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 con-

straints 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.

Chapter 1

Backgrounds

1.1 Introduction

This chapter presents the fundamental concepts and tools that form the foundation of our approach to temporal specification and verification in model-driven engineering. We begin with an overview of Model-Driven Engineering (MDE), which provides the methodological framework for our research. Within this paradigm, models serve as primary artifacts throughout the software development lifecycle, enabling rigorous analysis and verification before implementation.

We then introduce the Unified Modeling Language (UML), the industry-standard visual modeling language for specifying software systems. For our work, we focus specifically on class diagrams, which define the abstract structure of a system, and object diagrams, which provide concrete instances of that structure. These structural diagrams establish the vocabulary and framework upon which our temporal extensions are built.

While UML provides powerful visual notation, it lacks formal mechanisms for expressing detailed constraints. We address this by examining the Object Constraint Language (OCL), which complements UML by enabling precise specification of constraints that cannot be expressed graphically. We review OCL's core concepts and syntax, with particular attention to its strengths and limitations regarding temporal properties.

Finally, we explore the UML-based Specification Environment (USE), the modeling and verification tool that implements our approach. USE pro-

vides the infrastructure for defining UML models with OCL constraints and validating them through automated analysis. We describe USE's model validation capabilities that form the technical foundation for our verification approach.

Throughout this chapter, we emphasize the context and limitations of standard modeling approaches regarding temporal specifications, setting the stage for our extensions and contributions in subsequent chapters. Each section provides essential background knowledge required to understand our approach to specifying and verifying temporal properties in object-oriented systems.

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) [2].

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.

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

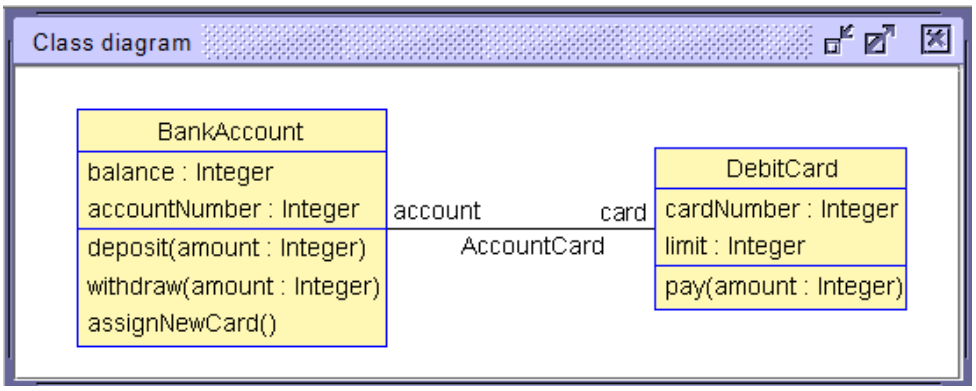

Figure 1.1: Class diagram of the Bank Account Model.

diagrams that describe the dynamic aspects. These structural and behavioral categories collectively contain fourteen different diagram types, as specified in the UML Reference Manual [1].

For this thesis, two related structural diagrams are particularly relevant and will be presented in the following subsections: class diagrams, which define the abstract structure of a system, and object diagrams, which provide concrete instances of that structure.

1.3.1 Class Diagram

Class diagrams are the foundation of structural modeling in UML and the most widely used diagram type in object-oriented systems. They illustrate the static structure of a system by depicting classes, their attributes, operations, and the relationships between classes. These concepts can be observed in Figure 1.1, which shows a class diagram of a simple bank account system.

In this diagram, we see two classes, `BankAccount` and `DebitCard`, which represent sets of objects that share common characteristics. Each class contains attributes that describe the data values their objects may contain. The `BankAccount` class has attributes such as:

- `accountNumber`: a unique identifier for the bank account

- `balance`: the current balance of the bank account

Similarly, the `DebitCard` class has attributes:

- `cardNumber`: a unique identifier for the debit card
- `limit`: the maximum amount that can be withdrawn using the debit card

Classes also include operations that specify the behaviors objects can perform. In our example, the `BankAccount` class defines three operations:

- `deposit(amount)`: adds the specified amount to the account balance
- `withdraw(amount)`: deducts the specified amount from the balance
- `assignNewCard()`: creates and assigns a new debit card to the bank account

These operations represent the functional capabilities of `BankAccount` objects, defining how they can interact with other objects and how their state can change over time. While attributes describe what an object knows, operations describe what an object can do.

Relationships between these classes are represented by the `AccountCard` association, which connects `BankAccount` and `DebitCard`. Multiplicity indicators on this association would show how many objects of one class can be linked to objects of another class. In addition to simple associations like this one, class diagrams can include more specialized relationship types: aggregation and composition (both representing whole-part relationships with different levels of dependency), and generalization (inheritance relationships where specialized classes inherit properties from a general class).

Class diagrams represent the static structure of a system at a particular point in time, providing the vocabulary and structural framework that other diagrams and behavioral specifications build upon.

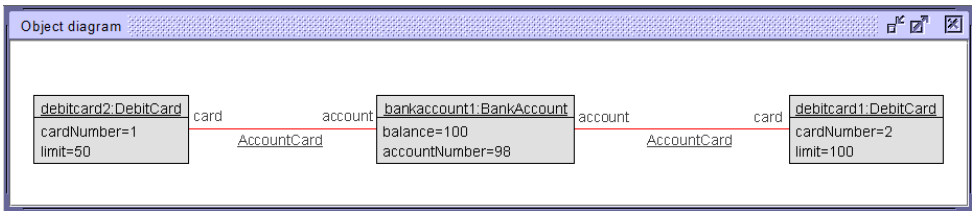

Figure 1.2: Object diagram of the Bank Account Model.

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.

Figure 1.2 shows an example object diagram for the banking system previously described in the class diagram (Figure 1.1). The links between objects in the diagram represent instances of the associations defined in the class diagram. Here, the `AccountCard` links connect the `bankaccount1` object to both debit card objects, showing that this particular bank account has two associated debit cards with different withdrawal limits.

Object diagrams provide concrete examples that help verify that a system model behaves as expected. They are valuable for validating class structures, illustrating complex relationships, and demonstrating specific scenarios during system design. While object diagrams excel at representing static information about system states, they do not capture the dynamic interactions that cause state changes. This characteristic defines both the strength and scope of object diagrams within UML modeling - they offer precise snapshots of system state at a particular moment in time, complementing the abstract structural representations provided by class diagrams.

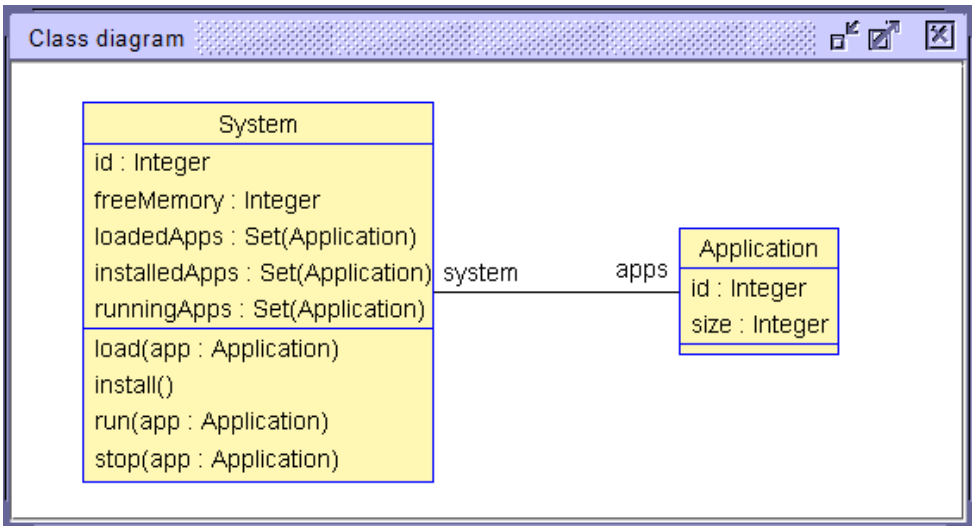

Figure 1.3: Class diagram of the Software System.

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) 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.

To demonstrate OCL's capabilities, we'll use a simple software system model shown in Figure 1.3. This model contains two classes: `System` and `Application`. Each class has an `id` attribute for unique identification. The `System` class has a `freeMemory` attribute representing available memory, while each `Application` has a `size` attribute indicating its memory requirements. The `System` class maintains three collections: `loadedApps`, `installedApps`, and `runningApps`, which track applications in different states

throughout their lifecycle.

The `System` class defines the following operations:

- `load(app : Application)`: downloads the application *app* given as parameter and adds it to the `loadedApps` collection.
- `install()`: installs all the loaded applications in the `loadedApps` collection and moves them to the `installedApps` collection.
- `run(app : Application)`: executes the application *app* given as a parameter that should be installed, adding it to the `runningApps` collection.
- `stop(app : Application)`: stops the application *app* given as a parameter that should be running, removing it from the `runningApps` collection.

1.4.2 OCL Constraints

Listing 1.1 demonstrates three typical aspects of OCL constraints. First, the `memoryConstraint` ensures system integrity by verifying that the system's free memory is non-negative, preventing memory overallocation. Second, the `notLoadedAndInstalled` constraint demonstrates OCL's ability to work with collections, ensuring that the sets of loaded and installed applications don't overlap - an application cannot be simultaneously in both states. This constraint uses the `intersection` and `isEmpty` operations to verify this condition. Third, the `sizeConstraint` demonstrates how OCL can define simple rules that apply to all instances of a class, in this case ensuring all applications have a positive size.

```
1 context System
2 inv memoryConstraint: self.freeMemory >= 0
3 inv notLoadedAndInstalled: self.loadedApps->intersection(
     self.installedApps)->isEmpty()
4
5 context Application
```

```
inv sizeConstraint: self.size > 0
```

Listing 1.1: OCL constraints.

OCL constraints typically appear in three forms:

- **Invariants:** Conditions that must always be true for all instances of a class throughout their lifetime, as shown in our examples above.
- **Preconditions:** Conditions that must be true before an operation executes. For instance, we could specify that an application must not be in any collection before the `load` operation can be performed.
- **Postconditions:** Conditions that must be true after an operation completes. For example, after executing the `load` operation, the application must be added to the `loadedApps` collection.

Listing 1.2 demonstrates pre- and postconditions for the `load` operation. The preconditions verify that (1) the application is not already in any of the three collections (`loadedApps`, `installedApps`, or `runningApps`) and (2) there is enough memory available for the application. The postconditions ensure that (1) the application is added to the `loadedApps` collection and (2) the available memory is reduced by the application's size.

```
pre notLoaded: not self.loadedApps->includes(app) and
               not self.installedApps->includes(app) and
               not self.runningApps->includes(app)
pre enoughMemory: self.freeMemory >= app.size
post loaded: self.loadedApps = self.loadedApps@pre->
   including(app)
post freeMemory: self.freeMemory = self.freeMemory@pre -
   app.size
```

Listing 1.2: OCL rules.

In the postcondition `freeMemory`, note the use of the `@pre` operator, which references the value of an attribute before the operation execution.

This allows OCL to express constraints that relate the state before and after an operation. In this case, it ensures that the system's free memory after loading is reduced by exactly the size of the loaded application.

These examples represent just a small subset of OCL's expressive capabilities. 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 provides powerful navigation capabilities for traversing relationships in the model, comprehensive collection operations for manipulating groups of objects, and quantifiers (forAll, exists) for building complex logical statements.

1.4.3 OCL Limitations

1.4.3.1 Temporal Dimension

To illustrate the temporal limits of OCL, let us consider the following temporal properties of our software system:

- **Safety 1:** An application loading must precede its run (i.e., an application can only be run if the `load` operation has previously been called on it)
- **Safety 2:** There must be an install operation between an application's loading and its running (i.e., an application must have the `install` operation called on it after loading and before running)
- **Liveness:** Every application in the `runningApps` collection will eventually be removed from it (i.e., the `stop` operation will eventually be called for every running application)

Such temporal properties are impossible to specify in OCL without at least enriching the model structure with state variables. In temporal logics,

we formally distinguish safety properties (which prevent bad events/states) from liveness properties (which ensure good events/states eventually happen). Safety properties consider finite behaviors and can sometimes be handled by modifying the model to save the system history, but this approach quickly becomes cumbersome and error-prone.

The fundamental limitation is that OCL expressions can only describe a single system state or a one-step transition from a previous state to a new state upon operation call. Therefore, there is no direct way to express OCL constraints involving different states of the model at arbitrary points in time—OCL has a very limited temporal dimension.

1.4.3.2 Events

OCL also has significant limitations in handling events. An event is a predicate that holds at different instants of time. Mathematically, it can be represented as a function $P : \text{Time} \to \text{true}, \text{false}$ which indicates, at each instant, whether the event is triggered. The subset $t \in \text{Time} \mid P(t) \subseteq \text{Time}$ represents all time instants at which the event P occurs.

In the object-oriented paradigm, we commonly distinguish five kinds of events:

- **Operation call events:** Instants when a sender calls an operation of a receiver object
- **Operation start events:** Instants when a receiver object starts executing an operation
- **Operation end events:** Instants when the execution of an operation is finished
- **Time-triggered events:** Events that occur when a specified instant is reached

- **State change events:** Events that occur each time the system state changes (e.g., when the value of an attribute changes)

OCL only provides implicit support for events through its pre- and post-conditions. Preconditions offer an implicit universal quantification over operation call events, while postconditions provide an implicit universal quantification over operation end events. For example, a precondition on the `load` operation implicitly quantifies over all instances when this operation is called.

However, OCL lacks explicit constructs for the finest type of events which is state change events. These events, which occur when attribute values or object relationships change, are particularly important for reactive systems that must respond to changes in their environment. This limitation, combined with OCL's restricted temporal expressiveness, makes it difficult to specify many realistic system requirements that involve reactions to events occurring over time.

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

OCL provides strong support for structural properties in UML models but falls short when specifying dynamic system behavior. Operating only on single states or individual transitions, OCL cannot express properties spanning multiple states or responding to system events. This limitation is significant for modern systems requiring temporal and reactive behaviors.

Temporal logics like LTL and CTL offer formal frameworks for temporal properties but require specialized knowledge unfamiliar to most UML designers. This creates a practical barrier for practitioners comfortable with UML/OCL but not with formal temporal notations.

This chapter presents two main contributions:

First, TOCL+ extends OCL with temporal and event capabilities. It adds temporal operators like *always*, *sometime*, and *until* for reasoning about system evolution over time, and introduces event constructs for detecting specific system occurrences such as operation calls and state changes. TOCL+ maintains OCL's familiar syntax while enabling complex dynamic specifications.

Second, we introduce a transformation approach that enables verification of TOCL+ specifications using existing tools. This approach transforms UML/OCL models into filmstrip models representing state sequences, and translates TOCL+ specifications into standard OCL constraints verifiable within these models.

The chapter is organized as follows:

- Section 2.2 presents the TOCL+ language extension, covering temporal operators, event constructs, and their integration.
- Section 2.3 details the transformation approach, explaining the model transformation and specification translation processes.

Together, these contributions provide a complete solution for both specifying and verifying temporal properties within the model-driven engineering paradigm.

2.2 An Extended OCL for Temporal and Event Specifications

2.2.1 TOCL

In this thesis, we leverage TOCL, as introduced by Ziemann and Gogolla [3], as the temporal foundation for specifying properties that must hold over time across multiple states of a system. Standard Object Constraint Language (OCL) is limited to evaluating constraints within a single system state or across a single state transition (via pre- and postconditions), which is insufficient for capturing the dynamic behaviors inherent in many system requirements. For instance, properties such as "eventually, the system will reach a stable state" or "once a condition is met, it must remain true thereafter" require reasoning over sequences of states. TOCL addresses this limitation by extending OCL with elements of linear temporal logic, enabling the expression of such temporal properties within a familiar OCL-like syntax.

TOCL's comprehensive set of temporal operators, categorized into future and past operators, provides the essential temporal reasoning capabilities for our work. In this thesis, we adopt these operators unchanged as the basis for modeling and verifying dynamic system behaviors over time. However,

to address systems that exhibit reactive behaviors driven by specific events, we extend TOCL into TOCL+ by integrating novel event-based constructs. This extension, detailed in the next section, complements TOCL's temporal framework, enabling a more holistic specification of both state-based temporal properties and event-driven dynamics. Below, we review the adopted TOCL temporal operators, their syntax, and semantics, which serve as the cornerstone of TOCL+.

2.2.1.1 Adopted TOCL Temporal Operators

TOCL defines two categories of temporal operators that we adopt in our work:

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** e_1 **until** e_2**:** True if e_1 remains true until e_2 becomes true, or if e_1 remains true indefinitely if e_2 never becomes true.
- **sometime** e_1 **before** e_2**:** True if e_1 becomes true at some point before e_2 does, or if e_1 becomes true and e_2 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** e_1 **since** e_2**:** True if e_1 has been true since the last time e_2 was true.
- **sometime** e_1 **since** e_2**:** True if e_1 has been true at some point since the last time e_2 was true.

For the formal semantics of these operators, we refer to the original work by Ziemann and Gogolla [3], which defines them over sequences of system states.

2.2.1.2 Example Specifications

To demonstrate the practical application of these operators, we apply them to the software system case study introduced in Chapter 1:

2.2.2 Event Constructs in OCL

To capture these concepts, TOCL+ introduces two primary event constructs:

- **isCalled**: A generic event construct that unifies operation events. It detects when an operation is invoked on an object, representing the atomic transition from a pre-state to a post-state.
- **becomesTrue**: A state change event that is parameterized by an OCL boolean expression P. It designates a step in which P becomes true (i.e., P was evaluated to false in the previous state and is true in the current state). In the object-oriented paradigm, a state change is necessarily a consequence of some operation call, therefore becomesTrue acts as syn-

tactic sugar for any operation call that causes P to switch from false to true.

2.2.2.1 Formal Definition

Formally, we define events in terms of operations and state transitions. Let O be the set of all operations and E be the set of all OCL boolean expressions in a model. An event is either:

- isCalled(op) - representing a call to operation op, optionally with precondition pre and postcondition post
- becomesTrue(P) - representing any operation call that transitions the system from a state where $\neg$P holds to a state where P holds

This formal definition enables precise reasoning about when events occur during system execution and forms the foundation for our verification approach.

2.2.2.2 Examples

To illustrate these event constructs, consider a banking system with an Account class:

- isCalled(account.withdraw(100)) - Detects when the withdraw operation is called with parameter 100 on an account object
- becomesTrue(account.balance $<$ 0) - Detects when an account's balance transitions from non-negative to negative

These examples demonstrate how event constructs enable the specification of critical moments in a system's execution, providing the foundation for more complex temporal properties. By combining these event constructs

with the temporal operators from TOCL, we create a powerful specification language capable of expressing reactive behaviors and complex temporal patterns.

Chapter 3

IMPLEMENTATION AND EXPERIMENTS

KẾT LUẬN

Phương hướng phát triển trong tương lai

REFERENCES

[1] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language reference manual*. GBR: Addison-Wesley Longman Ltd., 1998. ISBN: 020130998X. DOI: 10.5555/294049. URL: https://dl.acm.org/doi/book/10.5555/294049.

[2] *Unified modeling language specification version 2.5.1*. Object Management Group, 2017. URL: https://www.omg.org/spec/UML/2.5.1.

[3] Mustafa Al Lail et al. "Transformation of TOCL temporal properties into OCL". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 899–907. ISBN: 9781450394673. DOI: 10.1145/3550356.3563132. URL: https://doi.org/10.1145/3550356.3563132.