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.

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 provides the infrastructure for defining UML models with OCL constraints and validating them. We introduce two key plugins that extend USE's capabilities: the Filmstrip Plugin, which implements the filmstripping method by transforming dynamic model checking into static verification through sequences of snapshots connected by operation calls; and the Model Validator Plugin, which enables automated analysis of models against their constraints through systematic state space exploration. Together, these tools form the technical foundation for our verification approach, enabling both the representation of temporal properties and their efficient verification.

Throughout this chapter, we emphasize the context and limitations of standard modeling approaches regarding temporal specifications and verifications, 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

Modeling in software engineering involves creating abstract representations of software systems. In traditional software development approaches, models often serve merely as documentation or architectural blueprints for the system being developed. Model-Driven Engineering (MDE), by contrast, elevates models to first-class artifacts in the development process, using them as primary means to address the growing complexity of large software systems.

During the software design phase, developers create models using specialized modeling languages that facilitate the specification of system components. These models capture the system at an abstract level, including only information relevant to the design task at hand. This abstraction reduces complexity, accelerates the design process, and maintains independence from specific programming languages and implementation platforms. Furthermore, MDE technologies enable validation and verification of these models early in the development lifecycle, potentially improving the quality of the final system. Through this emphasis on modeling, MDE promises significant improvements in both productivity and quality in software development.

The models in MDE are typically constructed using standardized modeling languages such as the Unified Modeling Language (UML), often complemented by formal specification languages like the Object Constraint Language (OCL) to describe both structural and behavioral aspects of the system. A typical UML/OCL application model consists of a class diagram containing classes, attributes, associations, and operations. The structural integrity of the model is maintained through OCL invariants, while behavioral properties are specified using operation pre- and postconditions. Invariants constrain the possible system states to ensure valid object configurations, while pre- and postconditions govern valid system dynamics by defining permissible state transitions through operation calls. In this approach, the complete application is described as a cohesive model, with each transition initiated by a specific operation call.

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) [6].

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 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

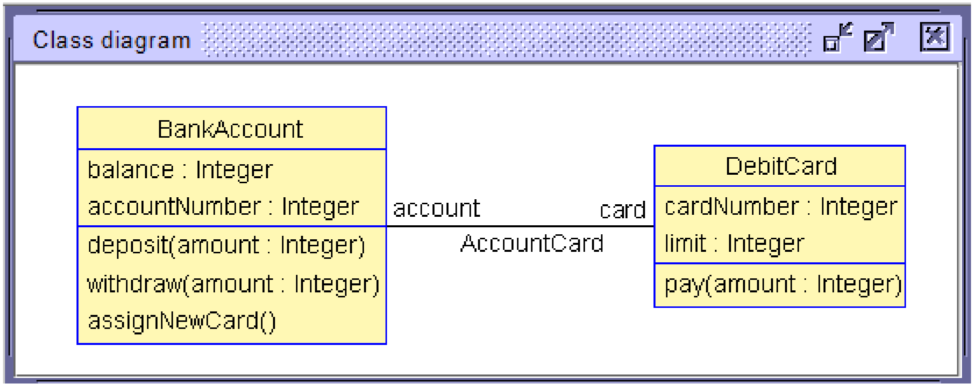

Figure 1.1: Class diagram of the Bank Account Model.

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

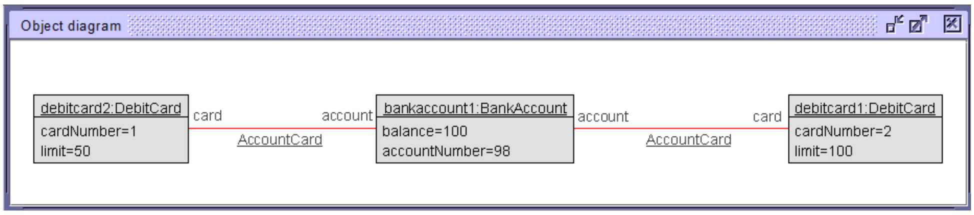

Figure 1.2: Object diagram of the Bank Account Model.

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.

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.

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:

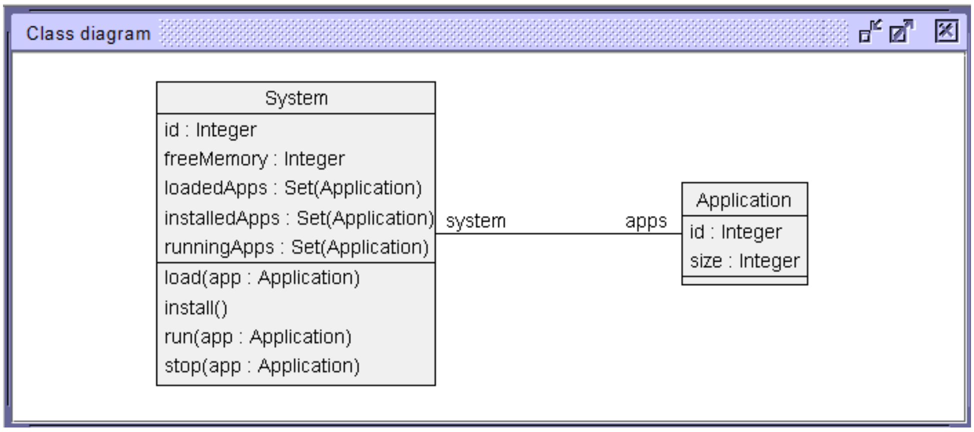

Figure 1.3: Class diagram of the Software System.

- `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.

```
context System
inv memoryConstraint: self.freeMemory >= 0
inv notLoadedAndInstalled: self.loadedApps->intersection(self.installedApps)->
    isEmpty()

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.

Safety 2: There must be an install operation between an application's loading and its running.

Safety 3: Each application can be loaded at most one time.

Liveness: Every loaded application will eventually be installed.

Figure 1.4: Temporal properties of the software system.

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 [4].

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 dynamic systems that must detect and respond to changes in their operating 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

The UML-based Specification Environment (USE) is a system for the specification and validation of information systems based on a subset of UML and OCL [2]. Models in USE are specified in textual form (as .use files) containing classes with their attributes and operations, associations, and OCL constraints. These constraints include class invariants and operation pre/-postconditions, all defined using OCL expressions. USE supports model animation to validate specifications against non-formal requirements, allowing developers to create and manipulate system states (snapshots) during animation. For each snapshot, USE automatically checks OCL constraints and highlights violations. The tool provides comprehensive graphical visualization

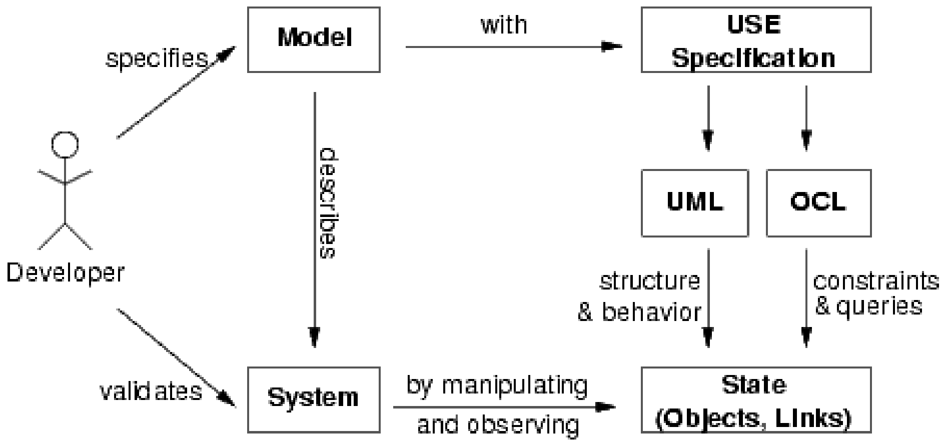

Figure 1.5: USE Overview.

of model elements through various diagram types, including class diagrams, object diagrams, and sequence diagrams. Additionally, USE allows users to enter and evaluate OCL expressions interactively to query detailed information about the current system state. This combination of precise specification with dynamic validation makes USE particularly valuable for detecting inconsistencies and design flaws early in the development process. Figure 1.5 gives a general view of the USE approach.

1.5.2 USE Model Validator

The Model Validator extends USE's capabilities through a specialized plugin that automates the generation of object diagrams from class diagrams within a configurable search space [2]. This plugin bridges the gap between manual model animation and systematic verification by employing a transformation-based approach. The validator converts UML/OCL models into relational logic using Kodkod, which is subsequently transformed into a boolean satisfiability (SAT) problem for efficient analysis. When a solution is found, it is immediately displayed as an object diagram in the USE interface, with the option to explore alternative valid states. Developers control the validation process through configuration files (.properties) that define search parameters, including upper and lower bounds for classes, attributes, and associations. These configurations can be supplemented with additional OCL invariants to target specific scenarios. When executed via the validate command, the Model Validator systematically searches for system states that satisfy all constraints, reporting either SATISFIABLE (with a corresponding object diagram) when a valid configuration exists, or UNSATISFIABLE when the constraints cannot be collectively satisfied. This automated approach significantly enhances USE's ability to detect inconsistencies and validate model properties that would be difficult to verify through manual testing alone.

1.5.3 Filmstripping

1.5.3.1 Overview

Filmstripping is a model transformation technique developed to extend USE's verification capabilities from static structure to dynamic behavior [3]. While standard OCL validation tools (including USE's Model Validator) primarily focus on structural aspects like invariants, the filmstrip approach enables verification of behavioral properties by transforming dynamic specifications into static ones. The method works by converting a UML/OCL model containing both invariants and operation pre/postconditions into an equivalent model containing only invariants. This transformed "filmstrip model" consists of the original application model augmented with specialized structures that capture system state progression. The key insight is the introduction of explicit `Snapshot` classes that represent individual system states, with `OperationCall` classes that connect consecutive snapshots. Through this transformation, temporal sequences of operations and object states are flattened into a single, verifiable object diagram. Pre and postconditions from the original model are systematically converted into invariants that constrain relationships between snapshots, effectively embedding behavioral specifica-

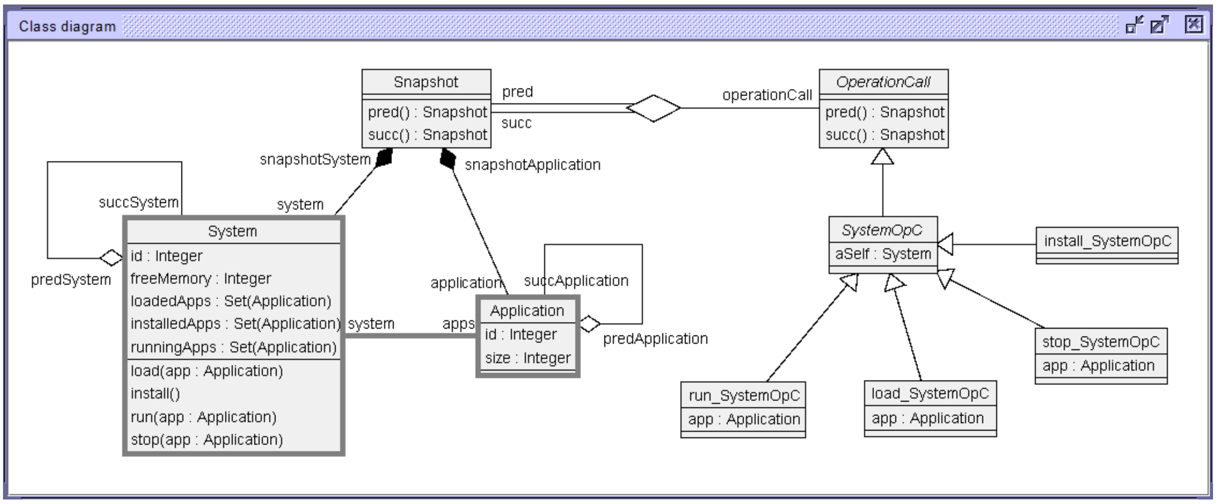

Figure 1.6: Filmstrip Model Transformation.

tions within the static structure. This approach enables the Model Validator to verify complex behavioral properties—including operation sequencing, state transitions, and temporal constraints—using the same validation mechanisms originally designed for structural verification. By bridging the gap between static and dynamic validation, filmstripping provides a comprehensive framework for verifying both aspects of a model within a single technical infrastructure. In the following subsection, we detail the specific transformation process that converts standard UML/OCL models into filmstrip models, explaining how operation contracts are transformed into invariants and how system state progression is represented.

1.5.3.2 Filmstrip Model Transformation

The filmstrip transformation process is best illustrated through an example. Figure 1.6 shows the transformation of our Software System model from Figure 1.3 into its filmstrip equivalent. The original application model—classes `System` and `Application` with their `SystemApplication` association—remains intact within the filmstrip model, visually distinguished by gray borders.

The transformation is performed automatically by the Filmstrip Plugin for USE, which augments the original model with additional elements (shown without gray borders). These elements include `Snapshot` objects that capture individual system states and `OperationCall` classes (with suffix `OpC`) that represent the operations from the application model. Each operation is converted into a corresponding `OperationCall` class containing attributes for the context object (self) and operation parameters.

The complete transformation process involves the following steps [3]:

Transformation of classes: All classes and attributes from the application model are preserved in the filmstrip model. Two essential classes are added: `Snapshot`, which associates objects with specific system states, and `OperationCall`, which represents state transitions. Operation parameters become attributes in their respective operation call classes, and all operation call classes inherit from the base `OperationCall` class through generalization.

Transformation of associations: All original associations are maintained in the filmstrip model. A crucial ternary association is added to link pre-snapshots to post-snapshots through operation calls, representing the system's state evolution. Additional associations connect application objects to their respective snapshots, ensuring that each object exists in exactly one snapshot state, while aggregation links represent object persistence across snapshots.

Transformation of operation definitions and invariants: Operation definitions and invariants from the application model are incorporated without modification.

Transformation of pre- and postconditions: Operation contracts (pre- and postconditions) are converted into invariants in the filmstrip model, associated with their respective operation call classes. These invariants are evaluated once for each operation call instance, preserving the semantic equivalence between the original contracts and their filmstrip representations.

1.6 Summary

This chapter has established the conceptual and technical foundations necessary for understanding our approach to temporal specification and verification. We explored the Unified Modeling Language (UML) as the standard visual notation for modeling object-oriented systems, and examined the Object Constraint Language (OCL) and its role in expressing precise constraints, highlighting its inherent limitations regarding temporal properties—particularly its inability to express constraints across multiple system states and event occurrences. We then presented the UML-based Specification Environment (USE) and its critical extensions: the Model Validator Plugin for automated constraint checking and the Filmstrip Plugin for transforming dynamic properties into statically verifiable constraints. This background reveals a significant gap in current modeling practices: while OCL effectively expresses structural constraints, it lacks constructs for temporal specifications essential for reactive and event-driven systems. The filmstripping approach provides a promising verification foundation, but requires corresponding specification mechanisms to fully address temporal properties. In the next chapter, we introduce TOCL+ as our solution to this specification challenge, extending OCL with temporal and event constructs while leveraging the filmstrip verification framework.

Chapter 2

Specification and Verification of Temporal Properties in 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 Specification of temporal properties

2.2.1 Temporal Operators in TOCL

TOCL (Temporal OCL), introduced by Ziemann and Gogolla [7], extends OCL with temporal operators for specifying properties across multiple system states. It incorporates linear temporal logic elements while preserving OCL's familiar syntax and type system. TOCL defines its semantics over an infinite sequence of states $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots \rangle$, where each operator is evaluated in an environment $\tau = (\hat{\sigma}, i, \beta)$ with i representing the current state index and β a variable assignment. TOCL organizes its operators into two categories: future operators and past operators.

In our work, we adopt the following temporal operators:

Future Operators:

- **next** e**:** True if e holds in the next state (state $i+1$).
- **always** e**:** True if e holds in the current state and all subsequent states (all states $j \geq i$).

- **sometime** e**:** True if e holds in the current state or at least one future state (some state $j \geq i$).
- **always** e_1 **until** e_2**:** True if e_1 remains true until e_2 becomes true, or indefinitely if e_2 never occurs.
- **sometime** e_1 **before** e_2**:** True if e_1 becomes true at some point before e_2 does.

Past Operators:

- **previous** e**:** True if e was true in the previous state or if at the initial state ($i = 0$).
- **alwaysPast** e**:** True if e was true in all past states (all states $0 \leq j < i$).
- **sometimePast** e**:** True if e was true in at least one past state (some state $0 \leq j < i$).
- **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 details regarding TOCL's formal semantics, grammar specification, and OCL integration, Ziemann and Gogolla [7] provide a comprehensive formalization that defines these operators over sequences of system states. Their work establishes how temporal expressions maintain OCL's type system while extending its scope to reason about multiple states. This formalization provides the theoretical foundation for our event extensions in the following section.

To demonstrate TOCL's capabilities, we apply it to the first two temporal properties from our Software System example 1.4:

```
context System
/*
An application loading must precede its run.
*/
inv safety1:
    self.runningApps->notEmpty() implies
    self.runningApps->forAll(app |
        sometimePast self.loadedApps->includes(app)
    )
/*
There must be an install operation between loading and running.
*/
inv safety2:
    self.loadedApps->notEmpty() implies
    self.loadedApps->forAll(app |
        sometime self.installedApps->includes(app)
        before self.runningApps->includes(app)
    )
```

Listing 2.1: TOCL Specification for Safety Properties.

TOCL can express properties spanning multiple states through state-based workarounds, as shown in Listing 2.1. For Safety 1, rather than directly detecting the `load` operation call, TOCL uses `sometimePast` with state predicates to infer that loading occurred before running based on collection membership. Similarly, for Safety 2, TOCL uses the `before` operator with state predicates to infer the sequencing of operations through their effects on system state. While indirect, these specifications work because they only need to track the order of state changes, not count specific events.

However, TOCL fundamentally cannot specify Safety 3: *"Each application can be loaded at most one time"*. This property requires counting operation call occurrences, which cannot be inferred from state changes alone. Since TOCL lacks constructs to identify when operations are called or to count events, it cannot express constraints that limit how many times an operation occurs. This limitation becomes a critical barrier when specifying common safety properties that restrict operation frequencies.

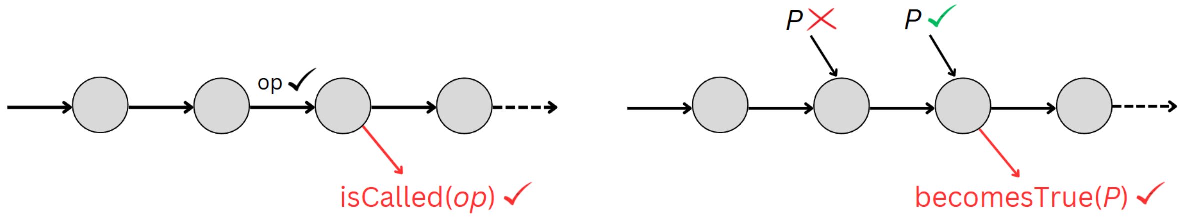

Figure 2.1: Events.

2.2.2 Event Constructs in OCL

To address TOCL's limitations in expressing event-based properties, we propose TOCL+, which extends TOCL with explicit event specification capabilities. Following the synchronous paradigm, TOCL+ represents operation calls as atomic transitions from pre-states to post-states without intermediate states. This approach simplifies verification while preserving essential system behaviors.

TOCL+ introduces two primary event constructs:

1. **isCalled**: Detects when an operation is invoked on an object. It takes the operation call with its parameters as an argument and represents the atomic transition from pre-state to post-state.

2. **becomesTrue**: Represents a state change event parameterized by an OCL boolean expression P. It identifies transitions where P changes from false to true between consecutive states.

We adopt the concept of events from [4], which defines events as predicates identifying specific instants in time. As discussed in Section 1.4, object-oriented systems typically recognize operation events, time-triggered events, and state change events. TOCL+ focuses on operation and state change events as they capture the fundamental interactions in object-oriented systems.

In addition, TOCL+ supports bounded existence properties with constructs like:

- `at most k times` - limiting event occurrences to no more than k
- `exactly k times` - requiring precisely k occurrences of an event
- `at least k times` - requiring event occurrences to be k or more

Applying TOCL+ to our Software System example from Chapter 1, we can express all four temporal properties from Figure 1.4:

```
context System
/*
An application loading must precede its run.
*/
inv safety1:
    self.runningApps->notEmpty() implies
    self.runningApps->forAll(app |
        isCalled(run(app : Application)) implies
        sometimePast isCalled(load(app : Application))
    )

/*
There must be an install operation between an application's loading and its
    running.
*/
inv safety2:
    self.runningApps->notEmpty() implies
    self.runningApps->forAll(app |
        isCalled(run(app : Application)) implies (
            sometime isCalled(install())
            since isCalled(load(app : Application))
        )
    )

/*
Each application can be loaded at most one time.
*/
inv safety3:
    self.installedApps->notEmpty() implies
    self.installedApps->forAll(app |
        sometimePast isCalled(load(app : Application))
        at most 1 times
    )

/*
Every loaded application will eventually be installed.
*/
```

```
inv liveness:
    self.loadedApps->notEmpty() implies
    self.loadedApps->forAll(app |
        sometime isCalled(install())
    )
```

Listing 2.2: TOCL+ Specifications.

These examples show TOCL+'s expressive power. With the `isCalled` construct, safety properties 1 and 2 directly reference operation calls rather than inferring them from state changes. Safety property 3 uses bounded existence ("at most 1 times") to limit operation occurrences - something impossible in both standard OCL and TOCL. The liveness property also benefits from direct operation call detection, making the requirement clearer.

2.2.3 Formal Definition for Event constructs

Formally, we define TOCL+ event constructs in terms of state transitions within the semantic framework established by TOCL. Let $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots \rangle$ be an infinite sequence of states, and $\tau = (\hat{\sigma}, i, \beta)$ be an evaluation environment where i represents the current state index and β is a variable assignment.

Unlike the original TOCL approach with process types, we directly associate operation calls with state transitions. We assume each transition from σ_{i-1} to σ_i is caused by exactly one atomic operation execution, with no intermediate states.

Our event constructs are formally defined as follows:

isCalled(op(a$_1$, ..., a$_N$)) This construct detects when an operation *op* is invoked on an object with specific parameters. It evaluates to true at state σ_i if the transition from σ_{i-1} to σ_i was caused by the operation *op* being called on the context object with the specified parameters.

For an operation *op* defined in class C with parameters $param_1 : type_1$,

$\ldots, param_N : type_N$, and context object $self$ of type C, the semantics at state σ_i in environment $\tau = (\hat{\sigma}, i, \beta)$ is:

$$\begin{aligned}
&I[\text{isCalled}(op(a_1, \ldots, a_N))](\tau) = \text{true} \iff \\
&\bullet\ i > 0 \text{ and} \\
&\bullet\ \text{The transition from } \sigma_{i-1} \text{ to } \sigma_i \text{ is labeled with a call } \text{call}_i = (\omega, o, \text{args}) \\
&\quad \text{where:} \\
&\qquad \circ\ \omega = \text{op and} \\
&\qquad \circ\ o = I[\text{self}](\tau) \text{ and} \\
&\qquad \circ\ \text{args} = (I[a_1](\tau), \ldots, I[a_N](\tau))
\end{aligned} \tag{2.1}$$

becomesTrue(P) This construct identifies transitions where a boolean expression P changes from false to true between consecutive states.

For a boolean OCL expression P, the semantics at state σ_i in environment $\tau = (\hat{\sigma}, i, \beta)$ is:

$$\begin{aligned}
&I[\text{becomesTrue}(P)](\tau) = \text{true} \iff \\
&\bullet\ i > 0 \text{ and} \\
&\bullet\ I[P](\hat{\sigma}, i-1, \beta) = \text{false and} \\
&\bullet\ I[P](\hat{\sigma}, i, \beta) = \text{true}
\end{aligned} \tag{2.2}$$

This definition is equivalent to:

$$I[\text{becomesTrue}(P)](\tau) = I[P \text{ and not previous } P](\tau) \tag{2.3}$$

Bounded Existence Constructs For the bounded existence constructs, we extend the semantics to count event occurrences within a temporal scope.

For an event e and temporal scope S (e.g., all past states for `sometimePast`):

$$\mathrm{count}(e, S) = |\{j \in S \mid I[e](\hat{\sigma}, j, \beta) = \mathrm{true}\}| \tag{2.4}$$

Then:

$$\begin{aligned} I[e \text{ at most } k \text{ times}](\tau) = \mathrm{true} &\iff \mathrm{count}(e, S) \leq k \\ I[e\ k \text{ times}](\tau) = \mathrm{true} &\iff \mathrm{count}(e, S) = k \\ I[e \text{ at least } k \text{ times}](\tau) = \mathrm{true} &\iff \mathrm{count}(e, S) \geq k \end{aligned} \tag{2.5}$$

These formal definitions provide a clean, process-type-free semantics for TOCL+'s event constructs and bounded existence operators. By directly associating events with state transitions and state changes, we establish a foundation for the verification approach described in the next section.

2.2.4 TOCL+ Grammar

To enable automated verification of TOCL+ specifications, we provide a formal grammar that precisely defines the language syntax. The original TOCL by Ziemann and Gogolla used mathematical notation to describe its syntax. Later, Lail et al. [8] defined a formal EBNF grammar for TOCL using ANTLR4, creating a parser-friendly representation of the language. Our work builds directly upon this EBNF foundation, extending it with productions for our new event constructs and bounded existence operators.

We leverage the ANTLR4 grammar developed by Lail et al. and augment it with additional productions to support TOCL+ features. This approach allows us to maintain compatibility with their TOCL parser while adding our event-based constructs. Listing 2.3 shows the key grammar productions for our event extensions.

Listing 2.3: EBNF Grammar for Event constructs.

```
events[Environment env] returns [ASTEvent ast]
        : isCalledEvent[$env] { $ast = $isCalledEvent.ast; }
```

```
        | becomesTrueEvent[$env] { $ast = $becomesTrueEvent.ast; }
        ;

isCalledEvent[Environment env] returns [ASTEvent ast]
        : 'isCalled' LPAREN eventOp[$env] RPAREN bounds[$env]?
        { $ast = new ASTEvent(); }
        ;

bounds[Environment env]
        : quantif=('at least' | 'at most')? n=NATURAL_N 'times'
        ;

eventOp[Environment env]
        : simpleName LPAREN parameters[$env]? RPAREN
        ;

becomesTrueEvent[Environment env] returns [ASTEvent ast]
        : 'becomesTrue' LPAREN e=binaryOperationExp[$env] RPAREN
        { $ast = new ASTEvent(); }
        ;
```

In this grammar, the `events` production serves as the entry point for event expressions, handling both `isCalled` and `becomesTrue` constructs. Each production builds an abstract syntax tree (AST) node that represents the event for further processing. The `isCalledEvent` production recognizes operation call events with optional bounded existence constraints defined by the `bounds` production. These bounds can be specified as "at least" or "at most" followed by an integer and "times", directly mapping to the formal semantics defined earlier. The `eventOp` production handles the operation name and parameters, while `becomesTrueEvent` captures state change events parameterized by boolean expressions.

Our grammar integrates these event constructs with the full range of TOCL temporal operators by defining events as a subtype of primary expressions (`primaryExp`). This design allows event expressions to be used in any context where OCL expressions are expected, including as arguments to temporal operators. For example, the grammar enables expressions like `sometimePast isCalled(op()) at most 3 times`, which combines a temporal operator, an event construct, and a bounded existence constraint.

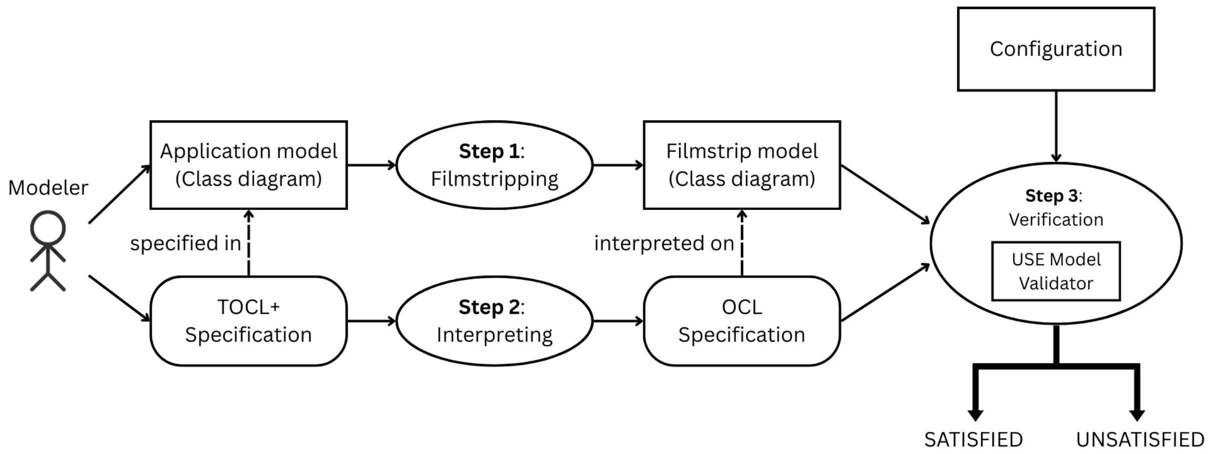

Figure 2.2: Verification approach.

This formal grammar specification serves two critical purposes. First, it provides a precise definition of the TOCL+ language syntax, complementing the semantic foundation established in the previous section. Second, it enables automated transformation from TOCL+ specifications to standard OCL constraints in our verification framework, which will be described in the next section.

2.3 Verification of TOCL+ Properties

Figure 2.2 presents a comprehensive overview of our verification approach for TOCL+ properties. The process begins when a UML modeler creates an Application Model and specifies temporal properties using TOCL+. Our approach consists of three sequential transformation and verification steps:

First, we transform the Application Model into a Filmstrip Model. This crucial step converts dynamic specifications into static ones by representing the system's behavior through a sequence of snapshots and operation calls. The Filmstrip Model effectively flattens temporal behavior into a structural representation that can be analyzed using static verification techniques.

Second, we translate the TOCL+ properties into equivalent OCL expressions interpreted over the Filmstrip Model. These OCL expressions navigate through snapshots and operation calls, ensuring that the temporal constraints are properly enforced across the system's execution. The translation process systematically converts temporal operators and event constructs into path expressions over the filmstrip structure.

Third, we verify the resulting OCL expressions using the USE Model Validator [5]. This static analysis tool examines the Filmstrip Model against the translated constraints within a configurable search space defined in a properties file. The validator explores possible system states, reporting either SATISFIABLE (when a valid system state exists that satisfies all constraints) or UNSATISFIABLE (when no valid state can satisfy all constraints simultaneously).

In the following subsections, we explain each step in detail using our Software System example from Figure 1.3.

2.3.1 Step 1: Transforming the Application Model into a Filmstrip Model

The first step in our verification approach applies the filmstripping technique (described in Section 1.5.3) to transform our Software System model into a static representation. Rather than repeating the general transformation process, we focus here on the specific application to our example and how it enables subsequent verification steps.

For our Software System model from Figure 1.3, the filmstrip transformation preserves the original classes (`System` and `Application`) and their associations while introducing the filmstrip infrastructure. The four operations in our model—`load(app: Application)`, `install()`, `run(app: Application)`, and `stop(app: Application)`—are each transformed into concrete `OperationCall` subclasses that capture their specific parameters and context.

The critical aspect of this transformation is how it represents temporal behavior through structural relationships. Each snapshot in the resulting filmstrip model corresponds to a distinct point in time, with operation calls connecting these snapshots to form execution sequences. This structural representation allows us to track how object states change over time, which is essential for verifying temporal properties.

The transformation of operation contracts is particularly important for our verification approach. Consider the pre- and postconditions for the `load` operation shown in Listing 2.4. These conditions are transformed into invariants in the filmstrip model, as shown in Listing 2.5. For example, the postcondition `loaded` becomes an invariant that navigates between pre- and post-operation states using `succSystem` and `predSystem` associations.

The resulting filmstrip model was previously illustrated in Figure 1.6 in Chapter 1, showing how the original application model is extended with filmstrip-specific elements. This structural representation of system behavior forms the foundation for the next steps in our verification approach, allowing us to express and check temporal properties as static constraints over the filmstrip model.

```
context System::load(app: Application)
    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 reduceMemory:
        self.freeMemory = self.freeMemory@pre - app.size
```

Listing 2.4: Pre and post conditions for load operation.

```
context load_SystemOpC
inv pre_notLoaded:
    not aSelf.loadedApps->includes(app) and
```

```
    not aSelf.installedApps->includes(app) and
    not aSelf.runningApps->includes(app)

context load_SystemOpC
inv pre_enoughMemory:
    aSelf.freeMemory >= app.size

context load_SystemOpC
inv post_loaded:
    aSelf.succSystem.loadedApps =
    aSelf.succSystem.predSystem.loadedApps->collectNested( a1:Application |
        a1.succApplication
    )->asSet()->including(app.succApplication)

context load_SystemOpC
inv post_reduceMemory:
    aSelf.succSystem.freeMemory =
    aSelf.succSystem.predSystem.freeMemory - app.succApplication.size
```

Listing 2.5: Invariants for load_SystemOpC

2.3.2 Step 2: Translating TOCL+ Properties into OCL Expressions

After transforming the application model into a filmstrip model, the second step of our verification approach involves translating TOCL+ temporal properties into equivalent OCL constraints. These constraints must be expressed in terms of the filmstrip model structure created in Step 1, allowing them to be verified using standard OCL tools.

The translation process begins with temporal properties specified in TOCL+ for the original Software System model. Each TOCL+ property is systematically transformed into an OCL constraint that navigates through the filmstrip structure of snapshots and operation calls. This transformation preserves the semantic meaning of the original temporal specifications while expressing them through the structural elements of the filmstrip model.

Listing 2.2 shows the original TOCL+ properties specified for our Software System model, including safety and liveness properties. As an example of the translation process, Listing 2.6 presents the OCL translation for the

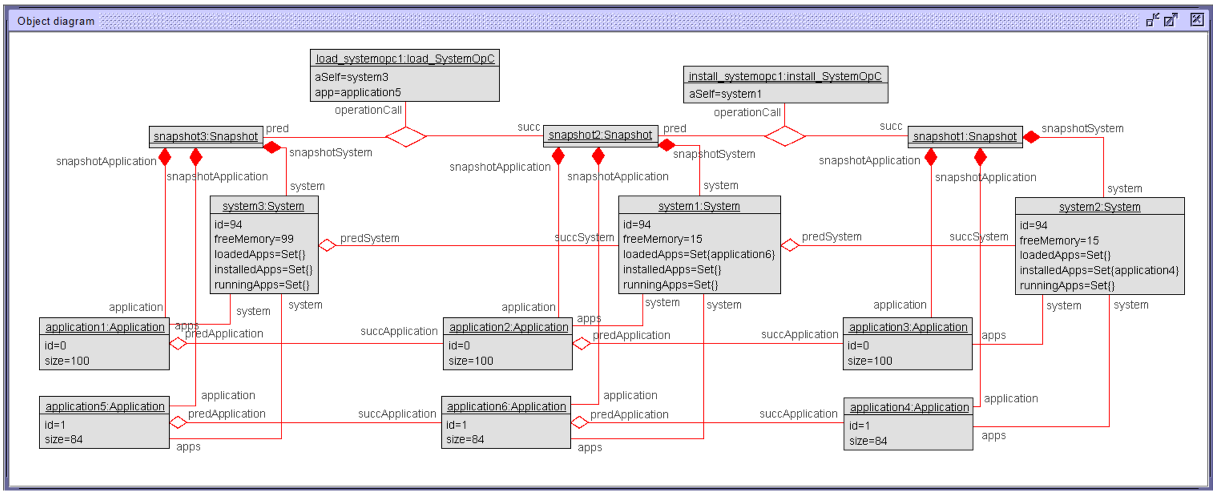

Figure 2.3: Object Diagram returned by the Model Validator.

liveness property, demonstrating how temporal requirements are mapped to structural constraints within the filmstrip framework. The translation uses snapshot navigation and operation call existence checks to capture the temporal semantics of the original property.

The resulting OCL constraints serve as the input for the verification step that follows, where they will be evaluated against potential system states to determine if the model satisfies the specified temporal properties. The detailed implementation of the interpretation process will be presented in Chapter 3.

```
context System
inv liveness:
    self.loadedApps->notEmpty() implies
    self.loadedApps->forAll(app |
        (let CS:Snapshot = self.snapshotSystem in
        Set{CS}->closure(s | s.succ())->excluding(null)->exists(s |
            install_SystemOpC.allInstances()->exists(op | op.succ() = s)
        ))
    )
```

Listing 2.6: OCL translation of liveness property.

2.3.3 Step 3: Verifying the translated OCL expressions

The final step in our verification approach employs the USE Model Validator [5] to verify the OCL constraints generated in Step 2 against our filmstrip model. This verification process determines whether the original TOCL+ properties are satisfied by the application model. The Model Validator systematically explores possible system states using a boolean satisfiability (SAT) solver, searching for valid instances of the filmstrip model that satisfy all constraints or demonstrating that no such instance exists.

Before verification, we configure the search space through a properties file (.properties) that establishes bounds for model elements including types, classes, and associations. These bounds define the scope within which the Model Validator explores possible system configurations. Listing 2.7 shows one specific configuration used in our Software System example.

When executed with these parameters, the Model Validator attempts to find a valid model instance that satisfies all constraints—including both the filmstrip model invariants and our translated temporal properties. If such an instance exists, the validator produces an object diagram as evidence; otherwise, it reports that the properties cannot be satisfied within the given search bounds.

Figure 2.3 displays an object diagram returned by the Model Validator checking against our liveness property. This diagram illustrates a scenario where the system first loads and then installs an application—a sequence that satisfies our liveness requirement. The diagram shows three snapshots representing the system's state at different points in time, connected by two operation calls: `load` followed by `install`.

You may notice the custom `id` attribute present in Figure 1.3 and Figure 2.3. We manually added this attribute to help identify corresponding objects across different snapshots. For example, the scenario shows three Application objects with an identical `id` value of 0, indicating they represent the same logical entity at different points in time. This attribute is not part of the standard filmstrip model, and we will explain the implementation details and necessary constraints for maintaining object identity in Chapter 3. This validation confirms that our Software System model can satisfy the temporal behavior specified by the liveness property.

Listing 2.7: Configuration file used for Fig. 2.3.

```
Integer_min = 0
Integer_max = 100
String_max = 10
Real_min = -2.0
Real_max = 2.0
Real_step = 0.5

### Classes
# Snapshot
Snapshot_min = 3
Snapshot_max = 3
# Filmstrip
Filmstrip_min = 2
Filmstrip_max = 2
# System
System_min = 3
System_max = 3
# Application
Application_min = 6
Application_max = 6

### Operation Classes
# load_SystemOpC
load_SystemOpC_min = 1
load_SystemOpC_max = 1
# install_SystemOpC
install_SystemOpC_min = 1
install_SystemOpC_max = 1
# run_SystemOpC
run_SystemOpC_min = 0
run_SystemOpC_max = 0
# stop_SystemOpC
stop_SystemOpC_min = 0
stop_SystemOpC_max = 0

### Associations
# SnapshotSystem
SnapshotSystem_min = 3
SnapshotSystem_max = 3
```

```
# SnapshotApplication
SnapshotApplication_min = 6
SnapshotApplication_max = 6
# PredSuccSystem
PredSuccSystem_min = 2
PredSuccSystem_max = 2
# PredSuccApplication
PredSuccApplication_min = 4
PredSuccApplication_max = 4
# SystemApplication
SystemApplication_min = 6
SystemApplication_max = 6

### Additional configurations
aggregationcyclefreeness = on
forbiddensharing = on
```

2.4 Summary

This chapter presented TOCL+, our extension to OCL for specifying and verifying temporal properties in UML models. We began by examining the limitations of standard OCL and existing temporal extensions, particularly the inability to express event-based and bounded existence properties. To address these limitations, we introduced TOCL+, which extends TOCL with two key event constructs: `isCalled` for detecting operation invocations and `becomesTrue` for capturing state changes. We complemented these with bounded existence operators that enable specifying constraints on event frequencies. We provided formal semantics for these constructs, defining them precisely in terms of state transitions, and developed a complete EBNF grammar to support automated processing. For verification, we presented a three-step approach: first transforming the application model into a filmstrip model, then translating TOCL+ specifications into standard OCL constraints over this model, and finally using the USE Model Validator to check these constraints. We demonstrated the effectiveness of our approach on a Software System example, showing how TOCL+ can express and verify complex temporal properties including safety, liveness, and fairness constraints that were previously inexpressible in OCL. This combination of enhanced expressiveness with automated verification provides modelers with powerful new capabilities for ensuring the correctness of dynamic behavior in their UML models.

Chapter 3

Implementation and Experiment

3.1 Introduction

In the previous chapter, we introduced TOCL+, an extension of OCL with temporal operators and event constructs, and presented a theoretical framework for specifying and verifying temporal properties in UML models. While the formal semantics and verification approach provide a solid foundation, they require practical tool support to be effectively applied. This chapter bridges the gap between theory and practice by presenting our implementation of a TOCL+ plugin for the USE tool and demonstrating its effectiveness through a detailed case study.

The primary objective of this chapter is twofold. First, we describe the implementation of our TOCL+ plugin for the USE tool, which enables modelers to specify temporal properties using our extended language and automatically verify them using the filmstrip approach. Second, we demonstrate the practical application of TOCL+ through a detailed case study of the Software System model introduced in Chapter 2, showcasing how various types of temporal properties can be effectively specified and verified.

In the following sections, we present the architecture and implementation of our plugin, focusing on how it integrates with the USE tool environment and implements the transformation process described in Chapter 2. We then explore the Software System case study, illustrating how our approach handles the specification and verification of different types of temporal properties in practice.

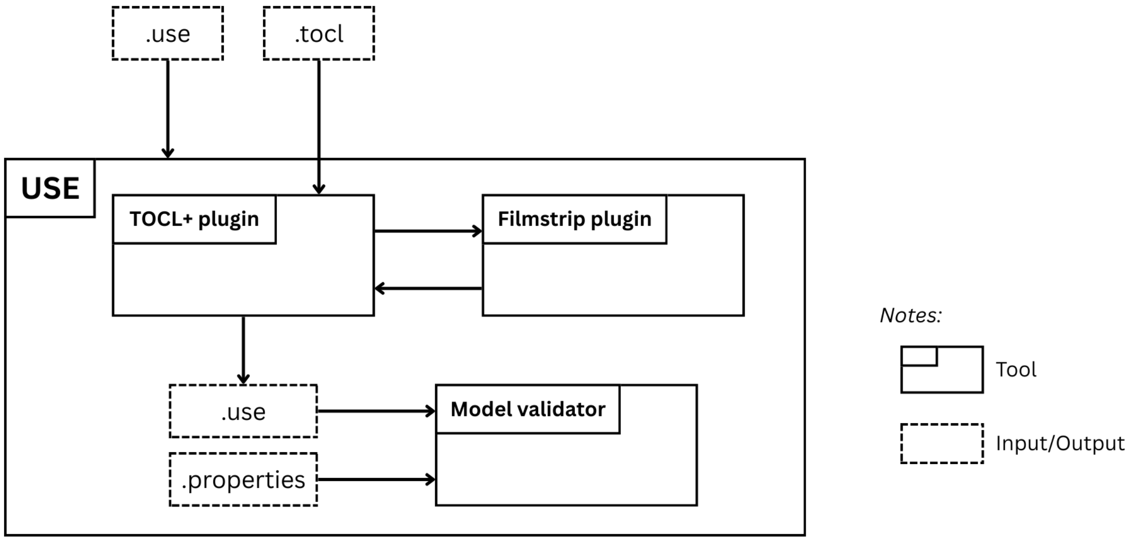

Figure 3.1: Architecture and Workflow of the approach.

3.2 TOCL+ Plugin Implementation

3.2.1 Plugin Architecture and Workflow

Figure 3.1 illustrates the architecture and workflow of our TOCL+ plugin. The plugin integrates with the USE tool environment while maintaining a clear separation between modeling, specification, and verification concerns. The workflow consists of five distinct steps, beginning with preparation and ending with verification. First, the user prepares two input files: a standard `.use` file containing the UML/OCL application model and a `.tocl` file containing TOCL+ property specifications. Second, the user loads the application model into USE to make it available for transformation. Third, the user activates our plugin through the USE interface, selecting both a destination path for the output model and the `.tocl` file containing the temporal properties to verify.

Internally, the plugin then executes a two-phase transformation process. In the first phase, it invokes the Filmstrip plugin to transform the application model into a filmstrip model following the rules described in Section 1.5.3. In the second phase, it processes the TOCL+ expressions using our ANTLR4-generated parser and listener components, which implement the transformation rules for converting temporal specifications into equivalent OCL constraints (detailed in the next subsection). These generated constraints are added to the list of invariants in the output model file alongside the filmstrip model elements.

To complete the verification process, the user loads this output model back into USE together with a configuration file that establishes search bounds, and then employs the Model Validator to analyze the constraints. The validator systematically explores the search space, determining whether the temporal properties are satisfied and providing a model instance as evidence when applicable. This architecture shields users from the complexities of the underlying transformation mechanisms while providing a streamlined workflow from specification to verification.

3.2.2 Implementation of TOCL+ to OCL Transformation

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

Table 3.1: Translation of TOCL+ operators to OCL.

<table>
<tr><th>No.</th><th>TOCL+</th><th>OCL Translation</th></tr>
<tr><td>1</td><td>next P</td><td>let nextSnapshot:Snapshot = self.snapshot.succ() in [nextSnapshot |= P]</td></tr>
<tr><td>2</td><td>always P</td><td>let CS:Snapshot = self.snapshot in Set{CS}->closure(s | s.succ())->forAll(s | [s |= P])</td></tr>
<tr><td>3</td><td>always P until Q</td><td>let CS:Snapshot = self.snapshot in let FS:Set(Snapshot) = Set{CS.succ()}->closure(s | s.succ()) in let AllFSQ:Set(Snapshot) = FS->select(s | [s |= Q]) in let FSQ:Snapshot = AllFSQ->any(s | Set{s}->closure(s | s.succ())->includesAll(AllFSQ)) in let afterQ:Set(Snapshot) = Set{FSQ}->closure(s | s.succ()) in let FSP:Set(Snapshot) = FS->select(s | [s |= P]) in if FSQ.isDefined() then (if (FSP->size() > 0) then (FS-afterQ = FSP-afterQ) else false endif) else (FS = FSP) endif</td></tr>
<tr><td>4</td><td>always P since Q</td><td>let CS:Snapshot = self.snapshot in let PS:Set(Snapshot) = Set{CS.pred()}->closure(s | s.pred()) in let AllPSQ:Set(Snapshot) = PS->select(s | [s |= Q]) in let PSQ:Snapshot = AllPSQ->any(s | Set{s}->closure(s | s.pred())->includesAll(AllPSQ)) in let beforeQ:Set(Snapshot) = Set{PSQ}->closure(s | s.pred()) in let PSP:Set(Snapshot) = PS->including(CS)->select(s | [s |= P]) in if PSQ.isDefined() then (if (PSP->size() > 0) then (PS->including(CS)-beforeQ = PSP-beforeQ) else false endif) else (PSP = PS->including(CS)) endif</td></tr>
<tr><td>5</td><td>sometime P</td><td>let CS:Snapshot = self.snapshot in Set{CS}->closure(s | s.succ())->exists(s | [s |= P])</td></tr>
<tr><td>6</td><td>sometime P before Q</td><td>let FS:Set(Snapshot) = Set{self.snapshot}->closure(s | s.succ()) in let PreS:Set(Snapshot) = Set{self.snapshot.pred()}->closure(s | s.pred()) in let AllFSQ:Set(Snapshot) = FS->select(s | [s |= Q]) in let FSQ:Snapshot = AllFSQ->any(s | Set{s}->closure(s | s.succ())->includesAll(AllFSQ)) in let FSP:Set(Snapshot) = FS->select(s | [s |= P]) in if FSQ.isDefined() then (if (FSP->size() > 0) then ((Set{FSQ.pred()}->closure(s | s.pred())-PreS)->exists(s_1 | FSP->includes(s_1))) else false endif) else false endif</td></tr>
<tr><td>7</td><td>sometime P since Q</td><td>let CS:Snapshot = self.snapshot in let PS:Set(Snapshot) = Set{CS.pred()}->closure(s | s.pred()) in let AllPSQ:Set(Snapshot) = PS->select(s | [s |= Q]) in let PSQ:Snapshot = AllPSQ->any(s | Set{s}->closure(s | s.pred())->includesAll(AllPSQ)) in let PSP:Set(Snapshot) = PS->select(s | [s |= P]) in if PSQ.isDefined() then (SetPSQ->closure(s | s.succ())->excluding(null)->intersection(PS)->exists(s | PSP->includes(s))) else false endif</td></tr>
<tr><td>8</td><td>previous P</td><td>let previousSnapshot:Snapshot = self.snapshot.pred() in [previousSnapshot |= P]</td></tr>
<tr><td>9</td><td>sometimePast P</td><td>let CS:Snapshot = self.snapshot in Set{CS.pred()}->closure(s | s.pred())->exists(s | [s |= P])</td></tr>
<tr><td>10</td><td>alwaysPast P</td><td>let CS:Snapshot = self.snapshot in Set{CS.pred()}->closure(s | s.pred())->forAll(s | [s |= P])</td></tr>
</table>

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] 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.

[3] Frank Hilken, Lars Hamann, and Martin Gogolla. "Transformation of UML and OCL Models into Filmstrip Models". In: *Theory and Practice of Model Transformations*. Ed. by Davide Di Ruscio and Dániel Varró. Cham: Springer International Publishing, 2014, pp. 170–185. ISBN: 978-3-319-08789-4.

[4] Bilal Kanso and Safouan Taha. "Specification of temporal properties with OCL". In: *Science of Computer Programming* 96 (2014). Selected Papers from the Fifth International Conference on Software Language Engineering (SLE 2012), pp. 527–551. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2014.02.029. URL: https://www.sciencedirect.com/science/article/pii/S0167642314000963.

[5] Nils Przigoda et al. "Integrating an SMT-Based ModelFinder into USE.". In: *MoDeVVa@ MoDELS*. Citeseer. 2016, pp. 40–45. DOI: 10.1109/MoDeVVa.2016.10.

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

[7] 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.

[8] 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.