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 primary artifacts for abstracting and designing software systems. The Unified Modeling Language (UML) and the Object Constraint Language (OCL) are widely used for specifying structural and behavioral properties of these models. However, OCL has fundamental limitations in expressing properties that span multiple system states or involve event occurrences, making it challenging to specify and verify dynamic aspects of systems. This thesis addresses these limitations by introducing TOCL+ (Temporal OCL+), an extension of OCL with temporal operators and event constructs. TOCL+ enables developers to specify complex temporal properties such as safety and liveness constraints using familiar OCL syntax, while incorporating temporal logic concepts. We present a systematic transformation approach that converts TOCL+ expressions into standard OCL constraints interpreted over filmstrip models—representations of system execution paths as sequences of snapshots. This transformation enables verification of temporal properties using existing OCL tools without requiring specialized temporal logic model checkers. We implement this approach as a plugin for the UML-based Specification Environment (USE) tool and demonstrate its effectiveness through a software system case study, showing how our approach bridges the gap between intuitive temporal specification and practical model verification.

Keywords: *Model-Driven Engineering, Object Constraint Language, Temporal Properties, Event Specification, Model Verification, Filmstrip Models, OCL Extensions, USE Tool, Formal Verification*

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 May 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
TOCL	Temporal Object Constraint Language
TOCL+	Temporal Object Constraint Language Plus

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) [3] addresses these challenges by shifting the focus from code to models. In MDE, developers use models to design systems, and tools can automatically generate code, documentation, and tests from those models. The Unified Modeling Language (UML) [17] and the Object Constraint Language (OCL) [16] 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, thereby enhancing its expressiveness in modeling dynamic system aspects. We implement this extension, called TOCL+, as a plugin for the UML-based Specification Environment (USE) [4], a tool that supports the specification and validation of software systems using UML and OCL. To enable both specification and verification of temporal properties, we employ a technique known as filmstripping [9], 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 filmstrip models, allowing modelers to leverage the existing USE model validator [15] 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: Background** lays the foundation for understanding the theoretical concepts and tools used in this thesis.
- **Chapter 2: Specification and Verification of Temporal Properties in OCL** presents our approach to extending OCL for specifying and verifying temporal properties.
- **Chapter 3: Implementation and Experiment** describes the implementation of our approach and evaluates it through a case study.
- **Conclusion** summarizes the contributions of this thesis and discusses future work.

Chapter 1

Background

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 (MDE). We begin with an overview of Model-Driven Engineering, 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.

This section introduces 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 a system's abstract structure, and Object Diagrams, which provide concrete instances of that structure. These diagrams establish the formal foundation on which our temporal extensions are built.

Although UML offers rich visual notation, it lacks a formal way to express detailed constraints. The Object Constraint Language (OCL) fills this gap by allowing precise, text-based specifications that UML diagrams cannot capture 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) [17].

UML serves as a standardized modeling language for visualizing and documenting software-intensive systems [17]. It incorporates various object-oriented and component-based methods to represent software architectures and designs. Software development projects face inherent complexity challenges when systems increase in size, functionality, and distribution across networks. UML addresses these challenges by offering structured graphical representations that abstract complex systems into comprehensible models, capturing essential structural and behavioral elements required for effective design and implementation processes. The notation system provides developers with a common vocabulary for communicating system architecture and behavior among stakeholders [19].

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 [19].

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 adapted from [21], which shows a class diagram of a simple bank account system.

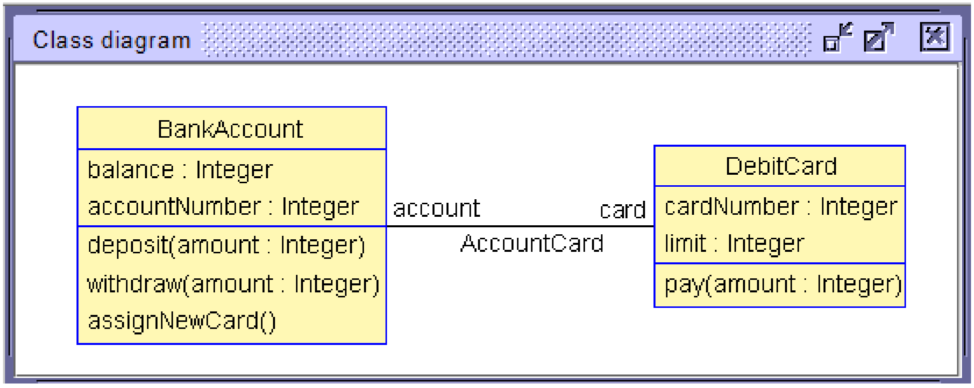


Figure 1.1: Class diagram of the Bank Account Model adapted from [21].

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.

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.

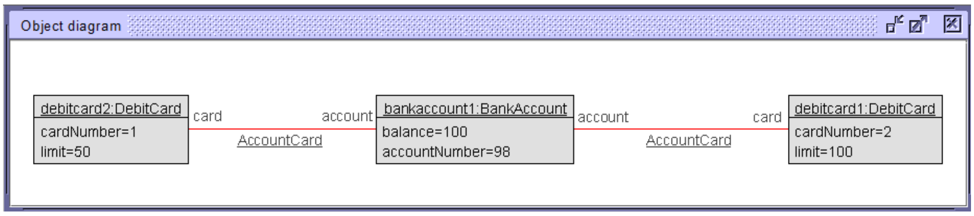


Figure 1.2: Object diagram of the Bank Account Model.

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 [16]. OCL is a formal assertion language with precise semantics that extends UML by allowing developers to specify constraints that cannot be expressed graphically.

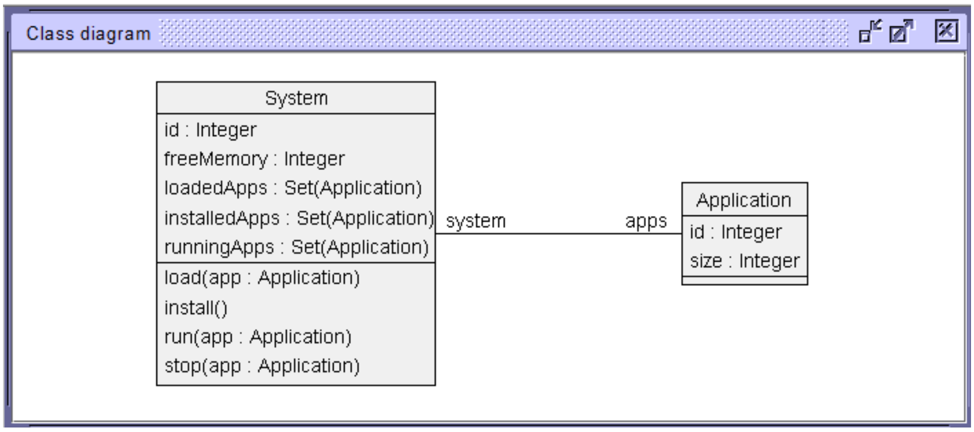


Figure 1.3: Class diagram of the Software System.

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.

Listing 1.1: OCL constraints

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

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

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.

Listing 1.2: OCL rules

```
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
```

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 [10].

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 postconditions. 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 [4]. 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 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 [4]. 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 [5], which is subsequently transformed into a boolean satisfiability (SAT) problem for efficient analysis. When a so-

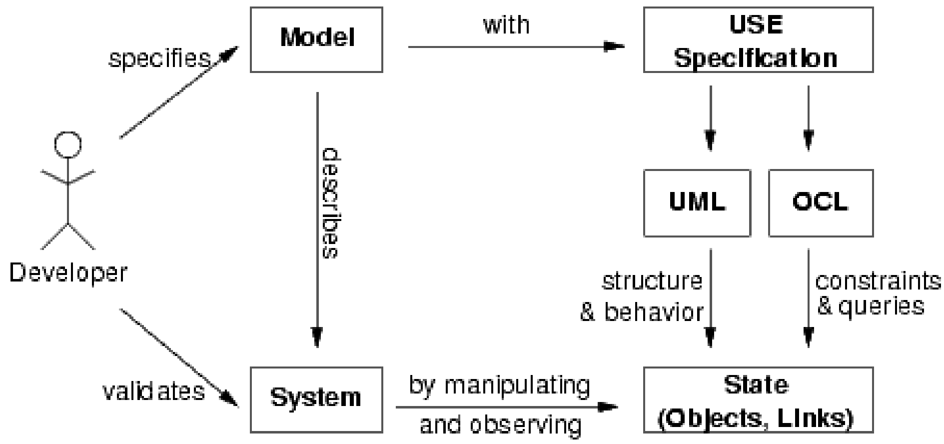


Figure 1.5: USE Overview.

lution is found, it is immediately displayed as an object diagram in the USE interface, with the option to explore alternative valid states [7]. 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 [13]. 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 [8] [9]. 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 specifications 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` associ-

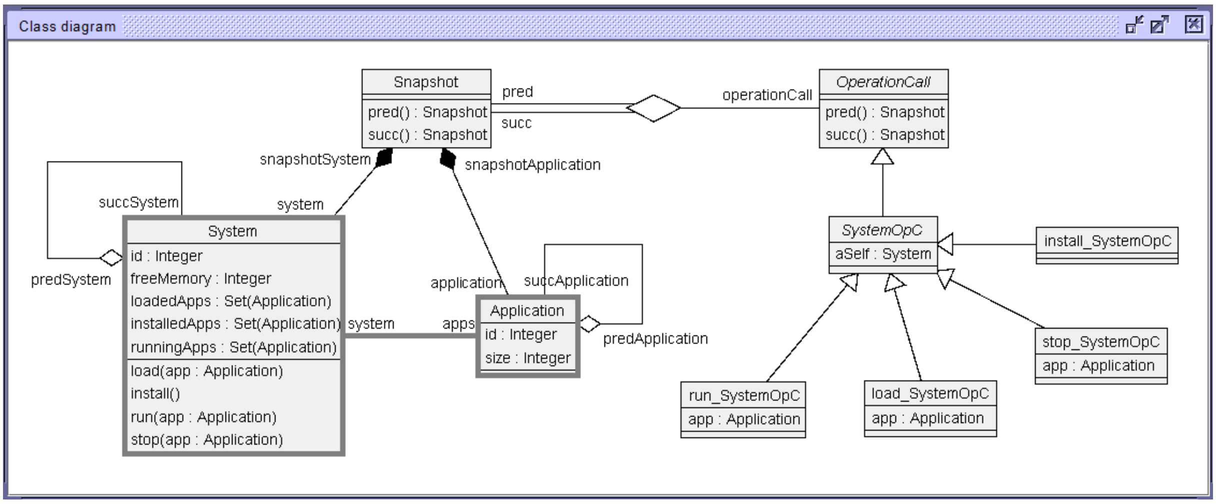


Figure 1.6: Filmstrip Model Transformation.

ation—remains intact within the filmstrip model, visually distinguished by gray borders.

The transformation is performed automatically by the Filmstrip Plugin for USE [9], 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 [9]:

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 [1] 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 first provides an overview of Temporal OCL (TOCL) [2], a foundation we build upon. We then present our two main contributions:

First, we introduce TOCL+, our extension to TOCL that addresses its limitations in expressing event-based properties. TOCL+ enhances TOCL with event constructs for detecting specific system occurrences such as operation calls and state changes, along with support for bounded existence properties (e.g., "an operation is called exactly n times" or "at most n times"). TOCL+ maintains OCL's familiar syntax while significantly expanding its capabilities for specifying event-driven and quantified temporal behaviors.

Second, we present a comprehensive verification approach that enables the checking of TOCL+ specifications using existing OCL 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. Our implementation defines practical translation patterns for temporal operators and event constructs, mapping them to structural navigations through snapshots and operation calls in the filmstrip model.

The chapter is organized as follows:

- Section 2.2 presents the specification approach, introducing TOCL and our TOCL+ extension.
- Section 2.3 details the verification approach using filmstrip models.
- Section 2.4 describes the implementation of TOCL+ to OCL transformation.
- Section 2.5 discusses related work in temporal specification and verification.
- Section 2.6 summarizes the chapter's contributions and findings.

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

This section introduces approaches for specifying temporal properties in UML/OCL models, focusing on extensions that address OCL's limitations with temporal and event-based constraints. We examine Temporal OCL (TOCL) as a foundational extension, then introduce our TOCL+ language that enhances temporal specification with event constructs and bounded existence operators. The formal definitions and grammar provided establish a precise foundation for both specification and subsequent verification of temporal properties.

2.2.1 Temporal OCL (TOCL)

Temporal OCL (TOCL), introduced by Ziemann and Gogolla [2], extends the Object Constraint Language (OCL) to address a fundamental limitation: the inability to specify constraints that span multiple system states over time. While standard OCL excels at defining invariants within a single state or basic pre/post-condition constraints for operations, it lacks the expressiveness needed for complex temporal behaviors. TOCL overcomes this limitation by incorporating elements from linear temporal logic while preserving OCL's familiar syntax and type system, making it particularly valuable for modeling reactive systems, concurrent processes, and real-time applications where temporal reasoning is essential.

TOCL defines its semantic framework over an infinite sequence of system states $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots \rangle$, where each temporal constraint is evaluated within an environment $\tau = (\hat{\sigma}, i, \beta)$. Here, i represents the current state index in the sequence, and β is a variable assignment. This framework enables reasoning about both the future and past evolution of system states, providing a comprehensive approach to temporal specification. Unlike standard OCL invariants that apply to a single state, TOCL invariants implicitly apply an "always" condition, meaning constraints must hold across the entire state sequence unless otherwise specified.

The expressive power of TOCL comes from its temporal operators, which enable precise specification of how properties evolve over time. These operators form the core of TOCL's extension to standard OCL, allowing constraints to reference past states, future states. By providing mechanisms to reason about both historical behavior and future evolution of the system, these operators significantly enhance OCL's ability to express dynamic properties.

TOCL organizes its temporal operators into two categories:

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.
- **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.

In TOCL, the expressions e, e_1, and e_2 represent boolean OCL expressions that evaluate to either true or false within a specific system state. These expressions may encompass standard OCL constructs, such as attribute access, navigation through associations, or operations on collections. The temporal operators specify the conditions under which these expressions must hold across a sequence of states, thereby extending OCL's traditionally static constraints into a dynamic, temporal framework.

The semantics of TOCL's temporal operators are formally grounded in state sequences and set theory, providing a rigorous basis for their evaluation. Consider a state sequence beginning at an initial state $i = 0$. The operator `next` e holds true at state i if e is true at state $i+1$, while `always` e requires e to hold for all states $j \geq i$. Conversely, `sometime` e is satisfied if e holds for at least one state $j \geq i$. For past-oriented operators, `previous` e is true at state i if e held at state $i-1$, with special consideration at the initial state ($i = 0$). These definitions enable precise reasoning about temporal relationships, making TOCL a powerful tool for specifying and verifying system properties.

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

Listing 2.1: TOCL Specification for Safety Properties

```
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)
    )
```

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.

2.2.2 TOCL+: An Extension of TOCL

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.

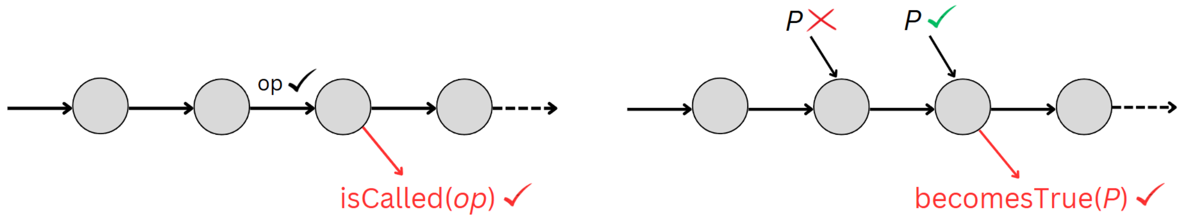


Figure 2.1: Events.

We adopt the concept of events from [10], 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 specifically for the `isCalled` event construct, allowing developers to specify constraints on operation call frequency:

- `isCalled(...) at most` k `times` - limiting operation call occurrences to no more than k
- `isCalled(...)` k `times` - requiring precisely k occurrences of an operation call
- `isCalled(...) at least` k `times` - requiring operation call 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:

These examples show TOCL+'s expressive power. With the `isCalled` construct, safety properties 1 and 2 directly reference operation calls rather

Listing 2.2: TOCL+ Specifications

```
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())
    )
```

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:

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

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:

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

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 specifically `isCalled` event occurrences within a temporal scope. For an `isCalled` event e_{ic} and temporal scope S (e.g., all past states for `sometimePast`):

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

Then:

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

These formal definitions provide a clean, precise semantics for TOCL+'s event constructs and bounded existence operators. By directly associating events with concrete state transitions and state changes, we establish a solid mathematical 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. [20] 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.

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

Listing 2.3: EBNF Grammar for Event constructs

```
primaryExp[Environment env] returns [ASTOclExpression ast]
        : literalExp[$env]
        | varExp[$env]
        | callExp[$env, null]
        | ifExp[$env]
        | toclOperatorExpression[$env]
        | LPAREN oclExpression[$env] RPAREN
        | events[$env] { $ast = $events.ast; }
        ;

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(); }
        ;
```

`sometimePast isCalled(op()) at most 3 times`, which combines a temporal operator, an event construct, and a bounded existence constraint.

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 [15]. 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).

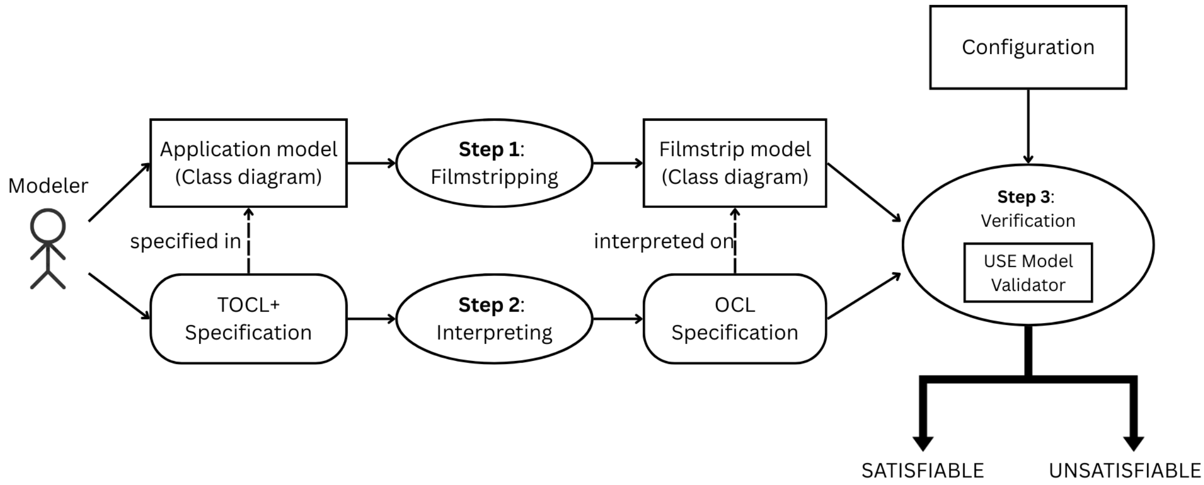


Figure 2.2: Verification approach.

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. 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.

Listing 2.4: Pre and post conditions for load operation

```
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
```

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+ tem-

Listing 2.5: Invariants for load_SystemOpC class

```
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
```

poral 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 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

Listing 2.6: OCL translation of liveness property

```
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)
        ))
    )
```

that follows, where they will be evaluated against potential system states to determine if the model satisfies the specified temporal properties. While this section has provided an overview of the translation approach, the detailed implementation of the transformation process from TOCL+ to OCL will be presented in the next section (Section 2.4).

2.3.3 Step 3: Verifying the translated OCL expressions

The final step in our verification approach employs the USE Model Validator [15] 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 2.3 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

```
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
```

Figure 2.3: Configuration file used to verify the liveness property.

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 2.4 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 2.4. 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

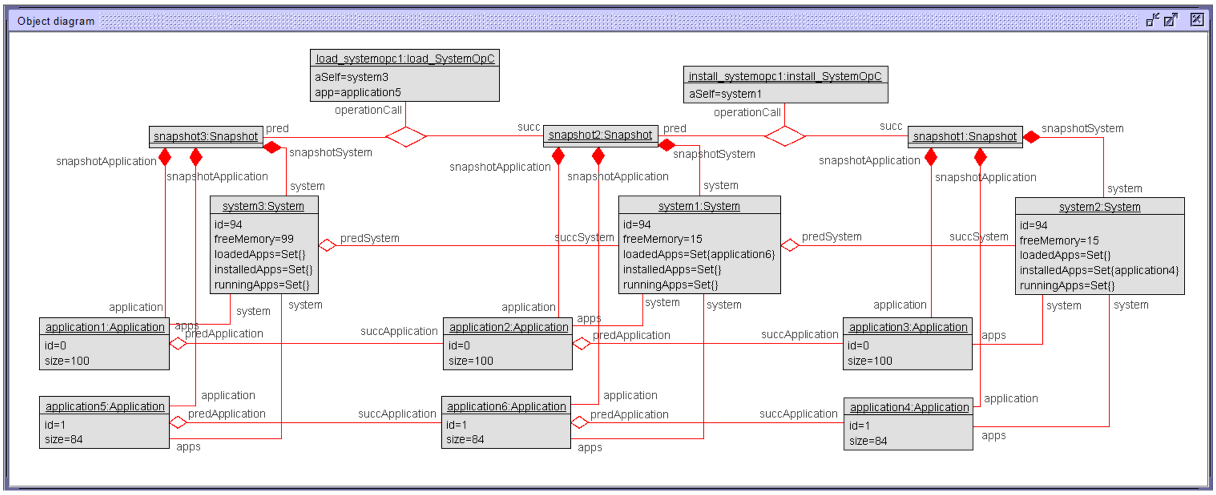


Figure 2.4: Object Diagram returned by the Model Validator.

standard filmstrip model, and we will explain the implementation details and necessary constraints for maintaining object identity in section 2.4. This validation confirms that our Software System model can satisfy the temporal behavior specified by the liveness property.

2.4 Implementation of TOCL+ to OCL Transformation

A central aspect of our approach is the systematic transformation of TOCL+ expressions into equivalent OCL constraints that can be verified over filmstrip models. This transformation process involves mapping temporal operators and event constructs to structural navigations through snapshots and operation calls. In this section, we describe our implementation approach and the key translation patterns we developed. Our approach provides practical, informal translation patterns that have been validated through our case studies. These patterns capture the intended semantics of TOCL+ expressions in terms of OCL navigations through the filmstrip model.

Our transformation approach is inspired by the work of [20], who transformed TOCL [2] into OCL in the context of a Snapshot-Transition Model (STM). While their approach also converts behavioural properties into static ones, we adapted and extended it to work within the filmstrip model context.

To transform TOCL+ expressions, we defined translations for TOCL+ operators and events to OCL, as shown in Table 2.1. To create these translations, we utilized several query operations provided by the filmstrip structure. The `self.snapshot` query accesses the snapshot associated with an object in the "current state" where the expression is being evaluated. The `pred()` and `succ()` operations, when applied to a snapshot, navigate to the previous and next state respectively. For objects to navigate to their corresponding versions in adjacent states, we use the two associations `.pred[ObjectClass]` and `.succ[ObjectClass]`. These navigation mechanisms form the foundation for implementing our temporal operator translations. The expressions contain square brackets that are placeholders; these placeholders will be explained later in this section.

Note that filmstrip model does not inherently provide any means to identify the same logical object across different states - it only provides the `.pred[ObjectClass]` and `.succ[ObjectClass]` associations to navigate between corresponding objects in adjacent snapshots. In order to overcome this limitation, we require modelers to add an `id` attribute to all classes in the application model. As seen in Figure 2.4, this allows us to identify the same logical object across different snapshots. Internally, when the TOCL+ plugin transforms the model, we add additional constraints (see Listing 2.7) to ensure this `id` remains consistent between different states. This `id` attribute is critical in the OCL translation, particularly for event constructs like `becomesTrue`. When navigating with expressions like `self.snapshot.pred()-.[ContextObject]`, we get a collection of objects of the same type, and we select the one with matching identity using `->any(o | o.id = self.id)`.

Table 2.1 presents the complete set of translation patterns we developed for TOCL+ operators and event constructs. Each pattern systematically maps a TOCL+ construct to an equivalent OCL expression interpreted

Listing 2.7: Additional constraints

```
context Application
inv ApplicationSameId:
    self.succApplication.isDefined()
    implies
    self.id = self.succApplication.id
context System
inv SystemSameId:
    self.succSystem.isDefined()
    implies
    self.id = self.succSystem.id

context Application
inv ApplicationDiffId:
    self.snapshotApplication.application->forAll(o | o <> self implies o.id <>
        self.id)
context System
inv SystemDiffId:
    self.snapshotSystem.system->forAll(o | o <> self implies o.id <> self.id)
```

over the filmstrip model. The patterns use placeholders (indicated by square brackets) that get substituted during the transformation process: `[s |= P]` indicates that property P holds in snapshot s; `[ContextSnapshot]` is the snapshot in which the property is evaluated; `[ContextObject]` is the object on which the property is evaluated; `[OpClassName]` is the class representing an operation call in the filmstrip model; `[ObjectClass]` is the class name of the current object; `[ContextClass]` is the class name representing the context of the current evaluation. When applying these patterns, each placeholder is replaced with the appropriate expression based on the model context. For example, in the liveness property translation shown in Listing 2.6, the `[ContextSnapshot]` is replaced with the local variable `s` inside the `exists` query; the bounded quantifiers (e.g., `at most`, `at least`) are translated into corresponding comparators (`<=`, `>=`). The bounded quantifiers in TOCL+ expressions are systematically translated into their corresponding OCL comparators: `at most` becomes less than or equal to (`<=`), while `at least` becomes greater than or equal to (`>=`). For bounded existence properties without an explicit quantifier (e.g., `isCalled(Op()) 3 times`), the translation applies the equality operator (`=`), enforcing that the event occurs exactly the specified number of times. In contrast, when dealing with unbounded event expressions without quantifiers (e.g., simply `isCalled(Op())`), the translation converts the `select` operation into an `exists` operation, requiring only that the event occurs at least once rather than counting occurrences.

We implemented the transformation using ANTLR4, a parser generator that creates a parse tree from TOCL+ expressions. After defining the translation patterns shown in Table 2.1, we created Java listener classes that extend the generated parser listeners. The transformation process employs these listener components to traverse the parse tree and produce corresponding OCL expressions by overriding the generated listener methods. As the parser walks through each node in the parse tree, our listeners intercept parse tree events and apply the appropriate translation rules, constructing equivalent OCL constraints that navigate through the filmstrip structure. The transformation follows a consistent pattern: when a temporal operator or event construct is encountered, the listener extracts relevant information from the parse tree nodes, applies the corresponding translation pattern, and builds the equivalent OCL expression. Our implementation maintains a stack of expressions to handle nested structures, pushing both the original TOCL+ expression and its OCL translation for later integration into the output model. Listing 2.8 provides an example of this process for the `becomesTrue` event construct, showing how we extract the expression to be evaluated, establish the necessary context, and construct the translated OCL expression.

After processing all the nodes in the parse tree, our implementation finalizes the translation in the root node visit. Since TOCL+ is an extension of OCL, many constructs remain unchanged and are directly preserved during translation. The completion process begins by accessing the token stream to retrieve the complete original expression text. Our implementation then systematically pops each translated OCL fragment and its corresponding

Listing 2.8: Translation of becomesTrue event to OCL

```
TokenStream tokens = parser.getTokenStream();
String originalEvent = tokens.getText(ctx);
String translatedEvent;

// P
String expressionToSatisfy = getOCL(ctx.getChild(2));
// e.g., "system", "application"
String roleName = toLowerFirstChar(currentContext);
String currentSnapshot = "self.snapshot";
String selectObject = "->any(o | o.id = self.id)";

// e.g. self.snapshot.system->any(o | o.id = self.id)
String objectAtCurrentSnapshot = currentSnapshot + "." + roleName + selectObject;
String objectAtPreviousSnapshot = currentSnapshot + ".pred()." + roleName + selectObject;
String P_at_currentSnapshot = expressionToSatisfy.replace("self", "currentObject");
String P_at_previousSnapshot = expressionToSatisfy.replace("self", "previousObject");

translatedEvent =
"let currentObject = " + objectAtCurrentSnapshot +
" in let previousObject = " + objectAtPreviousSnapshot +
" in not (" + P_at_previousSnapshot + ") and (" + P_at_currentSnapshot + ")";

eventStack.push(translatedEvent);
eventStack.push(originalEvent);
```

original TOCL+ expression from the stack. Using string replacement operations, it substitutes each temporal construct with its equivalent OCL translation while preserving the structure of the original expression. This approach allows us to handle nested temporal operators naturally, as inner expressions are processed before their containing expressions. The final translated OCL constraint is then attached to the root of the parse tree and ultimately added to the output model as an invariant. This complete process ensures that complex temporal properties are accurately transformed into standard OCL constraints that can be verified by the Model Validator.

Listing 2.9 shows a TOCL+ specification for the liveness property using the `becomesTrue` event construct, while Listing 2.10 displays its corresponding OCL translation produced by the process presented in Listing 2.8. This example illustrates our transformation approach, converting a concise TOCL+ expression into an equivalent OCL constraint that operates within the filmstrip model structure.

Listing 2.9: Liveness property specification using becomesTrue event

```
context Application
inv livenessBecomesTrue:
    self.system.loadedApps->includes(self) implies
    sometime becomesTrue(self.system.installedApps->includes(self))
```

Listing 2.10: OCL translation of liveness property in Listing 2.9

```
context Application
inv livenessBecomesTrue:
    self.system.loadedApps->includes(self) implies
    (let CS:Snapshot = self.snapshotApplication in Set{CS}->closure(s | s.succ
        ())->excluding(null)->exists(s | (let currentObject = s.application->any
        (o | o.id = self.id) in let previousObject = s.pred().application->any(o
         | o.id = self.id) in not (previousObject.system.installedApps->includes
        (previousObject)) and (currentObject.system.installedApps->includes(
        currentObject)))))
```

Table 2.1 Translation of TOCL+ operators and events to OCL

No.	TOCL+	OCL Translation
1	next P	let nextSnapshot:Snapshot = self.snapshot[ContextClass].succ() in [nextSnapshot |= P]
2	always P	let CS:Snapshot = self.snapshot[ContextClass] in Set{CS}->closure(s | s.succ())->forAll(s | [s |= P])
3	always P until Q	let CS:Snapshot = self.snapshot[ContextClass] 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

<table>
<tr><td>4</td><td>always P since Q</td><td>let CS:Snapshot = self.snapshot[ContextClass] 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[ContextClass] 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[ContextClass]}->closure(s | s.succ()) in let PreS:Set(Snapshot) = Set{self.snapshot[ContextClass].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[ContextClass] 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[ContextClass].pred() in [previousSnapshot |= P]</td></tr>
<tr><td>9</td><td>sometimePast P</td><td>let CS:Snapshot = self.snapshot[ContextClass] 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[ContextClass] in Set{CS.pred()}->closure(s | s.pred())->forAll(s | [s |= P])</td></tr>
<tr><td>11</td><td>isCalled(Op())</td><td>[OpClassName].allInstances()->exists(op | op.succ() = [ContextSnapshot])</td></tr>
<tr><td>12</td><td>isCalled(Op($param_1$, …, $param_n$))</td><td>[OpClassName].allInstances()->exists(op | op.succ() = [ContextSnapshot] and (Set{op.$param_1$.succ[ObjectClass]}->closure(p | p.succ[ObjectClass])->includes($param_1$) or Set{op.$param_1$.pred[ObjectClass]}->closure(p | p.pred[ObjectClass])->includes($param_1$)) and (…) and (Set{op.$param_n$.succ[ObjectClass]}->closure(p | p.succ[ObjectClass])->includes($param_n$) or Set{op.$param_n$.pred[ObjectClass]}->closure(p | p.pred[ObjectClass])->includes($param_n$)))</td></tr>
</table>

<table>
<tr><td>13</td><td>isCalled(Op()) [at most | at least |] n times</td><td>[OpClassName].allInstances()->select(op | op.succ() = [ContextSnapshot])->size() [<= | >= | =] n</td></tr>
<tr><td>14</td><td>isCalled(Op($param_1$, ..., $param_n$)) [at most | at least |] n times</td><td>[OpClassName].allInstances()->select(op | op.succ() = [ContextSnapshot] and (Set{op.$param_1$.succ[ObjectClass]}->closure(p | p.succ[ObjectClass])->includes($param_1$) or Set{op.$param_1$.pred[ObjectClass]}->closure(p | p.pred[ObjectClass])->includes($param_1$)) and (...) and (Set{op.$param_n$.succ[ObjectClass]}->closure(p | p.succ[ObjectClass])->includes($param_n$) or Set{op.$param_n$.pred[ObjectClass]}->closure(p | p.pred[ObjectClass])->includes($param_n$)))->size() [<= | >= | =] n</td></tr>
<tr><td>15</td><td>becomesTrue(P)</td><td>let currentObject = self.snapshot[ContextClass].[ContextObject]->any(o | o.id = self.id) in let previousObject = self.snapshot[ContextClass].pred().[ContextObject]->any(o | o.id = self.id) in not [previousObject |= P] and [currentObject |= P]</td></tr>
</table>

2.5 Related Work

Several approaches have addressed the challenge of specifying and verifying temporal properties in UML models. Al Lail et al. [21] proposed a technique similar to ours, transforming application models into Snapshot Transition Models (STM) and using TOCL for specification, but their approach inherits TOCL's limitations with event-based properties. Kanso and Taha [10] and Dadeau et al. [18] explored pattern-based specification approaches, implementing patterns in Eclipse but without providing verification capabilities, or focusing on model-based testing rather than comprehensive verification. Baresi et al. [12] present the MADES approach, which combines several heavyweight formalisms for verifying embedded systems, but requires steep learning curves and mathematical skills that make it inaccessible for many UML designers. Similarly, Combemale et al. [6] use a temporal extension to OCL with translation to Petri nets, requiring designers to learn additional formalisms to understand verification results. Meyers et al. [11] support verifying temporal properties in domain-specific modeling through a family of five languages, but rely on LTL and the Spin model checker, creating a separation between specification and verification languages. Hilken and Gogolla [14] presented a filmstripping approach similar to our verification technique, but without support for high-level temporal property specification. In contrast, our approach unifies specification and verification within the familiar OCL context, eliminating the need to learn additional formalisms.

2.6 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.

3.2 TOCL+ Tool Support and Implementation

Figure 3.1 illustrates the architecture and workflow of our TOCL+ support tool. This integrated verification framework combines several components: the existing USE environment, the Filmstrip plugin, our TOCL+ plugin, and the Model Validator. The architecture maintains a clear separation between modeling, specification, and verification concerns while providing a cohesive workflow for users.

The verification process consists of five distinct steps, beginning with preparation and ending with validation. 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. These generated constraints are added to the list of invariants in the output model file alongside the filmstrip model elements.

Finally, the user loads the output model into USE with a configuration file setting search bounds, then uses the Model Validator to analyze the constraints. The validator explores the search space to determine property satisfaction and provides model instances as evidence when applicable.

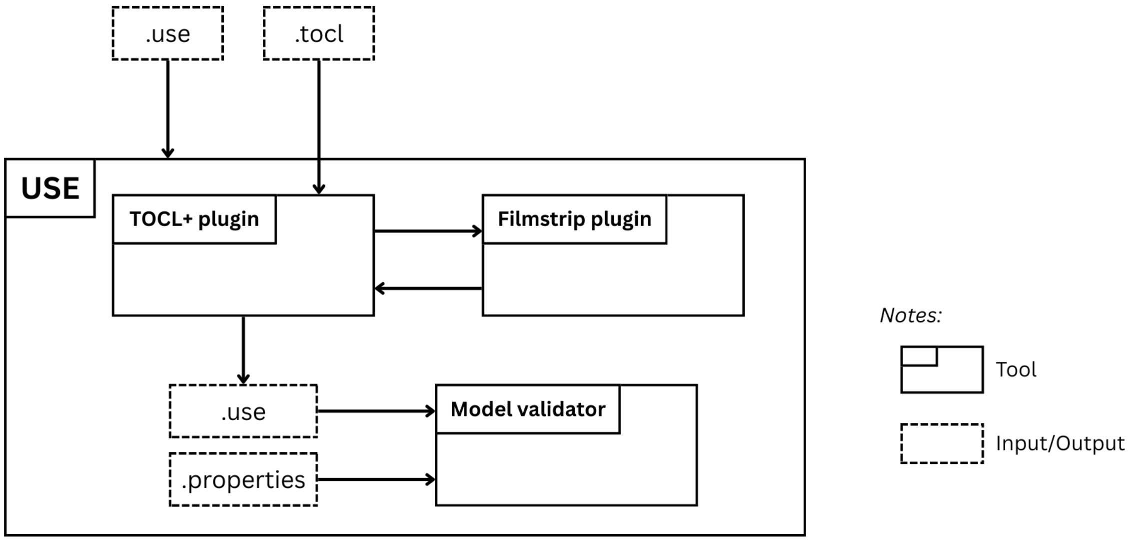


Figure 3.1 Support Tool Architecture and Verification Workflow.

Our primary contribution in this architecture is the TOCL+ plugin, specifically the TOCL+ to OCL transformation component that enables the verification of temporal properties using existing OCL tools. The implementation details of this transformation process, including the translation rules for different temporal operators and event constructs, were presented previously in Section 2.4. Our implementation follows the transformation patterns established in Chapter 2 and integrates them into the workflow described above.

3.3 Case study: Software System model

3.3.1 Model Specification

The Software System model shown in Listing 3.5 is the same model introduced in Chapter 1 to demonstrate OCL constraints (see Figure 1.3). As a reminder, this model represents a simplified operating system that manages applications through their lifecycle. It consists of two main classes: `System` and `Application`, connected through an association. The `System` maintains three collections (`loadedApps`, `installedApps`, and `runningApps`) representing different application states, and provides four operations to manage applications: `load`, `install`, `run`, and `stop`.

The `freeMemory` attribute represents available disk space, which decreases when applications are loaded. The `load` operation acts as a download action, reducing available memory when an application is acquired. The `install` operation processes all loaded applications at once, moving them from `loadedApps` to `installedApps`. When an application is running, it exists in both the `installedApps` and `runningApps` sets simultaneously. The `SystemApplication` association facilitates navigation between system and applications in this simplified model.

Note that the numerous postconditions like `sameInstalledAndRunning`, `sameRunning`, `sameMemory`, `sameLoaded`, `sameInstalled`, and the `unchanged` constraints serve as helper constraints in the verification context: since the Model Validator assigns random values to attributes when exploring possible states, these constraints ensure that attributes unaffected by an operation remain consistent between snapshots.

This model serves as an ideal case study for temporal property verification as it involves operations with clear sequential dependencies and state transitions that cannot be adequately expressed using standard OCL. The complete specification, including all constraints and operation contracts necessary for our verification experiments, is provided in Appendix A. This appendix contains the full USE model specification that forms the foundation for our temporal property verification.

3.3.2 Temporal Property Verification

To demonstrate the effectiveness of our TOCL+ to OCL transformation approach, we verified four temporal properties against the Software System model: safety1 (applications must be loaded before being run), safety2 (applications must follow the load-install-run sequence), safety3 (applications can be loaded at most once), and liveness (loaded applications eventually become installed). These properties, originally introduced in Chapter 2, exercise different aspects of TOCL+'s expressiveness. Listing 2.2 presents the complete TOCL+ specifications. Additionally, we developed an alternative formulation of the liveness property using the `becomesTrue` event construct (Listing 2.9), illustrating TOCL+'s flexibility in expressing semantically equivalent properties through different syntactic constructs.

Listing 3.1 shows the OCL constraints generated by our transformation plugin for these properties. While the resulting OCL expressions are considerably more complex than their TOCL+ counterparts—involving extensive navigation through the filmstrip model—they correctly implement the required temporal semantics. This demonstrates how our approach enables temporal reasoning within standard OCL, shielding modelers from the underlying complexity.

Listing 3.1 OCL Translations for TOCL+ properties shown in listing 2.2

```
context System
inv safety1:
  self.runningApps->notEmpty() implies
  self.runningApps->forAll(app |
    (run_SystemOpC.allInstances()->exists(op |
      op.succ() = self.snapshotSystem and
      (Set{op.app.succApplication}->closure(p | p.succApplication)->includes(
          app)
       or
       Set{op.app.predApplication}->closure(p | p.predApplication)->includes(
           app))
    ))
    implies
    (let CS:Snapshot = self.snapshotSystem in Set{CS.pred()}->closure(s | s.
        pred())->exists(s | (load_SystemOpC.allInstances()->exists(op | op.succ
```

```
            () = s and (Set{op.app.succApplication}->closure(p | p.succApplication)
            ->includes(app) or Set{op.app.predApplication}->closure(p | p.
            predApplication)->includes(app))))))
    )

context System
inv safety2:
    self.runningApps->notEmpty() implies
    self.runningApps->forAll(app |
        (run_SystemOpC.allInstances()->exists(op | op.succ() = self.
            snapshotSystem and (Set{op.app.succApplication}->closure(p | p.
            succApplication)->includes(app) or Set{op.app.predApplication}->
            closure(p | p.predApplication)->includes(app)))) implies
        (let CS:Snapshot = self.snapshotSystem in let PS:Set(Snapshot) = Set{CS
            .pred()}->closure(s | s.pred())->excluding(null) in let AllPSQ:Set(
            Snapshot) = PS->select(s | (load_SystemOpC.allInstances()->exists(op
             | op.succ() = s and (Set{op.app.succApplication}->closure(p | p.
            succApplication)->includes(app) or Set{op.app.predApplication}->
            closure(p | p.predApplication)->includes(app))))) in let PSQ:
            Snapshot = AllPSQ->any(s | Set{s}->closure(s | s.pred())->
            includesAll(AllPSQ)) in let PSP:Set(Snapshot) = PS->select(s |
            install_SystemOpC.allInstances()->exists(op | op.succ() = s)) in if
            PSQ.isDefined() then (Set{PSQ}->closure(s | s.succ())->excluding(
            null)->intersection(PS)->exists(s | PSP->includes(s))) else false
            endif)
    )

context System
inv safety3:
    self.installedApps->notEmpty() implies
    self.installedApps->forAll(app |
        (let CS:Snapshot = self.snapshotSystem in Set{CS.pred()}->closure(s | s
            .pred())->exists(s | (load_SystemOpC.allInstances()->select(op | op.
            succ() = s and (Set{op.app.succApplication}->closure(p | p.
            succApplication)->includes(app) or Set{op.app.predApplication}->
            closure(p | p.predApplication)->includes(app)))->size() <= 1)))
    )

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)))
    )

context Application
inv livenessBecomesTrue:
    self.system.loadedApps->includes(self) implies
    (let CS:Snapshot = self.snapshotApplication in Set{CS}->closure(s | s.succ
```

```
())->excluding(null)->exists(s | (let currentObject = s.application->any
(o | o.id = self.id) in let previousObject = s.pred().application->any(o
 | o.id = self.id) in not (previousObject.system.installedApps->includes
(previousObject)) and (currentObject.system.installedApps->includes(
currentObject)))))
```

3.3.3 Analysis of Results

Figure 3.2 shows a concrete scenario generated by the USE Model Validator when verifying our temporal properties. This scenario visualizes a complete application lifecycle through the software system, illustrating a sequence of operation calls: loading an application, installing it, running it, and finally stopping it. This execution path is particularly valuable as it exercises all four operations of our model in their expected sequence.

As shown in Figure 3.3, all temporal properties are satisfied in this scenario. Each property verification confirms an important aspect of our system's behavior: safety1 verifies that the application was indeed loaded before being run; safety2 confirms the correct operational sequence was followed (load→install→run); safety3 validates that the application was loaded exactly once; and the liveness property confirms that after being loaded, the application was eventually installed.

These results demonstrate two important aspects of our approach. First, they validate the correctness of our transformation rules by confirming that the generated OCL constraints accurately encode the intended temporal semantics. Despite their complexity, the translated constraints correctly identify valid execution paths. Second, they show how our approach supports automated verification of complex temporal properties that would be impossible to express in standard OCL.

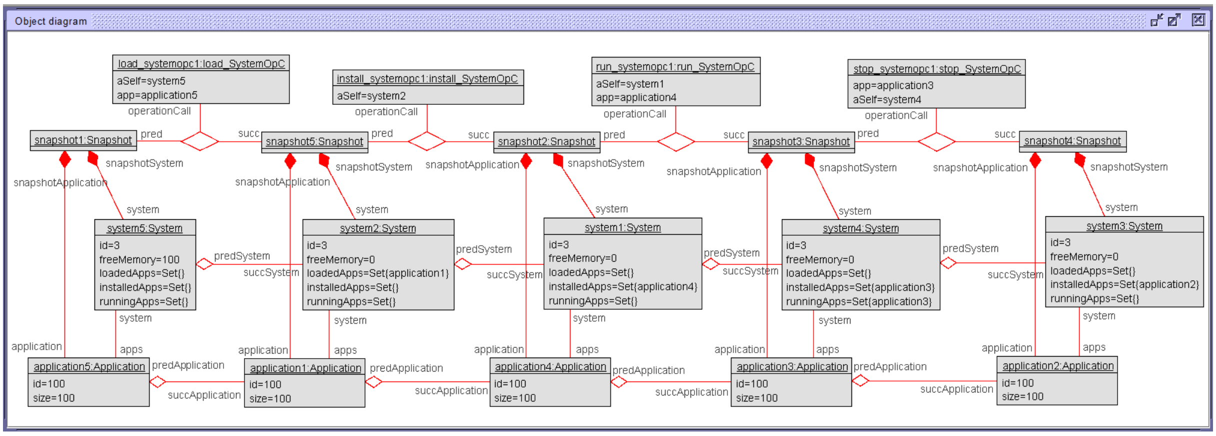
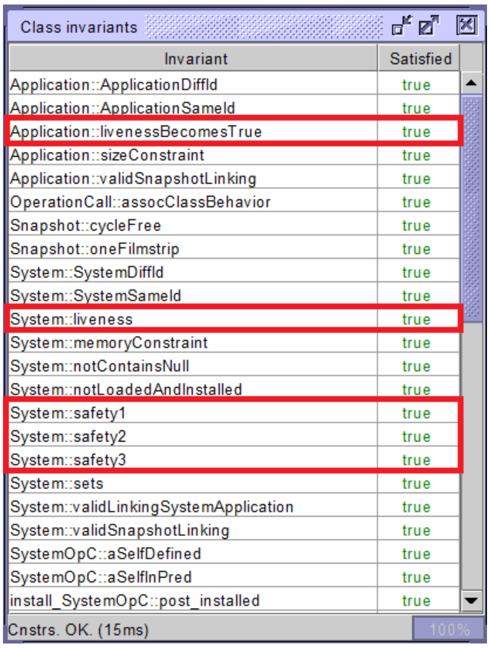


Figure 3.2 A scenario generated by the USE Model Validator.



Class invariants

Invariant	Satisfied
Application::ApplicationDiffId	true
Application::ApplicationSameId	true
Application::livenessBecomesTrue	true
Application::sizeConstraint	true
Application::validSnapshotLinking	true
OperationCall::assocClassBehavior	true
Snapshot::cycleFree	true
Snapshot::oneFilmstrip	true
System::SystemDiffId	true
System::SystemSameId	true
System::liveness	true
System::memoryConstraint	true
System::notContainsNull	true
System::notLoadedAndInstalled	true
System::safety1	true
System::safety2	true
System::safety3	true
System::sets	true
System::validLinkingSystemApplication	true
System::validSnapshotLinking	true
SystemOpC::aSelfDefined	true
SystemOpC::aSelfInPred	true
install_SystemOpC::post_installed	true

Cnstrs. OK. (15ms) 100%

Figure 3.3 OCL result for generated scenario 3.2.

3.4 Discussion

Our TOCL+ approach for specifying and verifying temporal properties in UML/OCL models offers several advantages over existing approaches but also comes with certain limitations that warrant discussion.

The primary strength of our approach lies in its integration with standard modeling environments and tools. By extending OCL rather than introducing an entirely new formalism, we maintain compatibility with existing modeling practices while adding temporal verification capabilities. This integration allows modelers to work within familiar environments while gaining the ability to verify a broader class of properties. Our transformation-based approach also offers considerable flexibility. By converting TOCL+ specifications to standard OCL constraints on filmstrip models, we leverage established OCL tool capabilities without requiring specialized temporal verification engines. This provides a pragmatic solution that can be adopted without significant infrastructure changes or learning costs. The event-based constructs in TOCL+ address a significant gap in existing temporal OCL extensions. While previous approaches like TOCL provide temporal operators, they lack robust support for event detection and bounded existence properties. Our extensions make it possible to express important constraints related to operation calls and state changes, significantly broadening the range of verifiable properties.

While effective in practice, our approach has several technical limitations. The OCL constraints generated by our transformation can be complex, potentially affecting verification performance for large models with numerous temporal properties. These constraints involve intricate navigation through snapshots and careful handling of object identity, which may impact scalability for very large systems. Our approach also requires modelers to add identifier attributes to domain classes to maintain object identity across snapshots. This requirement introduces a small burden on modelers and slightly modifies the original domain model. A more elegant solution would be to handle object identity tracking automatically within the transformation framework. The current implementation also has limited support for complex expressions within event specifications. For example, nested event expressions or complex guards on events might not translate correctly in all cases. This restricts the sophistication of temporal properties that can be reliably verified.

From a methodological perspective, our work lacks a formal definition of the TOCL+ language and its transformation to OCL. Although our patterns appear to work correctly based on empirical evidence from case studies, we haven't provided formal proofs of semantic preservation between TOCL+ expressions and their OCL translations. This represents a theoretical gap that could affect confidence in the correctness of verification results for complex scenarios. Additionally, while our approach handles the core temporal operators and event constructs well, it doesn't yet support probabilistic or real-time properties. Systems with strict timing requirements or probabilistic behavior patterns would require extensions beyond the current capabilities.

3.5 Summary

In this chapter, we presented the practical implementation of our TOCL+ approach through a plugin for the USE tool and demonstrated its effectiveness through a detailed case study. The implementation bridges the gap between the theoretical foundations established in Chapter 2 and practical model verification by enabling modelers to specify temporal properties in a high-level language while leveraging existing OCL verification tools. Our plugin successfully implements the transformation rules that convert TOCL+ expressions into equivalent OCL constraints interpreted over filmstrip models. The implementation uses ANTLR4 for parsing TOCL+ expressions and employs a listener-based approach to systematically translate temporal operators and event constructs into structural OCL navigations.

The Software System case study demonstrated how our approach enables verification of diverse temporal properties that would be impractical to express in standard OCL. The safety properties successfully enforced correct operational sequencing and uniqueness constraints, while the liveness property verified eventual progress in the system. The complexity of the generated

OCL constraints highlights the value of our approach: while these constraints involve intricate navigation through snapshots and require careful handling of object identity, the TOCL+ specification remains simple and intuitive.

The verification results across different temporal properties and scenarios provide confidence in the practical applicability of our approach to realistic modeling challenges. Despite the increased complexity of the generated constraints, the verification performance remained acceptable for our test cases. This chapter thus validates our approach experimentally, showing how temporal verification can be effectively integrated into standard UML/OCL modeling environments without requiring specialized temporal verification tools.

Conclusion

This thesis has addressed the challenge of specifying and verifying temporal properties in UML/OCL models without requiring specialized temporal verification tools. By extending OCL with temporal operators and event constructs, and providing a transformation mechanism to standard OCL, we have created an approach that enables modelers to verify complex behavioral properties while remaining within familiar modeling environments.

Our work has made several significant contributions to the field of model verification. First, we introduced TOCL+ (Temporal OCL+), an extension of OCL that incorporates temporal operators and event constructs that enable expressing complex temporal properties in a concise and intuitive way. Second, we developed a transformation approach that converts TOCL+ expressions into equivalent standard OCL constraints that can be verified over filmstrip models, leveraging existing OCL tools without requiring specialized temporal logic model checkers.

Third, we implemented our approach as a plugin for the USE tool, demonstrating its practical applicability. The plugin uses ANTLR4 for parsing TOCL+ expressions and employs a listener-based approach to generate the corresponding OCL constraints. Fourth, we validated our approach through a detailed case study of a Software System model, verifying various types of temporal properties including safety and liveness constraints.

The significance of our work lies in making temporal verification accessible within standard modeling environments. Our approach enables comprehensive model verification without requiring modelers to learn specialized temporal logics or verification tools. This is particularly valuable for verifying behavioral properties that cannot be expressed in standard OCL, such as correct operational sequencing, eventual progress, and bounded existence constraints.

Several directions for future work will build upon the foundation established in this research. We plan to define formal transformation rules for converting TOCL+ to OCL, providing rigorous mathematical underpinnings for our approach. Additionally, we intend to develop formal verification techniques for the translation process to ensure semantic preservation between the original specifications and their OCL translations. To further validate our approach, we will apply the method to more diverse and complex case studies across different application domains. We also aim to address current limitations of the filmstrip model, particularly related to performance and scalability for larger systems. Finally, we plan to integrate formal property specification patterns to reduce the effort required to specify common temporal properties, making our approach even more accessible to practitioners.

REFERENCES

[1] Amir Pnueli. "The temporal logic of programs". In: *18th annual symposium on foundations of computer science (sfcs 1977)*. IEEE. 1977, pp. 46–57.

[2] Paul Ziemann and Martin Gogolla. "OCL extended with temporal logic". In: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer. 2003, pp. 351–357.

[3] Robert France and Bernhard Rumpe. "Model-driven development of complex software: A research roadmap". In: *Future of Software Engineering (FOSE'07)*. IEEE. 2007, pp. 37–54.

[4] Martin Gogolla, Fabian Büttner, and Mark Richters. "USE: A UML-based specification environment for validating UML and OCL". In: *Science of Computer Programming* 69.1-3 (2007), pp. 27–34.

[5] Emina Torlak and Daniel Jackson. "Kodkod: A relational model finder". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2007, pp. 632–647.

[6] Benoıt Combemale et al. "A property-driven approach to formal verification of process models". In: *Enterprise Information Systems: 9th International Conference, ICEIS 2007, Funchal, Madeira, June 12-16, 2007, Revised Selected Papers 9*. Springer. 2008, pp. 286–300.

[7] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. "Extensive validation of OCL models by integrating SAT solving into USE". In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2011, pp. 290–306.

[8] Martin Gogolla et al. "From application models to filmstrip models: An approach to automatic validation of model dynamics". In: *Modellierung 2014*. Gesellschaft für Informatik eV. 2014, pp. 273–288.

[9] Frank Hilken, Lars Hamann, and Martin Gogolla. "Transformation of UML and OCL models into filmstrip models". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2014, pp. 170–185.

[10] Bilal Kanso and Safouan Taha. “Specification of temporal properties with OCL”. In: *Science of Computer programming* 96 (2014), pp. 527–551.

[11] Bart Meyers et al. “ProMoBox: a framework for generating domain-specific property languages”. In: *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings 7*. Springer. 2014, pp. 1–20.

[12] Luciano Baresi et al. “Formal verification and validation of embedded systems: the UML-based MADES approach”. In: *Software & Systems Modeling* 14 (2015), pp. 343–363.

[13] Martin Gogolla and Frank Hilken. “Model validation and verification options in a contemporary UML and OCL analysis tool”. In: *Modellierung 2016*. Gesellschaft für Informatik eV. 2016, pp. 205–220.

[14] Frank Hilken and Martin Gogolla. “Verifying linear temporal logic properties in UML/OCL class diagrams using filmstripping”. In: *2016 Euromicro Conference on Digital System Design (DSD)*. IEEE. 2016, pp. 708–713.

[15] Nils Przigoda et al. “Integrating an SMT-Based ModelFinder into USE.”. In: *MoDeVVa@ MoDELS*. Citeseer. 2016, pp. 40–45.

[16] *Object Constraint Language Specification*. Object Management Group, 2017. URL: https://www.omg.org/spec/OCL/2.5.1.

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

[18] Frédéric Dadeau, Elizabeta Fourneret, and Abir Bouchelaghem. “Temporal property patterns for model-based testing from UML/OCL”. In: *Software & Systems Modeling* 18 (2019), pp. 865–888.

[19] Lvar Jacobson and James Rumbaugh Grady Booch. *The unified modeling language reference manual*. 2021.

[20] 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*. 2022, pp. 899–907.

[21] Mustafa Al Lail et al. “Tpv: A tool for validating temporal properties in uml class diagrams”. In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 2024, pp. 114–118.

APPENDIX

A. The USE specification for Software System case study.

```
model SoftwareSystem
-- Classes
class System
attributes
    id : Integer
    freeMemory : Integer init = 10
    loadedApps : Set(Application) init = Set{}
    installedApps : Set(Application) init = Set{}
    runningApps : Set(Application) init = Set{}
operations
    load(app : Application)
    begin
        self.loadedApps := self.loadedApps->including(app);
        self.freeMemory := self.freeMemory - app.size;
    end
    install()
    begin
        self.installedApps := self.installedApps->union(self.loadedApps);
        self.loadedApps := self.loadedApps->reject(true)->excluding(null);
    end
    run(app : Application)
    begin
        self.runningApps := self.runningApps->including(app);
    end
    stop(app : Application)
    begin
        self.runningApps := self.runningApps->excluding(app);
    end
end
class Application
attributes
    id : Integer
    size : Integer
end
-- Associations
association SystemApplication between
    System[1] role system
    Application[0..*] role apps
end
-- Invariants
constraints
context System
    inv memoryConstraint: self.freeMemory >= 0
    inv notLoadedAndInstalled: self.loadedApps->intersection(self.installedApps)->isEmpty()
```

```
    inv sets: let appNumber: Integer = self.apps->size() in
        (self.loadedApps->size() <= appNumber and
        self.installedApps->size() <= appNumber and
        self.runningApps->size() <= appNumber)
    inv notContainsNull:
        not self.loadedApps->includes(null) and
        not self.installedApps->includes(null) and
        not self.runningApps->includes(null)

context Application
    inv sizeConstraint: self.size > 0

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 freeMemory: self.freeMemory = self.freeMemory@pre - app.size
    post unchanged:
        self.apps->forAll(app |
            app.size = app.size@pre and
            app.id = app.id@pre)
    post sameInstalledAndRunning:
        self.installedApps = self.installedApps@pre and
        self.runningApps = self.runningApps@pre

context System::install()
    pre hasLoadedApps: self.loadedApps->notEmpty()
    post installed: self.installedApps = self.installedApps@pre->union(self.loadedApps@pre)
    post loadedAppsEmpty: self.loadedApps = self.loadedApps@pre->reject(true)->excluding(null)
    post sameRunning: self.runningApps = self.runningApps@pre
    post sameMemory: self.freeMemory = self.freeMemory@pre
    post unchanged:
        self.apps->forAll(app |
            app.size = app.size@pre and
            app.id = app.id@pre)

context System::run(app : Application)
    pre isInstalled: self.installedApps->includes(app)
    pre notRunning: not self.runningApps->includes(app)
    post running: self.runningApps = self.runningApps@pre->including(app)
    post sameLoaded: self.loadedApps = self.loadedApps@pre
    post sameInstalled: self.installedApps = self.installedApps@pre
    post sameMemory: self.freeMemory = self.freeMemory@pre
    post unchanged:
        self.apps->forAll(app |
            app.size = app.size@pre and
            app.id = app.id@pre)
```

```
context System::stop(app : Application)
    pre isRunning: self.runningApps->includes(app)
            and self.installedApps->includes(app)
    post notRunning: self.runningApps = self.runningApps@pre->excluding(app)
    post sameInstalled: self.installedApps = self.installedApps@pre
    post sameLoaded: self.loadedApps = self.loadedApps@pre
    post sameMemory: self.freeMemory = self.freeMemory@pre
    post unchanged:
        self.apps->forAll(app |
            app.size = app.size@pre and
            app.id = app.id@pre)
```