relying on manual coding often struggle to manage this complexity, leading to higher error rates and extended development cycles. These problems often come from the development process, not the system requirements. Model-Driven Engineering (MDE) helps solve this by shifting the focus to models instead of code. In MDE, developers use models to design systems, and tools can automatically generate code, documentation, and tests from them. The Unified Modeling Language (UML) and the Object Constraint Language (OCL) have become the *de facto* standards for model-driven approaches. UML provides a rich set of visual modeling concepts to represent the structural and behavioral aspects of a system, while OCL allows specifying constraints and structural properties of UML models. However, for complex systems, it is often necessary to specify and verify dynamic behaviors that involve temporal constraints and event-driven conditions. Unfortunately, OCL lacks the expressiveness to model these dynamic aspects, which limits its ability to specify and verify temporal properties and event-based behaviors.

This thesis aims to address this limitation by extending OCL with constructs for temporal properties and events, enhancing its expressiveness in modeling dynamic system aspects. We implement this extension as a plugin, called TemporalOCL, for the UML-based Specification Environment (USE), a tool that supports the specification and validation of software systems using UML and OCL. To enable not only specification but also verification of temporal properties, we employ a technique known as filmstripping, which transforms models with dynamic temporal constraints into structurally equivalent models that can be analyzed using existing verification tools. Our plugin automatically translates temporal OCL expressions into standard OCL constraints on a filmstrip model, allowing modelers to leverage the existing USE model validator for verification. This approach bridges the gap between expressing temporal requirements and verifying them, providing a complete solution that integrates seamlessly with the established USE environment and its validation capabilities.

The thesis is structured as follows:

- **Chapter 1**: This chapter lays the foundation for the background of this thesis. We explore theoretical concepts and tools that are used in this thesis.
- **Chapter 2**: This chapter presents our OCL extension to specify temporal properties and events.
- **Chapter 3**: This chapter describes the implementation and evaluation of the USE-TemporalOCL plugin.
- **Conclusion**: This chapter summarizes the contributions of this thesis and discusses future work.

Each chapter starts with an *Introduction* section, then ends with a *Summary* section.

Chapter 1

Backgrounds

1.1 Introduction

This chapter presents fundamentals about concepts and artifacts essential to this thesis. The modeling languages such as, Unified Modeling Language (UML), together with Object Constraint Language (OCL), are used to describe structural and behavioral aspects of systems and are briefly described in this chapter. A description of the modeling and specification tool called UML-based Specification Environment (USE) is presented, including its model validation capabilities that form the foundation for our verification approach. We explain the filmstrip model transformation process in detail, as it serves as the underlying mechanism for our temporal verification approach. Additionally, we introduce Temporal OCL (TOCL) as developed in prior research by [Author et al.]. Their approach extends OCL with temporal operators to express properties over time and transforms UML and OCL models into a Snapshot Transition Model (STM) to handle dynamic behaviors. In their work, TOCL expressions are translated into standard OCL constraints in the context of the STM. We review this foundational work as it forms the theoretical basis that our approach builds upon, though our implementation adapts these concepts to work with filmstrip models rather than STM. Each of these topics forms an essential building block for understanding our approach to specifying and verifying temporal properties in OCL, which will be presented in subsequent chapters.

1.2 Model-Driven Engineering

1.3 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting software-intensive systems. This language is maintained by the Object Management Group (OMG) [58].

UML is one of the most widely used modeling languages for describing real-world application domains. It works with various object and component methods to represent software systems. As software systems grow in size, complexity, and distribution, building and maintaining them becomes more challenging. UML helps reduce this complexity by providing a high level of abstraction that captures essential information needed for designing and developing software systems [50].

UML includes multiple diagram types, each focusing on different aspects of a design. These diagrams fall into two main categories: (1) structural diagrams that represent the static aspects of a system, and (2) behavioral diagrams that describe the dynamic aspects. The UML 2.5 specification includes fourteen different diagram types [71].

For this thesis, two structural diagrams are particularly relevant:

1.3.1 Class Diagram

Class diagrams are the most common diagram type in object-oriented modeling. They illustrate the static structure of a system by depicting classes, their attributes, operations, and the relationships between classes [64]. Classes represent sets of objects, where attributes describe the values these objects may contain, and operations specify the behaviors objects can perform.

Associations in class diagrams describe connections between different classes, with multiplicity indicators showing how many objects of one class can be linked to objects of another class. Class diagrams also include other relationship types: aggregation and composition (both representing whole-part relationships, with different levels of dependency), and generalization (inheritance relationship where specialized classes inherit properties from a general class).

[Example here]

These diagrams form the foundation of object-oriented modeling and are essential for understanding the system structure that our temporal and event-based extensions will work with. In this thesis, class diagrams provide the structural framework upon which temporal properties will be defined and verified.

1.3.2 Object Diagram

Object diagrams are structural diagrams that represent real-world entities or modeled system elements as concrete instances of classes. While class diagrams show abstract structures, object diagrams provide snapshots of a system at specific points in time, showing actual objects with specific attribute values and the links connecting them [77].

Objects in these diagrams are instances of classes defined in the class diagram, with concrete values assigned to their attributes. Links between objects are instances of the associations defined in the class diagram. This concrete representation makes object diagrams particularly valuable for verification purposes.

[Example here]

An important limitation of object diagrams is that they represent only a single state of the system. When the system state changes through operation calls, previous state information is lost. A single object diagram cannot represent the flow of information or system evolution over time. This limitation is particularly relevant to our work, as it highlights why standard UML/OCL approaches struggle with temporal specifications. In this thesis, object diagrams play a crucial role in our validation approach, where sequences of object diagrams (filmstrips) are used to represent and verify temporal properties.

1.4 Object Constraint Language (OCL)

1.4.1 Overview

As explained in the previous section, UML is a graphical language for visualizing system structure and behavior. However, visual modeling with UML alone is insufficient for developing accurate and consistent software models, as UML diagrams cannot express all necessary constraints. The Object Management Group (OMG) developed the Object Constraint Language (OCL) in 1997 to address this limitation. OCL is a formal assertion language with precise semantics that extends UML by allowing developers to specify constraints that cannot be expressed graphically.

For example, while a UML class diagram can show that a Bank has multiple Accounts, it cannot express that "an Account must maintain a minimum balance of $100" - this requires OCL. OCL provides two kinds of descriptions: expressions that evaluate to values, and constraints that must evaluate to true. OCL is type-rich, supporting basic types (Boolean, Real, Integer, String), collection types (Set, Bag, Sequence, OrderedSet), and special types (tuples, OclAny, OclType). The language includes navigation operators to traverse model relationships, comprehensive collection operations, and quantifiers for building logical statements. OCL constraints typically appear as class invariants (conditions that must always be true for all instances) and operation pre/postconditions (conditions that must be true before and after operation execution).

1.4.2 OCL Limitations

1.5 UML-based Specification Environment (USE)

1.5.1 Overview

1.5.2 USE Features

1.5.3 USE Model Validator

1.5.4 Filmstripping

1.5.4.1 Filmstrip Model Transformation

Chapter 2

Temporal and Event Constructs for OCL

2.1 Introduction

As established in Chapter 1, OCL provides robust support for specifying structural properties and simple behavioral constraints in UML models. However, OCL has significant limitations when applied to dynamic system behavior. It operates on single system states or individual transitions, making it unable to express properties that span across multiple states or respond to events within the system's execution. These limitations become particularly problematic for modern software systems, which frequently require complex temporal and reactive behaviors.

This chapter presents two main contributions to address these limitations:

First, we present TOCL+, a comprehensive extension of OCL that enhances its expressiveness for dynamic system aspects. TOCL+ combines temporal operators adapted from Temporal OCL research with novel event-based constructs. The temporal operators enable reasoning about system evolution over time with constructs like *always*, *sometime*, and *until*. Our event constructs address a critical gap by enabling the detection of specific occurrences during system execution, such as operation calls and state changes. Together, these extensions create a more powerful specification language capable of expressing complex dynamic requirements such as "when a login attempt fails three consecutive times, the account must be locked."

Second, we introduce a transformation approach that enables the verification of TOCL+ specifications using existing model checking tools. This approach transforms UML/OCL models into filmstrip models that expose state sequences, and translates TOCL+ specifications into standard OCL constraints that can be verified within the filmstrip context. This transformation bridges the gap between expressing temporal requirements and verifying them, providing a complete solution that integrates with established verification technologies.

The chapter is organized as follows:

- Section 2.2 presents the TOCL+ language extension, covering both temporal specification capabilities and our novel event-based constructs, as well as their integration.
- Section 2.3 details the transformation approach for verification, explaining how UML/OCL models are transformed to filmstrip models and how TOCL+ specifications are translated to standard OCL for verification.

By addressing both specification and verification aspects, this chapter provides a comprehensive solution to the challenge of expressing and verifying dynamic system properties within the MDE paradigm.

2.2 An Extended OCL for Temporal and Event Specifications

2.2.1 Temporal OCL (TOCL)

Temporal OCL enhances the Object Constraint Language (OCL) by enabling the specification of properties that must hold over time, across multiple states of a system. While standard OCL is limited to evaluating constraints within a single system state or across a single state transition (via pre- and postconditions), many system requirements involve dynamic behaviors that unfold over sequences of states. Examples include properties such as "eventually, the system will reach a stable state" or "once a condition is met, it must remain true thereafter." To address this, Temporal OCL (TOCL), as introduced by Ziemann and Gogolla [28], extends OCL with elements of linear temporal logic, allowing these temporal properties to be expressed directly within a familiar OCL-like syntax.

TOCL introduces a comprehensive set of temporal operators, divided into future and past categories, which are adopted in TOCL+ as the foundation for temporal reasoning. Below, we review these operators, their syntax, semantics, and provide illustrative examples.

2.2.1.1 Adopted TOCL Temporal Operators

The temporal operators in TOCL are categorized as follows:

Future Operators:

next e: True if the expression e holds in the next state. always e: True if e holds in the current state and all subsequent states. sometime e: True if e holds in the current state or at least one future state. always e1 until e2: True if e1 remains true until e2 becomes true, or if e1 remains true indefinitely if e2 never becomes true. sometime e1 before e2: True if e1 becomes true at some point before e2 does, or if e1 becomes true and e2 never does.

Past Operators:

previous e: True if e was true in the previous state (or if there is no previous state, i.e., at the initial state). alwaysPast e: True if e was true in all past states. sometimePast e: True if e was true in at least one past state. always e1 since e2: True if e1 has been true since the last time e2 was true. sometime e1 since e2: True if e1 has been true at some point since the last time e2 was true.

These operators enable precise specification of temporal relationships, making TOCL suitable for modeling and verifying dynamic system behaviors.

2.2.1.2 Syntax and Semantics

The syntax of TOCL integrates these temporal operators seamlessly into OCL expressions, allowing them to be used within invariants, preconditions, and postconditions. For example:

An invariant using always: context C inv: always (self.attribute > 0)

A condition using next: context C inv: (self.state = #active) implies next (self.state = #idle)

The semantics of these operators are defined over infinite sequences of system states, where each state represents a snapshot of the system at a given time. The evaluation of an expression depends on its position within this sequence:

next e is true if e holds at the state immediately following the current one. always e is true if e holds at the current state and all future states. sometime e is true if e holds at the current state or some future state. For past operators, the evaluation considers the sequence of states preceding the current state, with previous e being true if e held in the prior state, and so forth.

Formal definitions of the semantics are provided in [28], based on a state sequence ($\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots \rangle$), ensuring a rigorous foundation for TOCL. For a detailed formal treatment, readers are referred to the original paper.

2.2.1.3 Example Specifications

To demonstrate the practical application of these operators, we adapt examples from the steam boiler control specification problem [1], as presented in [28]:

Initialization Persistence:

context Program inv: self.mode = #initialization implies always self.mode

= #initialization until (PhysicalUnit.allInstances->forAll(pu | pu.ready) or self.wlmdFailure)

This invariant specifies that if the program is in initialization mode, it remains in that mode until all physical units are ready or a water level measurement failure occurs.

Eventual Water Level Drop: context SteamBoiler inv: self.valve = #open implies sometime self.wlmd.q <= n2

This constraint ensures that if the steam boiler's valve is open, the water level will eventually drop to or below the normal upper boundary n2.

Mode Transition: context Program inv: (self.mode = #initialization and self.wlmdFailure) implies next self.mode = #emergencystop

This specifies that if a failure is detected during initialization, the next state must transition to emergencystop.

These examples highlight how TOCL's temporal operators enable the specification of complex dynamic properties, forming a critical component of the TOCL+ language. In the subsequent subsections, we build upon this foundation by introducing event-based constructs and their integration with these temporal capabilities.

2.2.2 Event Constructs in OCL

Chapter 3

IMPLEMENTATION AND EXPERIMENTS

KẾT LUẬN

Phương hướng phát triển trong tương lai

REFERENCES

[1] Robert France and Bernhard Rumpe. “Model-driven Development of Complex Software: A Research Roadmap”. In: *Future of Software Engineering (FOSE ’07)*. 2007, pp. 37–54. DOI: `10.1109/FOSE.2007.14`.

[2] Martin Gogolla, Fabian Büttner, and Mark Richters. “USE: A UML-based specification environment for validating UML and OCL”. In: *Sci. Comput. Program.* 69.1–3 (Dec. 2007), pp. 27–34. ISSN: 0167-6423. DOI: `10.1016/j.scico.2007.01.013`. URL: `https://doi.org/10.1016/j.scico.2007.01.013`.