FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Towards Software Model Checking in the Context of Model–Driven Engineering

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom–Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Robert Bill
Matrikelnummer 0727135

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Mitwirkung: Mag. Dr. Petra Kaufmann, Dipl.-Ing. Sebastian Gabmeyer
Dipl.-Ing. Dr. Martina Seidl

Wien, 19.03.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Robert Bill
Lichnowskygasse 12, 1110 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

Einen besonderen Dank möchte ich an meine Betreuerinnen und meinen Betreuer für die Hilfestellung bei Formulierung, Strukturierung und Formatierung geben. Sie haben mich nicht nur auf die Idee gebracht, diese Arbeit zu schreiben und aktiv während der gesamten Arbeit begleitet, sondern waren auch immer für Fragen offen, stellten einen ruhigen Platz zur Verfügung und ermöglichten ein reibungsloses Ablaufen der Evaluierung. So konnte auch ein Teil der Arbeit für ein Workshop–Paper[1] verwendet werden, das auf dem 13th International Workshop on OCL, Model Constraint and Query Languages (OCL 2013) veröffentlicht wurde.

Ich möchte auch allen, die sich teilweise viel Zeit für die Evaluierung genommen haben, danken, denn ohne sie wäre ein großer Teil der Arbeit nicht möglich gewesen und viele kleinere Verbesserungen im Tool nicht zustandegekommen.

Großer Dank geht auch an meine Eltern, die mir mein Studium nicht nur finanziell ermöglicht haben, sondern mich in meinen Entscheidungen auch unterstützten.

Weiters bedanke ich mich bei allen, die mich während des Studiums und der Arbeit moralisch unterstützt haben; insbesondere bei meiner Großmutter Gertrude und meiner Bekannten Christa, die sich für die Arbeit interessierten, obwohl sie sich selbst nicht tiefergehend mit Computern beschäftigen.

[1] `http://publik.tuwien.ac.at/files/PubDat_221651.pdf`

Abstract

The aim of this master thesis is to reduce the conceptual gap between software modeling and model checking. While model checking is successfully applied for hardware verification, it is not widespreadly used in model–driven engineering (MDE). Thus, we tried to reduce this gap by combining modeling and model checking concepts.

This thesis first describes the history and basic idea of both MDE and model checking with a focus on the technologies used in this thesis. Before presenting our new approach, existing solutions are compared. Most approaches propose to extend the Object Constraint Language (OCL) by temporal aspects. This allows to describe the behavior of a software system additionally to various properties of static models. However, one of the main missing features in general seems to be a user–friendly representation of the verification result helpful for debugging. Often, the technical spaces are changed.

With our solution we provide (i) a temporal OCL extension based on the Computational Tree Logic (CTL) and (ii) an OCL extension that introduces path selectors to extract interesting system configurations from the state space. Both OCL extensions were formally defined and implemented. We describe systems in terms of state spaces consisting of EMOF–model states and state transitions containing a mapping between model elements of different states. The system behavior is specified using an initial *Ecore* model and graph transformations based on the Henshin tool[2]. The approach, however, is designed to be flexible enough to allow an easy integration of any kind of behavior specification as long as a suitable state space can be derived thereof. Our model checking framework is developed with a focus on delivering not only the results, but also making the system behavior leading to the result comprehensible by providing a suitable tool including a web interface.

The implementation was evaluated in terms of performance to find out the maximum evaluable model size and query complexity. Further, a qualitative user study was conducted for evaluating the CTL extension and the tool. The results of this study indicate that both the CTL–based extension of OCL as well as the tool are a promising first step to integrate model checking in the MDE life cycle.

[2] `https://www.eclipse.org/henshin/`

Kurzfassung

Das Ziel dieser Diplomarbeit ist, die Welten von Modellprüfung und Softwaremodellierung anzunähern. Während Modellprüfungstechniken inzwischen erfolgreich für die Verifikation von Hardware und Software eingesetzt werden, konnten sie in der modellgetriebenen Softwareentwicklung (MDE) noch nicht Fuß fassen. Eine Ursache dafür könnte sein, dass sich die Begrifflichkeiten in beiden Bereichen noch zu stark unterscheiden. Daher wird in dieser Arbeit versucht, Modellierungs– und Modellprüfungskonzepte zu verbinden. Zunächst werden Geschichte und grundlegende Konzepte von MDE und Modellprüfung mit einem Fokus auf die in dieser Arbeit benutzten Technologien beschrieben. Schließlich werden existierende Ansätze in diesem Bereich verglichen. Die meisten Ansätze schlagen vor, die Object Constraint Language (OCL) um temporale Aspekte zu erweitern. Denn OCL ist zwar gut für die Abfrage von Eigenschaften aus statischen Modellen geeignet, bietet aber kaum Möglichkeiten, ein sich im Zeitverlauf veränderndes Modell zu behandeln. Es gibt aber bislang keine Realisierung dieser Ansätze, bei der die Präsentation der Verifikationsausgabe hilfreich für das Debuggen des Modells ist.

Die in der Diplomarbeit erarbeitete Lösung bietet (i) eine OCL–Erweiterung um zeitliche Aspekte, die auf der Computational Tree Logic (CTL) basiert und (ii) eine OCL–Erweiterung, die Pfadselektoren zur Auswahl interessanter Systemkonfigurationen bereitstellt. Beide OCL–Erweiterungen wurden formal spezifiziert und implementiert. Wir beschreiben Systeme durch Zustandsräume, in denen Zustände aus *Ecore*–Modellen bestehen. Zustandsübergänge enthalten eine Abbildung zwischen Modellelementen eines Zustands und dessen Nachfolgezustands. Das Systemverhalten wird durch ein Anfangsmodell und Graphtransformationen in Henshin[3] beschrieben. Der Ansatz wurde entworfen, flexibel genug für andere Verhaltensspezifikationen zu sein, die einen geeigneten Zustandsraum liefern. Unser Modellprüfungsframework wurde entwickelt, um das Systemverhalten, das zu einem Zustand geführt hat, verständlich zu machen. Dazu wurde ein geeignetes Werkzeug mit einer Webschnittstelle entwickelt.

Die Implementierung wurde hinsichtlich Skalierbarkeit im Bezug auf Eingabemodellgröße und Ausdruckskomplexität überprüft. Ebenso wurde eine qualitative Benutzungsstudie durchgeführt, um die Benutzbarkeit der CTL–Erweiterung und der Webschnittstelle zu überprüfen. Die Ergebnisse zeigen, dass die OCL–Spracherweiterung auf Basis von CTL sowie das entwickelte interaktive Werkzeug einen vielversprechenden Schritt in Richtung Integration von Modellprüfung in den MDE-Lebenszyklus darstellt.

[3] `https://www.eclipse.org/henshin/`

Contents

CHAPTER 1

Introduction

While the idea that model checking could be helpful for debugging software models by providing examples of invalid software behavior has been around for more than a decade, there is still no tool providing a complete integration of model checking and model–driven engineering (MDE) [17]. The common term *model* occludes that the underlying modeling concepts are quite different. This chapter contains a short description of the interconnection between MDE and model checking, the issues occurring when integrating both areas, and the contributions provided in this thesis.

1.1 Problem statement

With software projects getting larger and more complex, there has been a rising demand for structuring such projects adequately. Software models have been identified as powerful structure and abstraction mechanism [30] reducing the gap between problem and implementation. First, models mainly served as design artifacts used in early project stages when traditional software engineering techniques are applied. The code was manually derived from these models. Later, in the context of model–driven engineering (MDE), software models became the core artifacts to specify and develop a system. They play an important role during the complete development process and often provide the basis for the automatic generation of executable code. Here, abstract software models are used as basis for describing different, more concrete, fine–grained artifacts. While these artifacts might be other models or code and are either (semi-)automatically generated or constructed by hand, errors in the abstract models might propagate to the more concrete models. Thus, the correctness even of the abstract models is a prerequisite for the correctness of the system that is presented to the end user [65].

Today, the correctness of software is ensured by various software quality assurance techniques. For this purpose, either incomplete techniques like testing or complete techniques like formal verification approaches are applied. In the area of formal verification,

the technique of *model checking* has been extremely successful over the last decades [12]. Several attempts have been made to use formal verification techniques in MDE, however, it has been recognized that the technological gap hinders the successful adoption.

Typically, a modeling environment provides some language to express constraints that a model has to satisfy. For example, the Object Constraint Language (OCL) [36], which is based on classical first–order logic, is a widely adopted language to express invariants and pre- and postconditions over a static model. While it is possible to check certain properties of behavioral models using OCL, OCL cannot check properties with respect to the model's evolution, e.g. during the execution of the system. Thus, it might be used to check properties of any single execution snapshot, but it is not possible to assert that an object fulfills a certain property during its whole lifetime or that a certain system state is never reached.

Properties which have to hold during the system execution, however, are typically properties to be verified by model checking. Model checking requires a formal representation of the system and a specification that is often expressed in terms of a temporal logic formula [4]. Common choices are the computation tree logic (CTL) [14] and the linear temporal logic (LTL) [58] that are used to express constraints over the lifetime of a system.

The integration of model checking into MDE faces three challenges. The first challenge is that model checkers need precise semantics which current modeling standards like UML lack [48]. The second challenge is to integrate these languages operating on primitive, hardware–oriented, values with modeling languages offering rich object–oriented concepts. Part of this challenge is the technological gap between model checking and MDE. The third challenge is to actually evaluate these properties in a high–performance way [31].

In the past, there have already been attempts to integrate model checking techniques into MDE processes to help detect and avoid errors in models (cf. [31] for a survey). Recent works and tools, for example Hugo [45] and Groove [43], show that various kinds of software models can be suitably verified with model checking. Many approaches translate the software model into the input format of an off–the–shelf model checker. This again tends to result in two main challenges. The correctness of the translation from the original model to the model checker model must be ensured as well as the model checker output must be translated back to give information about the original model. For achieving easier translation, many approaches sacrifice usability. First, the properties of a specification might be expressed in a language similar or even equal to the language of the target model checker which is different to the languages available in the modeling environment. In particular, the properties are not expressed on the modeling layer but on the (lower) model checking layer. Secondly, the back–translation of the model checker output might be omitted diminishing its value for debugging found errors. The aim of this thesis is to develop an approach which overcomes these issues.

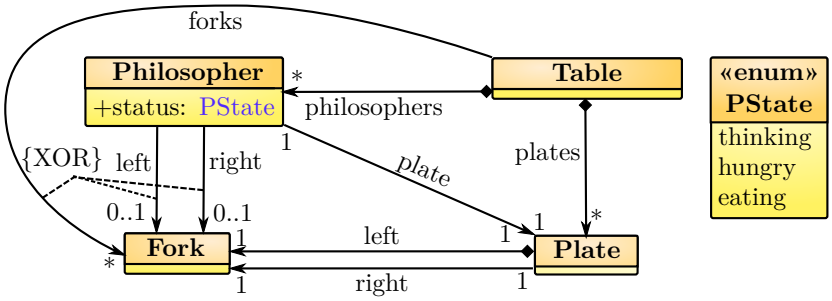


Figure 1: Dining philosophers model

1.2 Motivating Scenario

We motivate this work by an adaption of the widely known dining philosophers problem. The dining philosophers problem was initially described by Dijkstra [43] for showing issues of concurrent algorithms, in particular deadlocks and starvation. A deadlocked system cannot proceed as a whole while in starvation certain processes are never able to do work.

Figure 2 illustrates the initial scenario of the dining philosophers problem. There are `n` philosophers sitting around a table with a plate in front of them. Initially, there are two forks beside each plate, which are shared between neighbors. The actions of each philosopher consist of repeated sequences of thinking, being hungry, picking up the left fork, picking up the right fork and eating. A philosopher is thus not able to pick up the left fork if his/her left neighbor has already picked up his/her right fork and is not able to pick up the right fork if his/her right neighbor has already picked up his/her left fork.

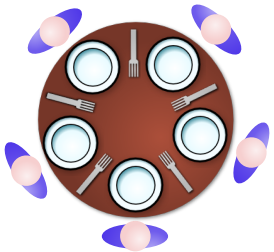


Figure 2: Problem setting of dining philosophers

The model shown in Figure 1 represents the dining philosophers problem in terms of a UML class diagram.[1] The model contains one table, an arbitrary number of philosophers, plates and forks. The behavior of the model could be expressed for example by means of state diagrams or model transformations. For the moment, an informal description of the behavior is sufficient. Later in this thesis we will use graph transformations. Philosophers initially have no fork in their hands and are thinking. In arbitrary order, they may expose the following behavior: A thinking philosopher may get hungry. A hungry philosopher may take the left fork if it is free. A hungry philosopher having a left fork may take the right fork if it is free and starts eating. An eating philosopher may put back both forks onto the table and start thinking again.

In Figure 3, an instance of the dining philosophers problem with three philosophers is shown and the lifecycle of philosopher P_1 is demonstrated. Starting with the initial

[1] Note, due to technical restrictions of the Henshin tool [59], each non–root object must have a container. For forks, this is the plate object.

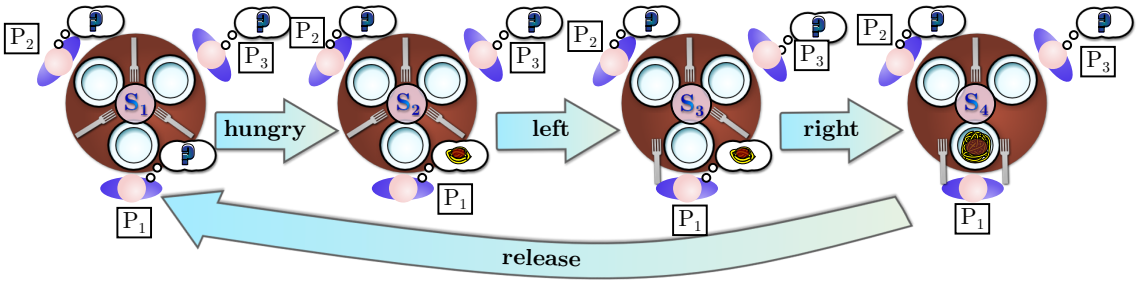


Figure 3: Philosopher P_1's lifecycle

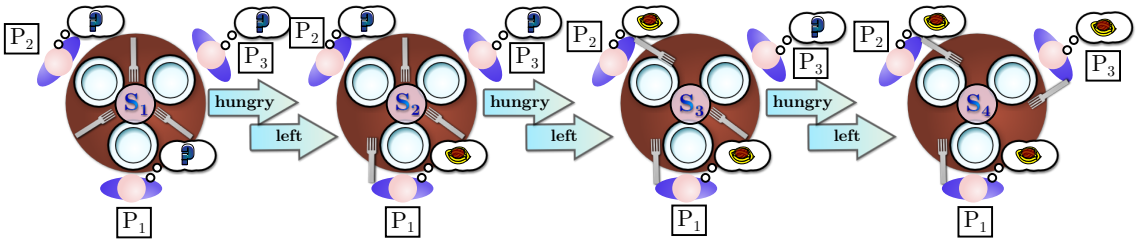


Figure 4: Deadlock scenario

configuration in state S_1 where all philosophers are thinking, philosopher P_1 is hungry in state S_2, has a left fork in S_3 and is eating with an additional right fork in state S_4. As soon as S_3 is reached, P_2 cannot finish eating until P_1 has finished eating. In state S_4, P_3 cannot even start taking a fork. Thus, this cycle already presents a simple case of possible starvation of P_2 and P_3. If P_1 always immediately manages to regrab the forks after putting them on the table, the other philosophers will never be able to eat something even though P_2 might pick the left fork. Still, P_1 may decide at any time to stop being too greedy and let the others eat something. This is different in the deadlock situation depicted in Figure 4. All philosophers got hungry at about the same time and picked up the left fork. Now the only action for every philosopher is to pick up the right fork, but this is possible for no one. Thus, the system has stopped working.

The main question is: how can we find out such problematic behavior? The idea of this thesis is to use semantically well–defined model transformations together with a CTL extension of OCL and a custom extension of OCL.

Note that the dining philosophers problem will guide us as running example through this thesis.

1.3 Outline

The three main contributions of this thesis are (1) a formal semantics of a CTL–based extension and a custom selector extension of OCL, (2) a prototypical implementation thereof as extension of the Eclipse OCL Engine[2] and a custom web interface displaying

[2] `http://wiki.eclipse.org/MDT/OCL/Plugins_and_Features`

evaluation results, and (3) an evaluation of this approach. This thesis is structured as follows.

In Chapter 2, we discuss some foundations of model–driven engineering (MDE). A brief historical tour of MDE is given illustrating challenges and opportunities in that area. In the same way system structures are described by models, models themselves adhere to a certain structure, the metamodel. The relation between both of them is explained in the next section. Additional properties of models might be expressed using the Object Constraint Language, whose syntax and semantics is explained then. OCL serves as basis of this thesis. A short overview of different approaches to model transformation is presented. Then, we give an overview of graph transformations usable for model transformations.

Chapter 3 discusses basic concepts of model checking. At first, an overview of general model checking concepts is given, then the computational tree logic (CTL) is explained in detail as it plays an important role in the rest of this thesis.

Related work is discussed in Chapter 4. Different tools and OCL extensions are described and compared to give an overview of existing approaches related to this work.

Chapter 5 presents the own approach. In particular, we present MocOCL, an extension of OCL with temporal operators. Formal semantics are given for both the CTL extension and the custom selector extension. The implementation of both extensions is discussed and an overview of the web interface is given.

Chapter 6 introduces the evaluation of the approach. A usability study of the CTL extension and the web interface and their results are discussed. Performance test results give an overview of the scalability of the presented approach.

Chapter 7 concludes this thesis with an outlook to future work.

CHAPTER 2

Model–Driven Engineering at a Glance

Since the early days of computer science, effort has been put in specifying more what to do instead of how to do. For example, in 1969 Dijkstra put up the question *"What can we do to shorten the conceptual gap between the static program text (spread out in "text space") and the corresponding computations (evolving in time)"* and proposed to deal with program composition. Programs should first be described in an abstract way, then iteratively refined by making decisions [18]. Abstraction was continuously raised by the development of machine code, simple programming languages, structured programming languages, and object oriented programming. Model–driven engineering (MDE) is a continuation of these efforts to let the developer not have to specify every single detail of the system, but instead model the required functionality and overall architecture. MDE automates routine programming tasks like system persistence and interoperability, and allows the developer to focus on creative and non–trivial tasks [3, 63].

2.1 Model–Driven Engineering

Until the late 1960s, software projects were often not conduced in a systematic manner; instead, just after the requirements were specified informally the programming began [54]. This resulted in code which is hard to maintain and a huge gap between the expectations of the user and the final program. For improving methodical approaches, structured programming was employed allowing a standardized and disciplined way of programming code. Ideas of that time included to explore interfaces where typed code skeletons should help finding components which are as independent from each other as possible [54]. This allowed easier maintenance, debugging and modularization, but did not help much improving fitting the program to the user's needs [11]. Thus, techniques were introduced for defining requirements of the system to be built, for example data flow

diagrams. While these techniques were successful in the sense that the resulting system was closer to the system specified and as well more easily maintainable, projects took longer. Thus, these techniques increased the development quality but not the productivity because of increased effort of specifying requirements which was not supported well by tools [11]. Computer–aided software engineering (CASE) in the 1980s was supposed to help programmers develop software faster. CASE tools allow an automated delivery and execution of software engineering methods, procedures and tools with assistance for all software engineering aspects and contain a set of highly integrated tools. They contain a user–friendly interface to all tools in the system including development tools like editors, for specifications as well as programs, management tools, a help subsystem and a database management system [11]. CASE tools gained remarkable success in research, but were not wildly used in practice.

While CASE tools already allowed to generate executable systems directly from visual models, manual customization and debugging was thwarted by the difficult transformation caused by the poor mapping between the employed general purpose modeling language and the target platform [63]. Additionally, team work was not supported properly because only one person could work on a system at a time. The lack of middleware made it hard to integrate CASE tool generated code with other software. Thus, in fact even in the cases where CASE tools were used, they were plainly used for drawing diagrams as specification for manual implementations without direct relation to the diagrams [63].

Program languages themselves have also continuously gained more abstraction mechanisms to ease inclusion of other software including middleware, for example for fault tolerance and security. Still, the platform complexity has risen faster then the language advancement could handle resulting in some effort required for considering small side effects and dependencies of the used frameworks and with code having to be updated for new platform versions [63]. Additionally, activities like system development, configuration and quality assurance is harder using these programming language concepts and still hard when using common XML–based deployment descriptor notations resulting from a high discrepancy in design intent specified in a few lines and the implementation requiring hundreds of lines [63].

Mode–driven engineering [35] nowadays is a promising approach addressing this complexity. It allows the specification of domain–specific modeling languages (DSML) to close the gap between the modeled system and the target platform. DSMLs formalize application structure, behavior and requirements and provide generators analyzing models and generating various artifacts including XML deployment descriptors. The automated generation ensures consistency between implementation and specification [63]. This has the potential to increase developers' productivity. Software might change during its lifetime, for example by adding new features or changing portions of the architecture. If models are not synchronized with the implementation, they get useless. Models need to outlive their personal creator which might leave the company and thus must be described using a concise and tailorable notation to be understood by all involved people. Changing requirements should have low impact on the system as a whole, especially

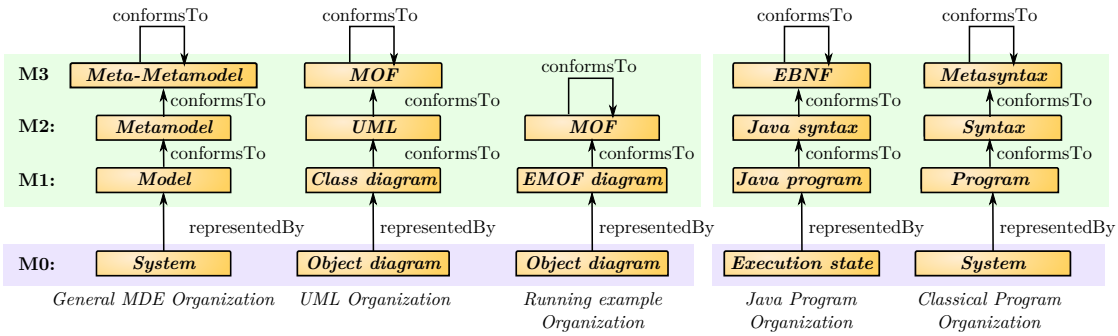


Figure 5: Metamodels and Metasyntax, adapted from [6]

at the running phase. Thus, new types must be addable at runtime. To be able to change the development platform, they need to support high interoperability levels for the software artifacts. Also the deployment platform could change; thus the retrieval of platform specific artifacts from platform independent ones should be as automatic as possible. Still, user–defined tailorization should always be possible [3].

The domain–specific modeling languages of MDE differ from the models used in CASE tools in the sense that they are less general but can be tailored to the specific needs of the domain, not only in terms of syntax and semantics, but also in terms of visual representation [63]. Since errors get more expensive late in the project cycle, MDE tools check domain–specific constraints and provide verification tools to find problems as early in the life cycle as possible [63].

2.2 Models and Metamodels

While domain–specific modeling languages are necessary in order to increase productivity as explained in the section above, the need for general solutions also requires the domain–specific modeling languages being described in a standardized way. Thus, the Object Management Group (OMG)[1] developed a standard for the modeling of modeling languages called Meta–Object Facility (MOF) [37]. MOF was designed to allow defining and extending metamodels and metadata models as easy as extending usual object models. It should be reusable and platform independent. Complete MOF (CMOF) is designed to be used as metamodel to specify other metamodels. Essential MOF (EMOF) is a subset of CMOF allowing to express features found in object–oriented programming languages. Both EMOF and CMOF are completely specified by themselves [37] as specification language.

Figure 5 shows the layered architecture proposed by the OMG. Each abstraction step results in a dedicated layer and is described by a modeling artifact. The lowest layer M0 describes the real system. The model layer M1 describes the elements of this real system. The metamodel layer M2 describes the elements of the model. The meta–

[1] http://www.omg.org

metamodel layer M3 describes the elements of the metamodel. For example, a Unified Modeling Language (UML) [38] object diagram at layer M0 might correspond to real objects in the system. The structure of this system might be described using an UML class diagram at layer M1. The class diagram conforms to UML which itself is specified in MOF. Principally, the number of layers is not fixed for a language specified using MOF. MOF allows any amount of layers larger than one. In the case of two layers, MOF is used as class diagram to specify object diagrams. The dining philosophers example used in this thesis is specified in MOF. A certain situation in the dining philosophers problem is specified using an instance of this model, the layer M0. The dining philosophers model itself is thus in layer M1. MOF is used as metamodel of the dining philosophers model, thus in layer M2. Since MOF is specified in itself, there are no more different layers and thus in total, there are three different layers for the running example of this thesis.

The basic concept of having multiple layers of representation is not new to MDE. In fact, a similar structure is exposed in classical programming. The metalanguage used to specify programming languages is the Extended Backus–Naur Form (EBNF) [29] which itself is defined in EBNF. The syntax definition of a programming language like Java can be written in EBNF. The program itself then is written in this programming language. When the program executes, its state can be represented as being in a certain point in the program with specific variable assignments including the stack elements.

Ecore

Ecore is an implementation close to the MOF standard and is commonly referred to as reference implementation of MOF. *Ecore* is the main metamodel language used in the Eclipse Modeling Framework (EMF) [10] which unifies Java, XML and UML by providing a common high–level representation.

Since *Ecore* is the basis for not only the running example, but also the metamodel language for defining models or metamodels used in the approach of this thesis, a description of *Ecore* follows. Figure 6 shows the general types used in *Ecore*. An *Element* is used as common superclass for all metaclasses in EMOF. *Comment*s allow attaching remarks to elements useful for modelers. Their *body* attribute, a string, defines the actual comment. The *annotatedElement* reference specifies the Elements which are commented. A *NamedElement* is an element with a name used for identification of the element within the namespace, i.e. the container it is defined in. Its *visibility* may be `public`(+), `private`(-), `protected`(#) or `package`(~). *Public* elements are visible to all elements which can access the owning namespace. *Private* elements are visible only within their namespace. *Protected* elements are visible to elements having a generalization relationship to the owning namespace. Elements with visibility *package* are visible to elements within the nearest enclosing package. A *package* contains types specified by the *ownedType* association and packages specified by the *nestedPackage* association. It may have an URI string as universally unique identification. The named element *Type* is used as type for a *TypedElement*. A *DataType* is an abstract class acting as supertype for instances identified only by their value. These instances have type *PrimitiveType* if

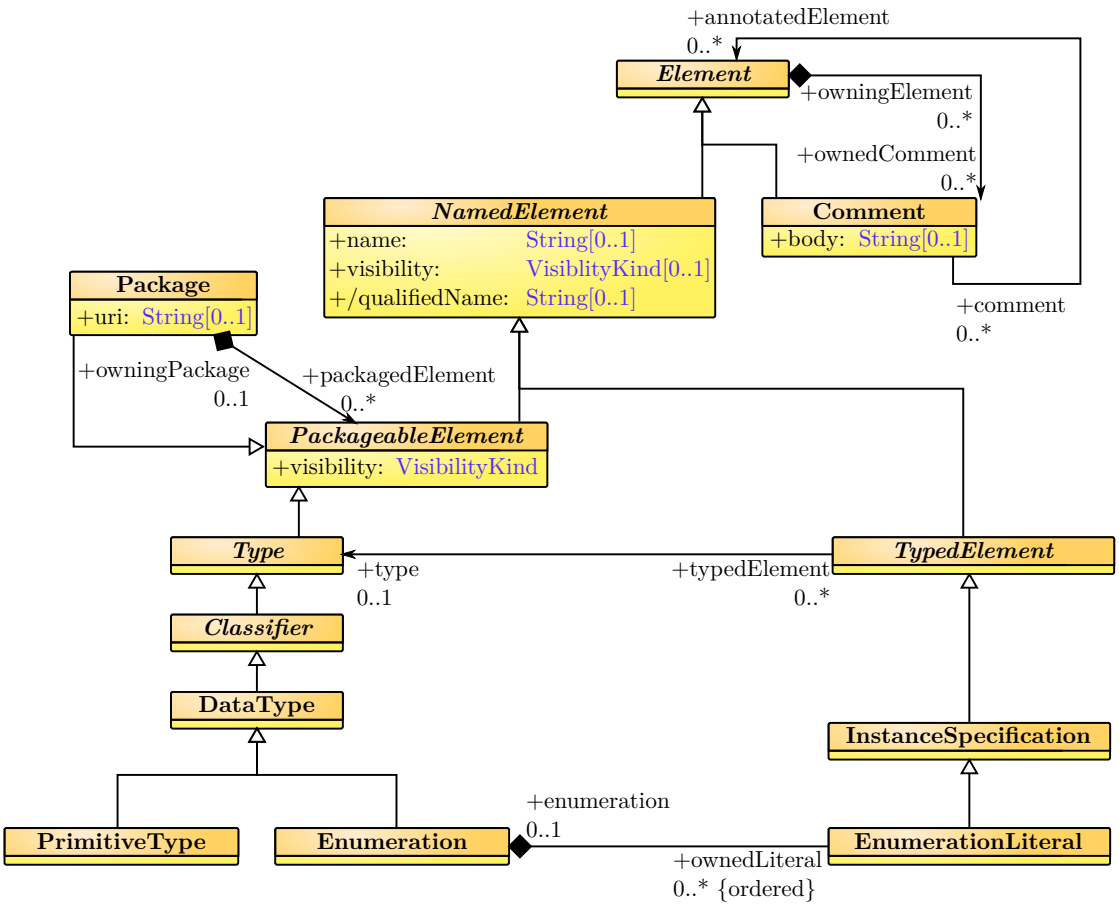


Figure 6: General overview of types and packages in EMOF, adopted from [37]

they are Integers, Booleans, Strings or UnlimitedNaturals or *Enumeration* if their value is contained in a finite set of values specified as *EnumerationLiteral*.

Figure 7 shows the basic classes contained in EMOF. Objects of type *Class* may have instances. Classes have a type and an arbitrary number of superclasses. They inherit from *Classifier* and thus can be abstract. Abstract classes cannot have direct instances, but every direct instance of a class is an indirect instance of all its superclasses. The superclasses attribute in class redefines the general attribute of the classifier. Classes have slots for their direct and inherited attributes. Objects allow the invocation of their class' operations and the operations of the superclasses with the context of the invoked object. *Operations* have an ordered set of parameters and exceptions they can raise. *Parameters* specify arguments for passing information in the context of operation invocations. The *direction* attribute defines whether the information is passed into or out of the operation with possible values `in`, `inout`, `out` and `return`. Parameters inherit from *MultiplicityElements*, which define an inclusive interval of non–negative integers with lower and (possibly infinite) upper bound. MultiplicityElements might also be ordered

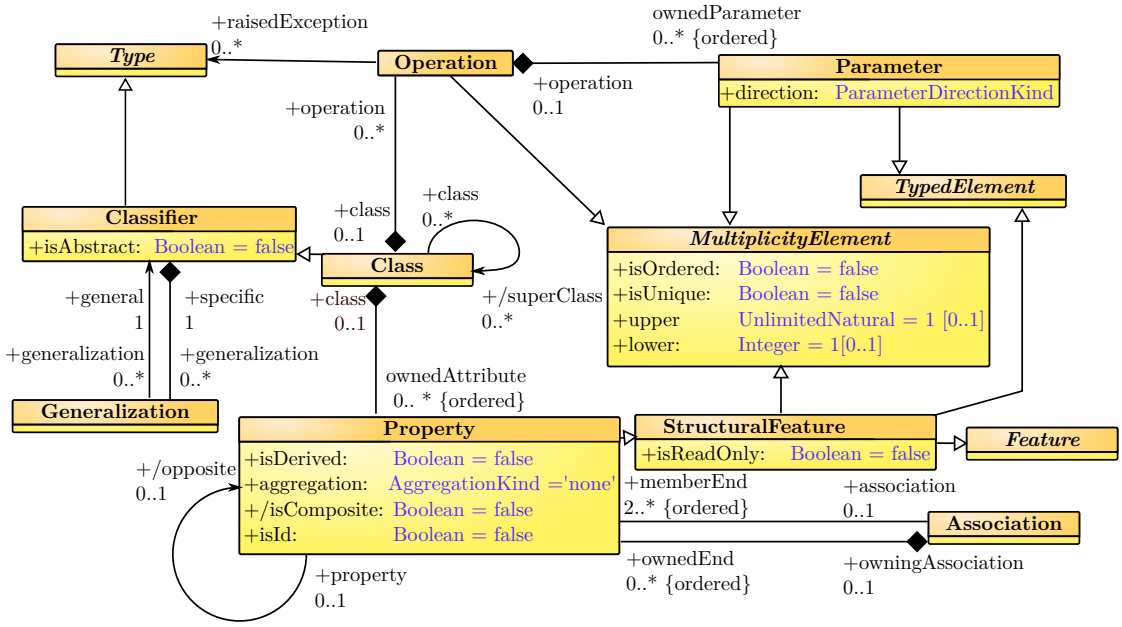


Figure 7: General overview of classes included in EMOF, adapted from [37]

or unique.

Properties are representing attributes of a class. *Composite* properties are contained in the object and have an aggregation of 'composite', else they have an aggregation of 'none'. *Derived* attributes are calculated from information elsewhere. For any property p_1 of object o_1 which is *opposite* to another property p_2 of object o_2, it holds that o_1.p1 refers to o_2 iff o_2.p2 refers to o_1, so they are bidirectionally navigable. A property which is an ID uniquely identifies an instance of the containing class. Properties are *StructuralFeature*s and thus can be defined to be read only and multiplicities can be defined like for operation parameters. *Association* instances are links with one value for each end of the association. The property of the end of the association is navigable. *Member ends* refer to instances connected by the classifier. *Owned ends* are owned by the association itself and are a subset of member ends.

Figure 8 shows an EMOF model of the dining philosophers problem setting. In many cases, attribute values can be omitted because they are equal to the default values. The actually used *Ecore* model of the dining philosophers problem looks similar with the main difference that there is a distinction between attributes (the status property) and references (all other properties). Note that there are different possible *representations* for models. For example, the model depicted in Figure 1 is the same as in Figure 8. The *concrete syntax* specifying the surface representation is different in these cases. The *abstract syntax* specifying the real, internal, structure stays the same. Figure 8 shows a more or less trivial concrete syntax allowing easy reasoning on the abstract syntax while Figure 1 shows a concrete syntax making the model easier to read but possibly forcing the reader to think about the abstract syntax of the model a bit.

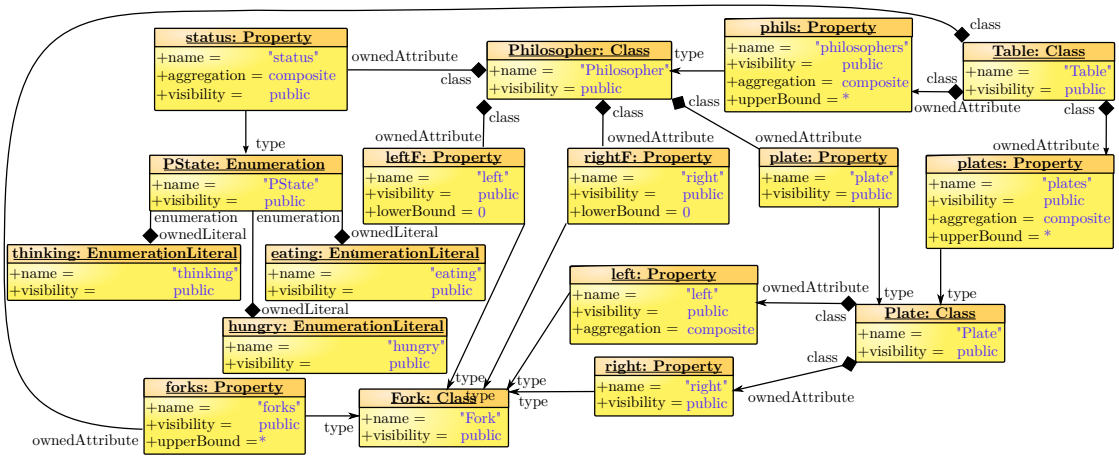


Figure 8: EMOF model of the dining philosophers problem as depicted in Figure 1

Beside having the syntax, a clearly defined semantics is important as well, especially for software model verification. All the constructs used in the model must have clearly defined semantics and there may be no semantically relevant information for the model outside the scope of defined semantics, e.g. in comments. A commonly known early compromise between expressibility and checkability in the past were petri nets [57]. Today, UML is more often used in modeling. Since the UML semantics are partly specified in an informal way, struggles have been made to define subsets of UML with formally defined semantics, for example Foundational UML (fUML) [39]. The behavioral semantics of our approach are based on graph transformations and discussed in detail in Chapter 5 .

Object Constraint Language

When models and metamodels have constraints wich cannot be formulated in the context–free (meta–)meta–modeling language alone, additional constraint specification languages are required. For example, the Object Constraint Language (OCL) [36] supplements MOF and provides a textual language for specifying expressions on UML and other MOF based models which enhances the metamodel's constraint specification facilities with first–order logic. OCL thus combines the advantage of being easier to be read by a modeler than typical formal languages constraining models as the modeler does not have to leave his/her technical space but still being formal enough to be evaluated automatically. OCL is a specification language and thus side–effect–free [36].

In the following, we describe the semantics of *Essential OCL*, the minimal OCL subset required to work with EMOF. Additionally, some selected parts of *Complete OCL*, the full OCL language, which are relevant for this thesis, are explained as well. A formal semantics for Essential OCL is listed in Definition 30 where we enhance OCL with temporal operators.

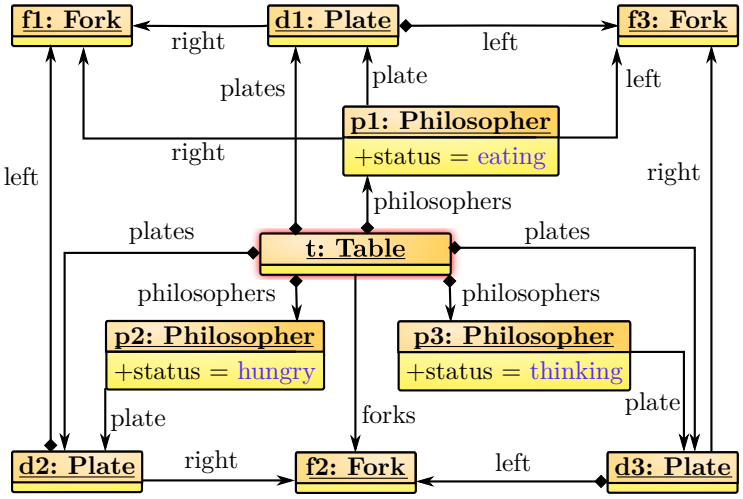


Figure 9: Object diagram of three dining philosophers where one is eating, one is hungry and one is thinking

Context Complete OCL provides the *context* for specifying the scope of an OCL expression. The *invariants* are general constraints on objects which hold forever. For operations, *pre–* and *postconditions* can be specified as well as their return (or *body*) value. For attributes, constraints can be specified that must hold *initially* only or *all the time*. In the following, these concepts will be discussed in detail.

Definition 1 (Context)**.** A *context* of an expression specifies where and how the expression is used in the (meta)model. The special expression *self* refers to the contextual instance on which the OCL expression is evaluated. In complete OCL, the context can be specified for types and operations and any other kind of behavioral feature.

`context` $(n$`:)?`t `inv:` e

This construct specifies an *invariant*, that is, the boolean expression e must evaluate to true for all instances of the type t. The `self` expression refers to the specific instance of the specified type on which the expression is evaluated. A different name for `self` may be defined using n. `self` may be omitted if the context is clear.

`context` t`::`n `(`pn_1`:`pt_1`,...):`t_r `pre:` e_{pre} `post:` e_{post}

This construct specifies pre– and postconditions of a certain operation. Whenever the precondition holds prior to the operation invocation, the postcondition must hold after the operation invocation. In a postcondition, e_p`@pre` can be used to refer to the result of the expression e_p at the beginning of the operation call. All further properties accessed give the new values. The *self* expression refers to object on which the operation was called. The special variable `result` is used to represent the return value. □

Figure 9 shows the model used for the following examples. It depicts a situation occurring in the dining philosophers problem. There are three philosophers sitting around a

table with one philosopher thinking and one philosopher hungry having no forks and one philosopher eating having two forks. The fork which does not belong to any philosopher belongs to the table.

The context of every example expression is, unless otherwise noted, the highlighted table object t. For every expression we assume that variables `f1`, `f2`, `f3` are defined for each of the objects $f1, f2, p1, ...$ in the diagram to allow easier representation of return values. In the examples and definitions, purple text in italics denotes *types*, blue text denotes constants, brown text objects and object access, except for variables in black text. Keywords are bold.

Example 1 (Context). The following expressions illustrate different application scenarios of `context`.

```
context p:Philosopher inv:  p.status <> null
```

The philosopher status must always be set.

```
context Philosopher::getHungry()
pre:  self.status = PState::thinking and left = null and right = null
post:  self.status = PState::hungry and left = null and right = null
```

When the operation `getHungry` is called, the philosopher must be `thinking` and have no fork in his/her hands. After the operation call, the philosopher must be `hungry` and still has no fork in his/her hands.

```
... post:  self.status <> self.status@pre
```

The status after the operation call should be different from the status before the operation call.

```
... post:  self.status <> self @pre.status
```

The status of the philosopher referred to by `self` at the end of the operation call is different from the status at the end of the operation call of the philosopher referred to by `self` at the beginning of the operation call. This implies that the philosopher referred to by `self` has changed.

```
context Philosopher::guessStatus() post:  if self.right <> null then result
= PState::eating else if self.left <> null then result = PState::hungry else
result <> PState::eating endif endif
```

A guess of the status from the forks owned will result in `eating` if there are two forks in the philosopher's hands and `hungry` if there is one fork in his/her hands. If the philosopher has no fork, he/she does surely not eat.

For query operations returning a value, a *body* expression might be used to return the result. The initial value of an attribute or association end can be specified with *init* and *derive* is used as derivation constraint, who must hold at all time [36].

Example 2 (Body). The following expression uses the `body` attribute to define the return value of the philosopher operation `isHungry()`.

```
context Philosopher::isHungry() body:  self.status = PState::hungry
```

The function `isHungry` returns `true` iff the philosopher's status is hungry.

Definition 2 (Init, Derive)**.**

```
context t::a_n:a_t init/derive:  e
```

This specifies the value of an attribute or association end named a_n with type a_t of the class t. Its initial value is the result of the evaluation of e after *init*, the *derive* constraint m must hold all the time. □

Example 3 (Init, Derive). The following examples illustrate the use of the `init` and `derive` keywords to specify the value of philosopher attributes at certain points in time.

```
context Philosopher::left:Fork init:  null
```

The left fork of a philosopher is initially `null`.

```
context Philosopher::right:Fork derive:
if self.status = PState::eating then plate.right else null endif
```

The philosopher always has a right Fork in his/her hands if his/her status is eating.

In Essential OCL, there is no `context` definition. Thus, the context or rather the value of the self attribute has to be provided to any Essential OCL evaluation engine.

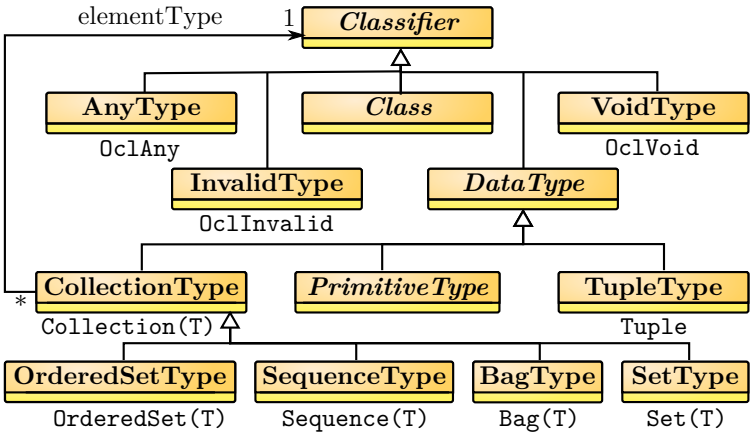


Figure 10: Partial OCL type hierarchy, adapted from [36]

Types There are basically two kinds of types in OCL: Basic types, which are predefined in OCL, and types of the metamodel the OCL expressions are attached to. The type is usually specified using a colon. For example, $n : Integer$ refers to a variable n with

type Integer. Figure 10 shows a part of the OCL type hierarchy. As in MOF, all types inherit from the common supertype *classifier*. A type `A` conforms to a type `B` if `A` can be used anywhere where `B` can be used.

There are three special OCL types: *AnyType*, written as `OclAny` (i.e. AnyType is the *metaclass* of OclAny) is a type to which all other types conform. *InvalidType*, written as `OclInvalid`, represents invalid values and does conform to all types except `OclVoid`. Its only instance is `invalid`. *VoidType*, written as `OclVoid`, is used for specifying the absence of any value and conforms to all types except `OclInvalid`. Its only instance is `null`.

The *primitive types* of OCL are UnlimitedNatural, Integer, Real, String and Boolean. *Real* represents a real number (e.g. 3.14). *Integer* represents an integer and conforms to Real. *UnlimitedNatural* represents a natural number including `*` denoting infinity. It conforms to Integer except for `*` which is replaced by `invalid` in expressions. *Boolean* represents the common truth values `true` and `false`. *String* represents a character sequence, e.g. `'hungry'`.

Each collection type is parameterized by the type of elements in the collection. Thus, there are potentially infinitely collection types. Collections may contain elements of any type including collections. The subtypes of collection are *Set* containing elements only once, *OrderedSet* having an order defined and containing elements only once, *Bag* allowing duplicates and *Sequence*, having an order defined and allowing duplicates. Collections literals are defined using curly brackets, for example `Sequence {1,2,3}` defines the sequence containing the numbers 1, 2 and 3. This could also be written als `Sequence {1..3}`. Collections conform to each other if their elements obey the same type or a super type.

The typical struct construct is represented using *TupleValues*. It contains name/value–pairs declared like a variable with potentially distinct types.

Example 4 (Tuples). The following example shows a tuple definition.

```
Tuple {status:String = 'hungry', numForks:Integer = 1}
```

This tuple contains an attribute `status` of type `String` with value `'hungry'` and an attribute `numForks` of type `Integer` with value `1`.

Users can define custom *enumeration types* including a name and a set of literals. They define a finite variable domain. Enumeration literals are accessed using `<EnumType>::<EnumLiteral>`.

Example 5 (Enums). The following example shows the basic use of enums.

```
Enum PState {thinking, hungry, eating}
```

This constructor creates an Enum containing all possible philosopher's status values. The elements can be accessed using `PState::thinking`, `PState::hungry` and `PState::eating`.

Since the exact dynamic OCL type is sometimes not known for an OCL object, there are methods for type checking and type conversation.

Definition 3 (Type Related Methods)**.** The following methods are defined on all instances of any type.

`obj.oclIsTypeOf(t:`*Classifier*`):`*Boolean*`/obj.oclIsKindOf(t:`*Classifier*`):`*Boolean*

These methods check if `obj` has exactly type `t` or rather type `t` or a subtype.

`obj.oclAsType(t:`*Classifier*`):`*t*

Change the known type of the instance `obj` to the type `t`. This may as well be used to access overridden properties of supertypes. If casting is not possible, i.e. the type of `obj` does not conform to `t`, `invalid` is returned.

`obj.oclIsUndefined():`*Boolean*

Returns true iff `obj` is `null` or `invalid`

t`.allInstances():`*Collection(t)*

Gives a collection of all instances of the specific type.

The only operations defined on enumeration types are the test of equality and the operation `allInstances()`. □

For the following examples, we use the notion of `<Expression>` $\overset{\text{ret}}{\Longrightarrow}$ `<Value>` to denote that the evaluation result of `<Expression>` is `<Value>`.

Example 6 (Type Related Methods). The following examples illustrate certain kinds of type relationships. The expressions are evaluated on the highlighted table object of Figure 9.

`self.oclIsTypeOf(`*Table*`)` $\overset{\text{ret}}{\Longrightarrow}$ `true`

`self.oclIsTypeOf(`*OclAny*`)` $\overset{\text{ret}}{\Longrightarrow}$ `false`

`self.oclIsKindOf(`*OclAny*`)` $\overset{\text{ret}}{\Longrightarrow}$ `true`

`self.oclAsType(`*Fork*`)` $\overset{\text{ret}}{\Longrightarrow}$ `invalid`

`self.oclAsType(`*OclAny*`)` $\overset{\text{ret}}{\Longrightarrow}$ `self`

It is not possible to access Table–specific properties any longer.

`self.oclAsType(`*OclAny*`.oclAsType(`*Table*`))` $\overset{\text{ret}}{\Longrightarrow}$ `self`

It is again possible to access Table–specific properties

`p2.left.oclIsUndefined()` $\overset{\text{ret}}{\Longrightarrow}$ `true`

Fork`.allInstances()` $\overset{\text{ret}}{\Longrightarrow}$ `Set{f1,f2,f3}`

Navigation Attributes and association ends can be retrieved using a dot separating object and attribute or association end name. The returned value is the value of the corresponding attribute; if the multiplicity is greater than one, a set is returned. The

dot operator can be chained. If the dot operator is applied to a collection, its result is the flattened collection

Example 7 (Navigation). The following examples illustrate examples of navigation in objects.

```
self.forks ⟹ret Set{f2}

self.philosophers ⟹ret Set{p1,p2,p3}

self.philosophers.status ⟹ret Set{PState::eating, PState::hungry,
PState::thinking}
```

Operations defined on OCL objects can be accessed using a dot followed by the operation name and the parameters in parenthesis. If an operation is called on a collection, it is applied to all elements of the collection.

OCL has a number of predefined operations including simple mathematical operations like basic arithmetical and rounding operations as well as comparison methods including maximum and minimum operations.

Definition 4 (Operation on Numeric Primitive Types)**.**

`a +/-/*///</>/<=/<>/>=/=/div/mod b`

The numeric operations have their usual meaning with the only exception that `/` of two integers or UnlimitedNatural values instances returns a real value. `div` is the integer division. All operations can be written either in infix notation or using the operation notation `a._'+'(b)`.

The expressions `max`/`min` Return the greater/smaller values of both numbers.

There are some additional operations on real values. The expression `round` rounds half up to integers, i.e. if the fractional part is 0.5, the greater value is chosen. The expression `floor` returns the greatest integer value not larger than the given value. Doing any mathematical operation with the `UnlimitedNatural *` results, like dividing by `0`, to `invalid`. □

Example 8 (Numerical Functions). The following examples show the use of typical numerical functions in OCL.

```
1.max(1.5) ⟹ret 1.5; (-5).min(3) ⟹ret -5; (-1.1).min(-0.9) ⟹ret -1.1

(2.5).round() ⟹ret 3; (2.9).floor() ⟹ret 2; (-2.1).floor() ⟹ret -3

Sequence{1,3,-1,-5}.div(3) ⟹ret Sequence{0,1,0,-1}
```

There are also predefined operations on Strings in OCL. These include concatenation and query operations for length and characters or substrings at specific positions, but also operations for searching substrings and basic string conversation operations. Strings can also be compared lexicographically and parsed into Integers and Booleans.

Definition 5 (Operations on Strings)**.**

`a.concat(b:`*String*`):`*String* and `a + b`

These operations concatenate two strings.
The operation `a.size():`*Integer* returns the number of characters in a string.
The operation `a.at(i:`*Integer*`):`*String* returns the i–th character of the string (starting with 1).

`a.substring(lower:`*Integer*`,upper:`*Integer*`):`*String*

This operation returns the sub string starting at character `lower` up to, including, `upper`.
The operation `a.indexOf(s:`*String*`):`*Integer* returns the index of the first occurrence of `s` in `a` or `0` if there is no such index.
The operation `a.characters():`*Sequence(String)* converts a string into a sequence of characters.

`a.toLowerCase():`*String* and `a.toUpperCase():`*String*

These operations convert the string to lower– and upper case according to the set `oclLocale`.

`a.equalsIgnoreCase(s:`*String*`):`*Boolean*

This operation checks if `a` and `s` are equal if both are converted to upper case.

`a < / > / >= / <= b`

These operations compare the string lexicographically considering `oclLocale`.
The operation `a.toBoolean():`*Boolean* converts a string to a boolean which is `true` if the string is `'true'`, else `false`.
The operations `a.toInteger():`*Integer* and `a.toReal()` convert a string to an Integer and a Real, respectively. If the string cannot be represented as such type, `invalid` is returned. □

Example 9 (Operations on Strings). The following examples show the use of typical string functions in OCL.

```
concat('The philosopher is ','hungry') ⟹(ret) 'the philosopher is hungry'
'10' <= '9' ⟹(ret) true
```

The following operations are all evaluated on the string `'hungry'`:

```
self.size() ⟹(ret) 6; self < 'eating' ⟹(ret) false; self.at(2) ⟹(ret) 'u'
self.substring(2,4) ⟹(ret) 'ung'; self.indexOf('ung') ⟹(ret) 2
self.toUpperCase() ⟹(ret) 'HUNGRY'; self.characters() ⟹(ret)
Sequence{'h','u','n','g','r','y'}
'TRUE'.toBoolean() ⟹(ret) false; '3.4'.toInteger() ⟹(ret) invalid; '3.5'.toReal() ⟹(ret)
3.5
```

The basic operations on booleans are also supported as well in OCL.

Definition 6 (Operations on Booleans)**.**
a `and/or/xor/not/implies` *b*

These operations have their usual meaning. □

Example 10 (Operations on Booleans). The following examples show the use of boolean operators in OCL.

`true xor true` $\overset{\text{ret}}{\Longrightarrow}$ `false; false implies true` $\overset{\text{ret}}{\Longrightarrow}$ `true; true implies false` $\overset{\text{ret}}{\Longrightarrow}$ `false`

There are also various operations on collections predefined in OCL. The simplest operations allow to check whether some objects are in a collection or not, how many objects are in a collection and some simple aggregate collection functions. Collection operations are accessed using the arrow syntax `col->op()` instead of the dot syntax. For every collection, we assume the collection contains elements of type `T`.

Definition 7 (Operations on Collections I)**.** The expression `col->size():`*`Integer`* returns the size of the collection.

`col->includes(obj:`*`T`*`):`*`Boolean`*`, col->excludes(obj:`*`T`*`):`*`Boolean`*

These expressions determine whether `obj` is contained for `includes` or rather not contained for `excludes` in `col`.

`col->includesAll(c2:`*`Collection(T)`*`):`*`Boolean`*

This expression determines if all objects in `c2` are included in `col`.

`col->excludesAll(c2:`*`Collection(T)`*`):`*`Boolean`*

This expression determines if no object in `c2` is included in `col`.
The expression `col->count(obj:`*`T`*`):`*`Integer`* returns the number of `obj`s in the collection. The expression `col->isEmpty():`*`Boolean`* (`col->notEmpty():`*`Boolean`*) returns `true` (`false`) if the collection is empty.
For numeric elements, the expressions `col->max():`*`T`* and `col->min():`*`T`* returns the biggest or rather smallest element of the collection. `col->sum():`*`T`* returns the sum of all elements in the collection. □

Example 11 (Operations on Collections I). The following examples illustrate the use of operations providing information about collections in OCL.

`self.philosophers->size()` $\overset{\text{ret}}{\Longrightarrow}$ `3; self.forks->includes(f1)` $\overset{\text{ret}}{\Longrightarrow}$ `false`

Table only has an association to the fork `f2`.

`self.philosophers->includesAll(`*`Philosopher`*`.allInstances())` $\overset{\text{ret}}{\Longrightarrow}$ `true`

The table in fact contains all philosophers.

`self.forks->excludesAll(self.philosophers.left)` $\overset{\text{ret}}{\Longrightarrow}$ `true`

The table contains in fact only forks not held by philosophers, whereof `self.philosophers.left` is a subset.

`Set{1,2,2}->count(2)` $\overset{\text{ret}}{\Longrightarrow}$ `1`

2 is only contained once in each set.

`Sequence{1,2,2}->count(2)` $\overset{\text{ret}}{\Longrightarrow}$ `2`; `Set{1,2,2}->max()` $\overset{\text{ret}}{\Longrightarrow}$ `2`

`Set{1,2,2}->sum()` $\overset{\text{ret}}{\Longrightarrow}$ `3`; `Sequence{1,2,2}->sum()` $\overset{\text{ret}}{\Longrightarrow}$ `5`

As well as types can be converted, the type of Collections can be also converted to other collection types.

Definition 8 (Operations on Collections II)**.**
`col->asSet():`*`Set(T)`*`, col->asSequence():`*`Sequence(T)`*`, col->asBag():`*`Bag(T)`*`, col->asOrderedSet():`*`OrderedSet(T)`*
These expressions convert the collection to the specified type. The `c1 = c2` (`c1 <> c2`) operator returns `true` (`false`) iff both sets contain the same elements equally often and, if the collections are ordered, in the same order. □

Example 12 (Operations on Collections II). The following examples illustrate the use of collection type conversion operations in OCL.

`Bag{1,2,3}->asSet() = Bag{1,2}->asSet()` $\overset{\text{ret}}{\Longrightarrow}$ `true` `Bag{1,2} = Set{1,2}` $\overset{\text{ret}}{\Longrightarrow}$ `false`

`Set{1,2,2} = Set{1,2}` $\overset{\text{ret}}{\Longrightarrow}$ `true`; `Bag{1,2,2} = Bag{1,2}` $\overset{\text{ret}}{\Longrightarrow}$ `false`

After multiple processing steps, collections might be deeply nested. In these cases, it is sometimes useful to get a single collection containing all the values stored in the most deeply nested collections. This is provided by the *flatten* operation.

Definition 9 (Operations on Collections III)**.**
The operation `col->flatten():`*`Collection(T2)`* returns the recursively flattened out collection. This is, it returns the same collection if the contained elements are not collections, else the collection containing all elements $e \in c$ of the flattened elements $c \in col$ of the collection `col`. The operation `col->product(c2:`*`Collection(T2)`*`):` *`Set(Tuple(first:T,second:T))`* returns the cartesian product of both collections. □

Example 13 (Operations on Collections III). The following examples illustrate the use of collection transformation operations in OCL.

`Set{Set{1,2,3},Set{2,3,4}}->flatten()` $\overset{\text{ret}}{\Longrightarrow}$ `Set{1,2,3,4}`

`Sequence{Set{1,2,3},Set{2,3,4}}->flatten()` $\overset{\text{ret}}{\Longrightarrow}$ `Sequence{1,2,3,2,3,4}`

`Sequence{1,2}->product(Sequence{2,3})` $\overset{\text{ret}}{\Longrightarrow}$ `Set{Tuple{first: 1, second: 2}, Tuple{first: 1, second: 3}, Tuple{first: 2, second: 2}, Tuple{first: 2, second: 3}}`

The classical set operations are also implemented in OCL including *union*, *intersection*, set *difference* and *symmetric difference*.

Definition 10 (Operations on Collections IV)**.** The operations `col->union(col2)` and `col->intersection(col2)` are defined for both `Set` and `Bag` as each of their operands. The operation `union` returns a `Bag` retaining all multiplicities of both collections if at least one collection is a `Bag`, else a `Set`. The operation `intersection` returns a `Bag` containing the minimum amount of both collections if they are `Bags`, else a `Set`. The operation `col->including(obj:`*`T`*`):`*`Set(T)`* returns a `Set` with the specified element added. `col->excluding(obj:`*`T`*`):`*`Bag(T)`* returns a `Bag` with all occurrences of the specified element removed.

The operation `col->symmetricDifference(s:`*`Set(T)`*`)` defined on sets return a set of all elements included in exactly one set and `col - (s:`*`Set(T)`*`)` returns all elements contained in the first, not in the second set. □

Example 14 (Operations on collections IV). The following examples illustrate the use of set operations in OCL.

`Set{1,2,3}->union(Bag{1,4})` $\overset{\text{ret}}{\Longrightarrow}$ `Bag{1,2,3,1,4}`

`Bag{1,2,2}->intersection(Set{2,2})` $\overset{\text{ret}}{\Longrightarrow}$ `Set{2}`

`Bag{1,2,2}->intersection(Bag{2,2})` $\overset{\text{ret}}{\Longrightarrow}$ `Bag{2,2}`

`Set{1,2}->including(3)` $\overset{\text{ret}}{\Longrightarrow}$ `Set{1,2,3}`

`Bag{1,2,2}->excluding(2)` $\overset{\text{ret}}{\Longrightarrow}$ `Bag{1}`

`Set{1,2}->symmetricDifference(Set{2,3})` $\overset{\text{ret}}{\Longrightarrow}$ `Set{1,3}`

`Set{1,2,3}-Set{3,4}` $\overset{\text{ret}}{\Longrightarrow}$ `Set{1,2}`

Simple operations on sequences are also predefined in OCL. Elements can be added to the beginning, to the end or somewhere inbetween. The first and last elements of a collection can be retrieved and the position of an Element can be figured out. Additionally, sequences can be reversed.

Definition 11 (Operations on collections V)**.** Operations defined for `OrderedSets` and `Sequences` are the following: `col->append(obj:`*`T`*`):`*`Collection(T)`* adds `obj` as last element to the collection. The operation `col->append` is equivalent to `col->including`. The expression `col->prepend(obj:`*`T`*`)` adds `obj` as first element to the collection. The expressions `col->first():`*`T`* and `col->last():`*`T`* retrieving the first or rather last element of the collection. The expression `col->insertAt(index:`*`Integer`*`, obj:`*`T`*`)` inserts `obj` at the specified position and moves the other elements backwards. The expression `col->at(i:`*`Integer`*`):`*`T`* retrieves the element at the specified index while `col->indexOf(obj:`*`T`*`):`*`Integer`* returns the position of `obj` and `invalid` if `obj` is not found. The expression `col->reverse()` reverses the order of the collection. □

Example 15 (Operations on Collections V). The following examples illustrate the use of element access and sequence modification operations in OCL.

`Sequence{1,2}->append(5)` $\overset{\text{ret}}{\Longrightarrow}$ `Sequence{1,2,5}`

```
Sequence{1,2}->prepend(5) ⟹ret Sequence{5,1,2}

OrderedSet{3,2,1}->first() ⟹ret 3

Sequence{1,8,9}->insertAt(2,5) ⟹ret Sequence{1,5,8,9}

OrderdSet{1,2,3}->indexOf(5) ⟹ret invalid; Sequence{4,5,6}->indexOf(5) ⟹ret 2

Sequence{1,2,3}->reverse() ⟹ret Sequence{3,2,1}
```

For both sequences and ordered sets, subcollections can be determined by providing indices or values, respecively, of first and last element.

Definition 12 (Operations on Collections VI)**.**

```
col.subOrderedSet(lower:Integer, upper:Integer):OrderedSet(T)

col.subSequence(lower:Integer, upper:Integer):Sequence(T)
```

These expressions give the subsequence starting with the `lower` element up to, including, the `upper` element. □

Example 16 (Operations on Collections VI). The following examples illustrate the use of subsequence access operations in OCL.

```
Sequence{3,4,5,6,7}->subSequence(2,4) ⟹ret Sequence{4,5,6}

OrderedSet{1,1,2,3,4,5,6}->subOrderedSet(2,4) ⟹ret OrderedSet{2,3,4}
```

OCL also provides collection operations called *iterator operations*. They provide more flexible information retrieval for sets. It is possible to check whether every, at least one or exactly one element of a set meets a specific condition, to select elements in a set matching a condition and to recursively select elements. It is also possible to define custom iteration functions in a limited way.

Iterator operations have the form `col->operation(iterVar | <boolean expression in iterVar>)` and are evaluated as follows: For each element in the collection, the *iterator* `iterVar` gets assigned to this element and the boolean expression on the right–hand side of the `|` symbol is evaluated. The exact accumulation of these expressions is depending on the actual iteration operation. The iteration variable definition `iterVar` may be omitted in case the context object equals to the collection object. Iterators may be typed using `iterVar: iterType`. Single objects behave as sets containing only the relevant object. The iterator operations `forAll` and `exists` support multiple iterators, where each combination of values is tried.

Operation	Description
`forAll`	Returns true iff for all elements, the boolean expression evaluates to true.
`exists`	Returns true iff for at least one element, the boolean expression evaluates to true.
`one`	Returns true iff for exactly one element, the boolean expression evaluates to true.
`select`	Returns a collection containing all elements of the collection with the boolean expression evaluating to true.
`reject`	Returns a collection containing all elements of the collection with the boolean expression evaluating to false.
`any`	Returns an object with the boolean expression evaluating to true. If none exists, `invalid` is returned.

Table 1: Table of important collection iteration operations

Example 17 (Collection Operations VII). The following examples illustrate the use of iteration operations in OCL.

```
self.philosophers->forAll(p | p.status <> PState::thinking implies
p.left <> null) ⟹ret false
```

The implication that if a philosopher is not thinking he/she has a left fork is checked for all philosophers. The philosopher `p2` is hungry but has no fork. Thus, false is returned.

```
self.philosophers->forAll(p,q | p.status <> q.status implies p.left <> q.left)
⟹ret false
```

The implication that if two philosophers have different status values, exactly one of them has a fork in his/her left hand is checked. The philosopher `p2` is hungry and `p1` thinking, so their status values are different, but both have no fork.

```
self.philosophers->select(p | p.status <> PState::thinking
implies p.left <> null) ⟹ret Set{p1,p3}
```

The selection of all philosophers which have a left fork if they are not thinking. This is the philosopher `p1` who has a left fork, and `p3` who is not thinking.

```
self.philosophers->any(p | p.left = null) ⟹ret p2 or ⟹ret p3
```

The selection of any philosopher who has no fork. Both `p2` and `p3` have no fork and it is not guaranteed which one is given.

```
self.philosophers->one(p | p.status = PState::thinking) ⟹ret true
```

Whether there is exactly one philosopher thinking. This is true since just the philosopher `p3` is thinking.

There are three important collection operations with slightly different syntax.

Definition 13 (Iteration collection operations)**.**

`col->collect(iterVar:`*Type* `| <expression in iterVar>)`

This expression calculates the specified expression in iterVar for all elements of the collection. A collection containing all these evaluation results is returned.

`col->closure(iterVar:`*Type* `| <expression in iterVar>)`

This expression calculates the transitive closure of the specified expression which is the union of the collection and all elements returned by any evaluation of the expression on an element of this union.

`col->iterate(iterVar:`*Type*`, acc:`*Type* `= <init expression> | <acc expression in iterVar and acc>)`

This expression provides generic means of describing collection operations like the ones mentioned above. Initially, `acc` is assigned the value of `<init expression>`. Then, for each element, `acc` is assigned the value of the `<acc-expression>` with `iterVar` containing an element of the collection. If the collection does not contain any more elements, the current value of `acc` is returned. □

OCL provides two kinds of control structures: `If` and `let`. The control structure `if` evaluates different expressions depending on a conditions, the structure `let` allows to define variables with specific value which can be used evaluating an OCl expression.

Definition 14 (If Expression)**.**

`if` e_c **`then`** e_t **`else`** e_e **`endif`**

This expression returns the value of e_t if e_c evaluates to `true`, else the value of e_e. □

Example 18 (If Expression). The following example illustrates the use of `if` to find out whether there are forks picked by philosophers.

`if self.forks->size() < Fork.allInstances()->size() then 'sometaken' else 'nonetaken' endif` $\overset{\text{ret}}{\Longrightarrow}$ `'sometaken'`

There is only one fork associated to the table, but there are three fork instances in the model altogether. Thus, the string `'sometaken'` is returned.

Definition 15 (Let Expression)**.** The expression **`let`** n**`:`** t **`=`** e **`in`** e_n declares a variable with the name n of type t with the value of e. n must not be already used as variable name. The result of this expression is the result of e_n. □

Example 19 (Let Expression). The following examples illustrate the use of the `let` expression to reduce the expression length.

```
let usedforks = self.philosophers.left->union(self.philosophers.right)
in self.plates->select(p | usedforks->includes(p.left) or
usedforks->includes(p.right)) ⟹ret Set{d1,d2,d3}
```

This expression selects the plates which have forks taken by philosophers. All plates are returned because `d1` has `f1` and `f2` used by `p1`, `d2` has `f1` used by `p1` and `d3` has `f3` used

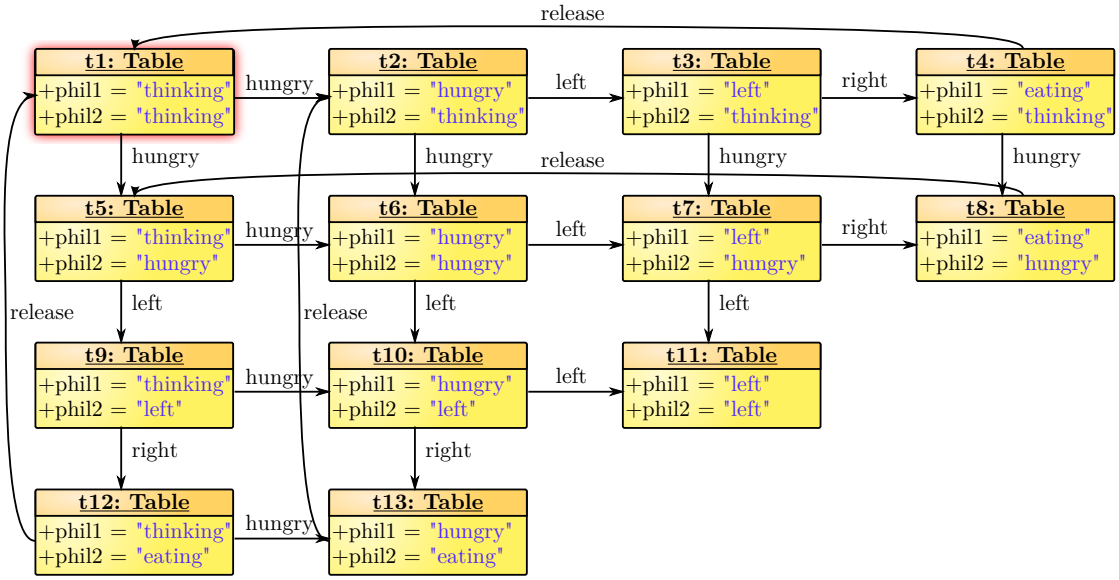


Figure 11: Possible configurations for two dining philosophers

by `p1`.

For the following examples, we will use the model depicted in Figure 11. The model contains several table elements with all possible configurations of two dining philosophers. The references between the table elements reflect the modification from one configuration to another. We start with both philosophers thinking, which then can get hungry, have a left fork in their hands and then take the right fork and start eating. Each table consists just of two attributes describing the different philosopher states. In order to distinguish between the state `'hungry'` with no and one fork, the state hungry with one fork is called `'left'`.

Example 20 (Collection Operations VIII). The following examples illustrate the use of advanced iteration operations in OCL.

```
Table.allInstances()->collect(t:Table | t.phil1.status + '/' + t.phil2.
status) ⟹ret Collection{'thinking/thinking','hungry/thinking','hungry/left',...}
```

This expression calculates string representations of states and includes every existing combination of two philosopher statuses.

```
self->closure(t | t.hungry) ⟹ret Set{t1,t2,t5,t6}
```

This expression collects all tables reached directly or indirectly via the `hungry` reference. In the first step, only `t1` is in the set. Then, `t1.hungry` is evaluated and `t2` and `t5` are added to the result set. For each of the new elements, `t.hungry` is evaluated again and thus `t6` would be added two times, but since the closure operation does not add elements twice to avoid some infinite calculations, `t6` is only once in the result set.

```
Table.allInstances()->iterate(t:Table, acc: Integer = 0 | acc + if
t.phil1.status = 'eating' then 1 else 0 endif + if t.phil2.status = 'eating'
then 1 else 0 endif) ⟹ret 4
```

This expression calculates the total number of `eating` occurrences in the configurations.

The following example uses the concepts introduced above for reasoning over the possible philosopher configurations. It serves as basic witness that the combination of model checking and modeling is possible. While most if not all existing approaches of combining model checking with OCL enhance the OCL language using special constructs, this example shows that using a suitable model structure, Essential OCL itself can be used for model checking. To be more precise, this example formulates the property `A phil1 <> 'eating' and phil2 <> 'eating' U phil1 = 'eating' or phil2 = 'eating'`, i.e. that in every case, there will be a philosopher eating and before this happens the first time, no philosopher will be eating. See Section 3.4 for more details on model checking. Note that this approach has several serious drawbacks: (1) Essential OCL might not be the best choice for writing a model checker because due to the nature of OCL, it is not possible to interrupt loops. Thus, it is not possible to finish the evaluation as soon as the result is known. (2) All philosopher configurations have to be known when starting the evaluation process. This might be suboptimal for queries requiring only a few configurations. (3) In order to trace model elements, many additional model elements would have to be added to the model and it might be cumbersome to actually use these elements for tracing objects in different states.

Example 21 (Complex Example). We want to find out whether we will surely find a philosopher eating by traversing tables where no philosopher is eating starting from the `self` table.

```
let phi = self->closure(t | t.hungry->union(t.left)->union(t.right)
->union(t.release)->select(t.phil1 <> 'eating' and t.phil2 <> 'eating') ⟹ret
Set{t1,t2,t3,t5,t6,t7,t9,t10,t11}
```

This expression gives us all tables reachable from the initial table from the closure of the union of all association ends which continuously have no philosopher eating. Initially, only `t1` is in the closure. The associated tables are `t2` and `t5` both having no philosopher eating. Then we continue with `t2` and find `t3` and `t6` being addable to the set. This is continued until all tables except the ones with philosophers eating are selected.

Now we can select border tables `phineighbor` which are tables where there is a philosopher eating not in the set phi.

```
let phineighbor = phi->collect(t | t.hungry->union(t.left)->union(t.right)->
union(t.release)->select(t.phil = 'eating' or t.phil2 = 'eating'))->flatten()
in let psi = phineighbor->reject(t | phi->includes(t)) ⟹ret Set{t4,t8,t12,t13}
```

This gives us a set `phi` where we first collect all associated tables of `phi` which fulfill the eating–condition, then we flatten this collection of collections and then we exclude all elements in `phi`. The result of phineighbor is every table except for `t1` because this is the only table not reachable by a state in `phi`. If we exclude all states `phi`, we get the

four border states t4, t8, t12 and t13.

```
psi->union(phi)->includesAll(psineighbor) ⟹ret true
```

If we want to be sure to find a philosopher eating, then, there may be no associated table of psi neither in psi nor phi. This returns true because psi->union(phi) covers in fact all tables.

```
phi->forAll(t | t->closure(t.hungry->union(t.left)->union(t.right)->
union(t.release)->intersection(phi))->excludes(t)) ⟹ret true
```

There must not be a possibility that we traverse only table associations in phi without ever getting to psi. This can be expressed as that there may be no navigation possibility from any table in phi through tables in phi to itself. This is the case because there is no cycle within our phi set.

```
phi->forAll(t | t.hungry->union(t.left)->union(t.right)->union(t.release)
->notEmpty())⟹ret false
```

We have to be able to actually get to some neighbor from each table in phi. This is not the case because we cannot reach any table from t11. Thus, we will not surely find a philosopher eating by traversing tables with no philosopher eating.

OCL offers various features and functions for evaluating properties of static models. It allows easy navigation and collection handling. However, the only support for dynamic model properties are pre– and postconditions which do not allow to reason over longer model behavior. While dynamic models can be converted into static models, the access of properties occurring in specific situations gets difficult. OCL also is not turing complete [50] so maybe some properties used in model checking cannot be expressed at all.

2.3 Model Transformations

Model transformations rewrite a source model into a target model. Both Czarnecki and Helsen [16] and Mens and Gorp [51] provide an overview of model transformation. Basically, a transformation engine reads a source model conforming to a source metamodel, executes the transformation definition referring to source and target metamodel and writes the target model conforming to the target metamodel. They can be classified in various ways, for example, whether they are *endogenous*, which means source and target models are in the same language or *exogenous* where the source model is rewritten into a model of a different language. If source and target models are the same, the changes are called *in–place*, else, they are called *out–place*. Note that this can only be the case for endogenous transformations. *Horizontal transformations* have source and target models which are on the same abstraction level while *vertical transformations* have them at different abstraction layers. *Syntactical transformations* only change the form, while *semantical transformations* also change the meaning of a model. Other important characteristics include whether manual intervention is needed, how complex the transformations are, which aspects of the model are preserved during the transformation

and whether the transformation is uni– or bidirectional, i.e. whether the transformation can be executed in only one direction or in both directions.

There are *declarative* or *imperative* approaches to model transformation. Declarative approaches describe what should be done to transform the model. They are useful because an underlying reasoning engine can be used for source model navigation and target model creation. Further, the control flow of the rule applications has not be given explicitly and thus the transformations are easier to write and understand. If these declarative approaches, however, are not sufficient, imperative approaches are used. They are required especially when transformation application order needs to be controlled explicitly or to adjust the model after being changed by the transformation tool. In any case, a precise metamodel is required for automated model transformation .

Typical ingredients of model transformations include *transformation rules*, the smallest units of transformations. Typically these are rewrite rules with left–hand side (LHS) and right–and side (RHS), but also functions can be used as transformation rules. *Rule application conditions* controls the order of transformations as well as the model elements which are transformed.

In this thesis, we need model transformations for defining the behavior of the system to be verified. In particular, we use graph transformations because of their well defined semantics.

Graph Transformations

In the following, the core concepts and usage of graph transformations as one approach of model transformation are discussed. We give an informal introduction to graph transformations like given by Heckel [41], as it is sufficient to get a basic understanding. However, a more formal approach regarding the execution of graph transformations in an algebraic manner [22] can be found in the Appendix A.

Graphs are defined in rather different ways in literature. For simplicity, we regard graphs here as valued nodes connected by labeled edges. Each edge has a source and target node. EMOF models are considered as graphs here by regarding attributes as associations to their values. Figure 12 shows an example. Graphs may also be typed. Each node and edge of a graph is connected to their type node or edge in a type graph. The type graph of a graph corresponds to the metamodel of a model. For example, the type graph of the graph in Figure 12 could look similar to the dining philosophers model in Figure 1.

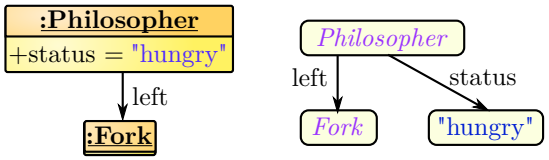
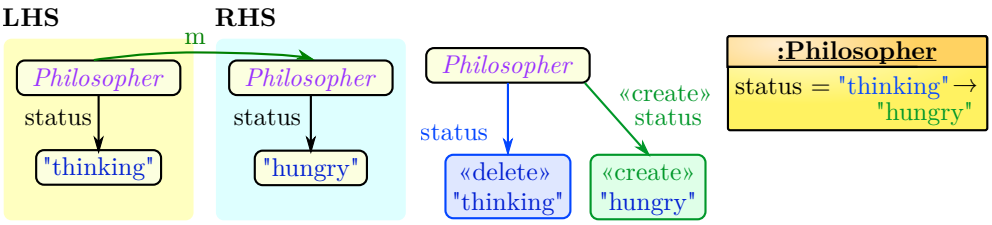


Figure 12: EMOF model and corresponding graph

Graph transformation rules may abstract the behavior of a system modeled by graphs by concentrating on the relevant subgraphs for changes [41]. Typical graph transformation rules consist of a *left–hand side* (LHS) and a *right–hand side* (RHS) where the graph of the left hand side is converted to the graph of the right hand side. A graph trans-

formation application consists of two phases: First, a matching between the LHS of a graph transformation rule and the graph must be established. Then, the rule must be applied.



(a) Classical notation (b) Storyboard notation I (c) Storyboard II

Figure 13: Graph transformation described in different ways

There are multiple notations of graph transformations for the visual representation. A classical notation displays LHS and RHS of a graph and the mapping function assigning at most one element of the RHS to every element of the LHS. The storyboard notation described in [27] and [55], on the other hand, shows only a single graph where graph changes are described using symbols for deletion, creation and update. The storyboard notation is more expressive then the simple, classical notation. Objects may be displayed weakly to show that the match is optional. An object may be displayed as staple to indicate that a set of multiple objects may be matched to this object. Changes are applied to every object in this set. There also exists a notion of negative objects, i.e. objects where no mapping is allowed. Figure 13 shows both the classical notion of a graph transformation rule in Figure 13a together with a variation of the storyboard notation in Figure 13b and another variant of the storyboard notation used for the transformation in this thesis in Figure 13c. In this rule, a philosopher's status changes from `thinking` to `hungry`. Thus, the LHS consists of a node depicting a Philosopher with status attribute being thinking, the RHS depicts the same philosopher having a status attribute which is hungry. The graph transformation written in storyboard notation in the middle shows the different syntax for the creation and deletion of objects. Objects and transitions are created using `«create»` in green and deleted using `«delete»` in blue. The storyboard notation allows an even more concise representation when used together with models instead of simple attributed graphs. Here, an attribute value change is represented using an →.

A graph transformation is performed in two steps. At first, an occurrence of the LHS has to be found in the host graph, i.e. the graph the rule is applied to. A match describes the relation between LHS and LHS occurrence in the host graph. In the most simple case, such a match connects each element of the LHS to a corresponding element of the host graph. This correspondence can be established in multiple ways of complexity. The simplest way is to look if the LHS graph occurs as part of the whole graph. More sophisticated methods include elements from the storyboard notation with one element allowing multiple (or no matches) and forbidding the occurrence of certain

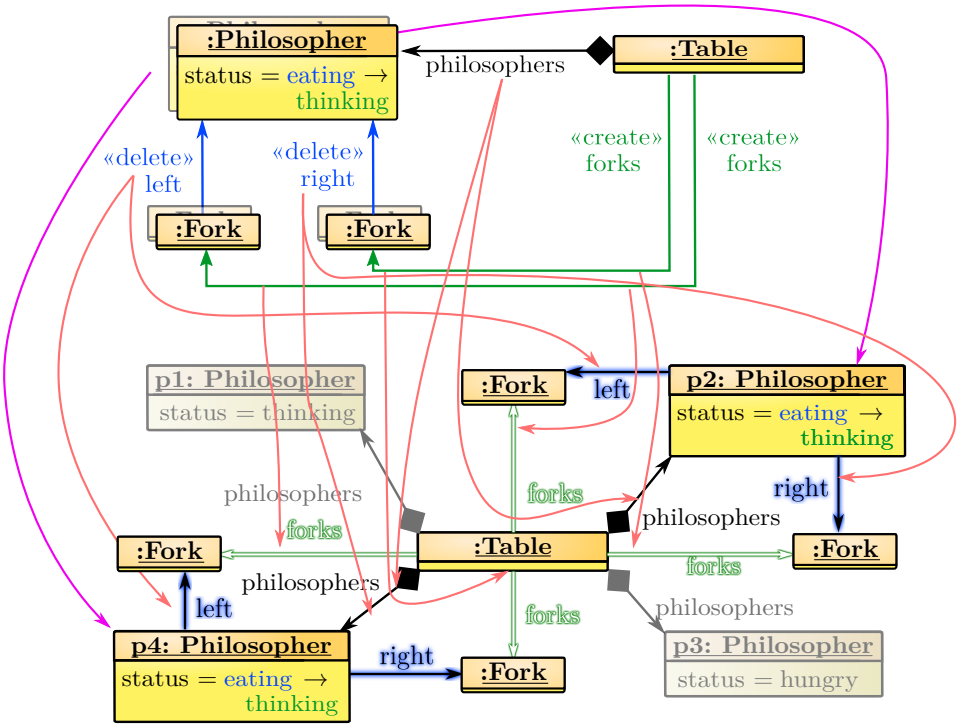


Figure 14: Mapping example of the *finish eating* rule. Purple arcs connect objects, flesh–colored arcs connect transitions

elements. In some approaches, it can be specified that node values of different nodes have to be equal or being in a certain relation. GROOVE [60] supports even more advanced concepts like specifying that two nodes are connected not by a single edge, but by a certain regular expression of edges. In fact, there seem to be nearly no limits in the specification of matching conditions. A common, general differentiation of matchings is between *injective* and *non–injective* matchings. Injective matchings require that at most one element of the LHS is mapped to an element of the graph. This allows, on the one hand, easier algorithms as seen in Section A and on the other hand, it also matches the intuition that different elements in the LHS are actually different elements in the graph.

After the mapping has been found, the graph transformation has to be executed in the second step. Elements occurring only in the LHS, but not in the RHS have to be deleted. Elements occurring only in RHS, but not in the LHS are created. While this seems trivial, there are some corner cases which need special handling, e.g. when deleting a node and adding an edge to the node at the same time. In these cases, deletion typically takes precedence. Figure 14 shows an example of a mapping for the `finish eating` rule. The upper part of the Figure shows the transformation rule, a part of the host graph is shown in the bottom and the mapping is defined by the connecting arcs.

The finish eating rule makes all philosophers finish eating. In the graph, two philosophers have `eating` as their status value. Since the rule covers multiple philosophers, both

philosophers are matched. The other two philosophers have different status values and thus are not considered. The rule deletes the connections between both philosophers and their forks and adds connections between these forks and the table.

In case multiple rules exist for transforming a system, the control flow of rule applications should be specified in some way. One way of specifying control flow are *application conditions* defining that a rule may only be applied if the application condition is fulfilled. Habel et al. [40] introduce *negative application conditions* (NACs). NACs forbid the existence of certain nodes and edges. Figure 15 shows an example for such a NAC. The rule shows, that a philosopher may only take the left fork if the philosopher left to him/her, characterized by this left fork being the left philosopher's right fork does not hold any fork. A NAC is only fulfilled if the whole graph specified by the NAC is matched. The rule may still be executed if only a part of the NAC matches. In fact, the same rule could be specified with all of the NAC except of the fork being in the LHS instead of the NAC, but then, a left philosopher would have to be matched or rather exist.

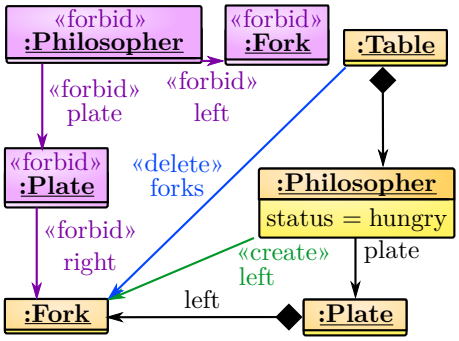


Figure 15: Rule for taking the left fork safely

A second approach is to describe the control flow directly by *programmed graph transformations*. Graph transformations might be applied in a specific order, a certain number of times or with other restrictions. Larger graph transformations can be created by using multiple other graph transformations in loops, conditions or sequences [41]. Of course, the effort for determining whether such a large rule is applicable might be considerable higher than for a single rule because all application conditions for any rule in the graph have to be fulfilled. For example, a rule `direct eating` could be constructed by concatenating the `left`, `right` and `release` rule. This rule can only be applied if both the right fork of the left philosopher and the left fork of the right philosopher are available. This application condition is stronger than the one of the `left` rule where only the left fork has to be available.

2.4 Summary

MDE started with general purpose modeling tools which were not suited well for the problems developers had and thus were merely used for drawing models, the actual implementation was done in the classical way. DSMLs have narrower purpose and thus can be tailored to the problems of the domain. These languages can be described using metamodeling languages like MOF. MOF and EBNF are similar in the sense that both are top level languages to describe systems and mainly differ in how the system descriptions are done. MOF models consist of classes having typed attributes and labeled associations with defined multiplicity. Advanced properties on MOF metamodels are specified using OCL, a query language with finite–domain first–order logic seman-

tics. OCL properties can be declared on attributes and operations. While OCL is not turing–complete it could be used for model checking purposes on suitable models, but requires the model to be completely available. The dynamic behavior of software can be described using model transformations. Models specified as graphs can be transformed using graph transformations. Common graph transformation rules describe the part of the graph being transformed. At first, a suitable part of the graph is searched, then other rule application conditions are evaluated and then the part is transformed.

CHAPTER 3

Model Checking

Model Checking is one of the most successful techniques in software and hardware verification [20] because it allows fully automated proofing that certain properties in a system hold. Thus, it is especially successful in areas where bugs impose great risks or are hard to fix. However, in MDE it has not found such a widespread adoption yet. In this thesis, a goal is to increase the usability of model checking in the area of MDE. To this end, model checking is introduced in detail. At first, the general idea behind model checking is explained, then a short note on the historical development of model checking is given. Furthermore, CTL is introduced which is the language used in the approach of this thesis. Other important languages are briefly described.

3.1 Software Verification

One major goal of software engineering is to deliver software that matches the user's needs. One way to achieve this goal and to fulfill functional an non–functional requirements is the application of testing and formal verification technologies. Adrion et al. [1] and D'Silva et al. [20] give an overview over software verification techniques.

The usual method for ensuring the quality of software is software testing. Programs are tested by executing the program with multiple test inputs and checking if the actual result matches the expected one. Various techniques explained by Zeller [68] are used in practice to increase software reliability. Therefore, testing values have to be chosen carefully. Typical test data includes valid and invalid inputs with a focus on data requiring special handling by the tool. Test automation reduces the costs for a single test; thus it allows to test more input values and many different versions of programs. Often, it is important to reason about the quality of the testing used so far. With respect to this, code coverage metrics give a hint about what parts of the program have been tested. For example, the *path coverage* is the fraction of the number of possible branching paths considering the program flow, but many different coverage metrics have

been defined. If the location of found bugs is stored, this information can be used to derive locations of higher bug density. Another estimation of the number of errors in the program can be given by artificially introducing errors in the code after testing with some test data and counting the percentage of found errors. Under the assumption that artificially introduced and accidental errors are equally findable, this percentage should be about the same.

Tests are not only used for showing that a program contains defects to uncover details, but also for debugging. Thus, tests not only need to provide information about whether a program is correct, but also support for finding the location of bugs. Debuggers execute the code step–by–step, allowing to follow each program statement. The origin of an offending value can be traced back by looking at what was assigned to the value and what conditions made this assignments execute [68].

Unfortunately, testing has a major problem. It can only show the presence of bugs, not their absence [18]. A complete verification process, however requires that exhaustive testing is used, i.e., the test process is executed for every possible program environment and every possible input. Unfortunately, even for rather simple, finite, domains, this is often infeasible [1]. Another approach, where the code is not run but is analyzed itself, is *static analysis* [1]. Typically, the concern is to be able to guarantee certain properties are fulfilled. It is acceptable that in some cases, properties are fulfilled even if the static analysis does not discover that, but any property static analysis determines to be fulfilled must be really fulfilled. *Flow analysis* uses the structure of the program. The control flow represents the program as graph with nodes being statements or program segments. Edges represent the allowed control flow. Data flow considers statements as nodes, the edges represent the control flow. A typical example for control flow analysis is unreachable code detection while typical examples of data flow analysis are uninitialized or unused variable detection. *Abstract interpretation* [15] uses an abstract domain whose elements represent sets of concrete values instead of actual data values. Additionally, abstract operations define the behavior of the concrete operations on the abstract domain. Simple abstract domains are e.g. *negative*, *positive* or *zero* for numerical values. *Model checking* is used to determine if a system model satisfies a given (partial) specification and further explained in the sections below as it plays an important role in this thesis.

3.2 Historical Background of Model Checking

Clarke [13] gives a short historical overview of model checking. Model checking originated from solving the problem of concurrent program verification as automation of manual proof construction using specialized logics like Hoare logic. In the 1980s, people started working on computing the reachable state space of a system for protocol verification. Temporal logics were developed in the 1970s and were first used for concurrency in 1977 by Pnueli [58]. Lamport showed that linear–time logics like LTL and branching–time logics like CTL can express different sets of properties in 1980. Temporal model checking algorithms were provided in the early 1980s for CTL by Emerson and Clarke. They suggested an improved version in 1982 using fixpoints. In 1986, LTL algorithms

were provided. This algorithm was implemented in a model checker with about 10^4 to 10^5 states being searchable. In 1985, model checking was used in hardware verification. Other model checking approaches in the 1980s were based on automata.

In order to handle the state space explosion problem, several approaches were proposed. In 1987, McMillan used a symbolic representation of the state graph using ordered BDDs handling about 10^{20} states. A symbolic model checker was implemented in 1992, but other symbolic model checking algorithms have been presented a few years before. With asynchronous software, often different events are independent of each other. Many approaches in the early 1990s exploited this characteristic to perform an a partial order reduction [13]. Other techniques include exploiting composition in the late 1990s, data value abstraction in the early to middle 1990s, symmetry reduction in the early to middle 1990s and allowing to reason about some sets of system families in the late 1990s. Bounded model checking was introduced in the late 1990s.

The Pentium FDIV Bug [8] resulted in slightly wrong floating point division results in certain cases. Starting with this bug, hardware verification became extremely important [33]. Intel developed a specification language, FSL, a linear temporal logic inspired by LTL, which led to the detection of many bugs which would have only been found later or not at all without model checking. Later, more model checking techniques were introduced [28]. Model checking of cache protocols now is routine in processor design [47]. The SLAM toolkit [5] of Microsoft is used as part of the static driver verifier for Windows drivers using model checking. Today, model checking is also used in the verification of aircraft systems [52] and in various other areas like legal contracts, processes in living organisms and e-business processes [24] but success stories are mainly written in the area of hardware model checking [47].

3.3 Basic Idea

Christel Baier and Joost–Pieter Katoen [4] provide a good overview about basic model checking concepts used on which we base the short introduction given in this section. Roughly, we consider model checking as evaluating whether a formal property holds for a certain state in a finite–state model of a system.

In model checking, the *system's specification* describes the requirements for the system. A system is considered correct if it fulfills all requirements occuring in this specification. The model describing the system itself has to be specified in a precise manner. Often, the process of formalizing the system's description and its specification already helps discovering inconsistencies.

The process of model checking can be described as follows:

- *Modeling*: The model of the system and the property is described using the input language of the model checker

- *Running*: The validity of the property is asserted

- *Analysis*: If the property is violated, the generated counterexample is analyzed, the model is corrected and the property is checked again. If the model checker cannot check the property, the model needs to be simplified.

Model checker models are often expressed as *finite–state automata* consisting of a set of states and transitions between states. The states are characterized by properties of the system at a specific time and the transitions describing how the states are related to each other. Properties to be checked are often specified in terms of a *temporal logic*, a propositional logic extended by temporal operators. In contrast to classical propositional or first–order logic, which allows to check properties of single states, temporal operators allow reasoning over multiple states connected by transitions.

Note, however, that even though model checking provides means to guarantee that a system fulfills a property, this does not mean that the system behaves as expected. In other words, model checking can help solving the *verification problem*, i.e. whether the system fulfills the specification, but not the *validation problem*, i.e. whether the system fulfills the actual needs. Providing model checking on layers closer to the actual modeling layers might reduce the validation problem because the gap between the system modeled for the model checker and for humans is reduced.

A property falsified by model checking may have different causes: A *modeling error* occurs when the model is not describing the design right. Then, the model has to be changed and all properties revalidated. A *property error*, on the other hand occurs if the property does not describe the requirement right. If both modeling and property errors can be excluded, a falsified property means a *design error* has been found in which case the design has to be improved. In many cases, errors found by model checking are errors occurring from unforeseen interleaving of processes.

The state space describing the execution of a system can be represented either in an explicit or in a symbolic way. Whereas the explicit representation directly captives the behavior of a system, it easily gets too large. Symbolic approaches provide a compressed form by considering sets of states.

3.4 Computational Tree Logic

CTL, the computational tree logic [14], is a branching–time temporal logic for specifying properties in a world consisting of a set of discrete states. Each state is characterized by a set of propositional properties. Transition relations describe connections between states, i.e., they describe when it is possible to go from one state to the next state. For introducing CTL, at first a state transition system allowing us to specify the system to be verified has to be introduced.

In the following, a weather forecast example depicted in Figure 16 is used. Principally, we could have also considered the dining philosophers problem, but for the ease of presentation, we show a smaller example. The weather for today is known. For every day, the weather of the next day depends on the direction of the wind and the current weather. For example, if there is `south wind`, then on Monday it will have 22°C and it

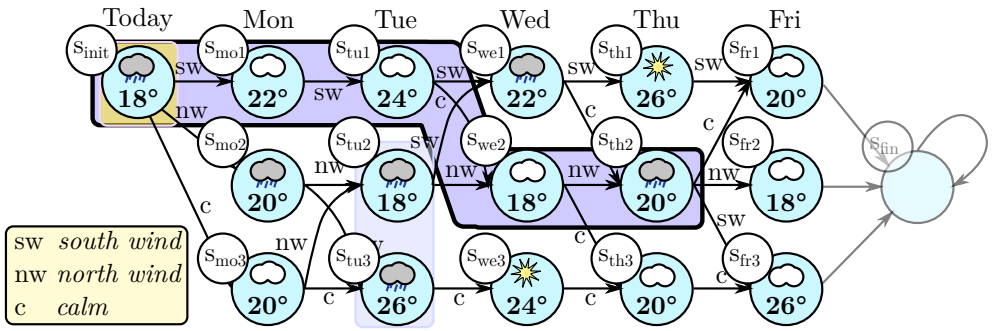


Figure 16: Weather forecast transition system

will be cloudy. A sequence of wind directions thus induces a certain weather sequence. If there is south wind today and on Monday, then it is calm and then there is north wind, it will be cloudy or raining all of the time as depicted in the highlighted path.

In general, the typical context for CTL statements is a transition system. There are various types of transition systems in the literature. Here, we follow the definition of a transition system given by Baier [4].

Definition 16 (Transition System)**.** A *transition system* TS $= (S, Act, \rightarrow, I, AP, L)$ is a tuple with a set of states S, a set of actions Act, a transition relation $\rightarrow \subseteq S \times Act \times S$, a set of initial states $I \subseteq S$, a set of atomic propositions AP and a labeling function $L : S \rightarrow 2^{AP}$. A transition system is called is called *finite* if it has a finite number of states, actions and atomic propositions. A transition relation is called *total* iff that there is at least one outgoing transition for each state. □

Example 22 (Transition system). The transition system in Figure 16 represents a weather forecast where temperature and cloudiness depend on the wind direction. Every circle represents a state, the three actions are *nw*, *sw* and *c*, the arrows depict the transition relation and the `today` state is the single initial state. The assigned atomic propositions describing the weather are *cloudy*, *raining* and *sunshine*. The temperature properties are *tempX* with X being equal to the temperature. The transition relation of this transition system is total only if the state s_{fin} is in the transition system.

A simple way to traverse a transition system is to start from a state and successively go to next states. For this purpose, the *next* function is defined.

Definition 17 (Next function)**.** Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. The successor function $next(s) = \{s_t | \exists a : (s, a, s_t) \in \rightarrow\}$ is defined as the set next states of a state s, i.e. the states occurring as target states of the transition relation for a given source state. □

Example 23 (Next function). This example illustrates the *next* function. The notation of $f(x) \overset{\text{ret}}{\Longrightarrow} y$ indicates that the the function f with parameter x has result y.

When starting from the state s_{we2}, we can reach the state s_{th2} by a transition labeled *nw* and the state s_{th3} by a transition labeled *c*. Thus, $\texttt{next}(s_{we2}) \overset{\text{ret}}{\Longrightarrow} \{s_{th2}, s_{th3}\}$

The sequence of states and transitions of a traversal of the transition system is described by paths and path fragments and execution paths. They differ in whether transitions are included in the description of the traversal and

Definition 18 (Paths and path fragments)**.** Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A *path fragment* $\pi = s_0 s_1 ... s_n$ with $\forall\ 0 \leqslant i < n : \exists a_i : (s_i, a_i, s_{i+1}) \in \rightarrow$ is a sequence of states connected by transitions. The length of a path $length(\pi)$ is n for finite paths and ∞ for infinite paths. A path fragment π is called *finite* if $length(\pi) \in \mathbb{N}$, else it is called *infinite*. A path π is called *maximum path*, iff $length(\pi) = \infty \vee \nexists a, t : (s_{length(\pi)}, a, t) \in \rightarrow$, i.e. it is infinite or the last state has no outgoing transition. The projection function $\pi(i) = s_i$ returns the i–th state of a path fragment π. A path fragment is called *initial*, iff $\pi(0) \in I$, i.e. the first state of the path fragment is an initial state. An initial, maximal path fragment is called *path*. An *execution path fragment* $f = s_0 a_0 s_1 a_1 ... s_n$ with $\forall\ 0 \leqslant i < length(\pi) : \ (s_i, a_i, s_{i+1}) \in \rightarrow$ is a sequence of states connected by the given transitions. All definitions for path fragments also apply for execution path fragments. □

Example 24 (Paths). In the transition system depicted in Figure 16 the highlighted *path fragment* is $s_{init} s_{mo1} s_{tu1} s_{we2} s_{th2}$. The highlighted execution path fragment is $s_{init} sw s_{mo1} sw s_{tu1} cs_{we2} nw s_{th2}$. The *path fragment* could be extended by the state s_{fr2} to form the *path* $s_{init} s_{mo1} s_{tu1} s_{we2} s_{th2} s_{fr2}$ if the sink is not added to the statespace, else a path could be $s_{init} s_{mo1} s_{tu1} s_{we2} s_{th2} s_{fr2} s_{fin} s_{fin} ...$.

Syntax and Semantics The syntax of CTL is described with respect to a transition system $TS = (S, Act, \rightarrow, I, AP, L)$ [4].

Definition 19 (CTL formulae)**.** Given a set of atomic propositions AP, the maximal set $\mathcal{L}$ of syntactically valid CTL formulas is defined as follows:

- $\top, \bot \in \mathcal{L}$.
- If $p \in AP$, then $p \in \mathcal{L}$.
- If $\phi \in \mathcal{L}$, then $\neg(\phi) \in \mathcal{L}$
- If $\phi_1, \phi_2 \in \mathcal{L}$ then, $(\phi_1 \wedge \phi_2)$, $(\phi_1 \vee \phi_2)$, $(\phi_1 \rightarrow \phi_2) \in \mathcal{L}$
- If $\phi \in \mathcal{L}$, then $(\mathsf{E}\ \mathsf{X}\ \phi), (\mathsf{A}\ \mathsf{X}\ \phi), (\mathsf{E}\ \mathsf{F}\ \phi), (\mathsf{A}\ \mathsf{F}\ \phi), (\mathsf{E}\ \mathsf{G}\ \phi), (\mathsf{A}\ \mathsf{G}\ \phi) \in \mathcal{L}$
- If $\phi_1, \phi_2 \in \mathcal{L}$ then $(\mathsf{E}\phi_1\mathsf{U}\phi_2)$, $(\mathsf{A}\phi_1\mathsf{U}\phi_2)$, $(\mathsf{E}\phi_1\mathsf{W}\phi_2)$, $(\mathsf{A}\phi_1\mathsf{W}\phi_2) \in \mathcal{L}$

□

The literal $\top$ is read as *true*, $\bot$ is read as *false*. The operators $\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U}$ and W are called *state operators* and read as follows: X is read as *next*, G as *globally*, F as *finally* or *eventually*, U as *until* and W as *weak until*. The operators A and E are called *path operators* and read as follows: A is read as *always* or *for all* and E as *exists*. Parentheses

are be omitted if no ambiguities arise using the usual boolean operator precedence with $\rightarrow$ binding weaker than $\vee$ binding weaker than $\wedge$.

Example 25 (Syntactically valid and invalid CTL expression:).

Syntactically valid CTL expressions are:

$\mathit{cloudy} \vee \mathit{raining}$, $(\mathsf{A}\,\mathsf{X}\,\neg\mathit{raining}) \wedge (\mathsf{E}\,\mathsf{X}\,\mathit{sunshine})$, $\mathsf{A}\,\mathsf{X}\,(\mathit{sunshine} \rightarrow (\mathsf{A}\,\mathsf{G}\,\neg\mathit{raining}))$

Syntactically invalid CTL expressions are: $\mathsf{A}\ \mathit{sunshine}\ \mathsf{W}\ \mathrm{temp22}$ is not a valid CTL expression because cold does not occur in AP. $\mathsf{A}\ ((\mathsf{X}\ \mathit{sunshine}) \vee \mathit{cloudy})$ is not a valid CTL expression because $\vee$ may only be used for CTL state expressions but $(\mathsf{X}\ \mathit{sunshine})$ is a CTL path expression.

Definition 20 (Semantics of CTL expressions [4])**.** Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system, $\mathcal{L}$ the set of syntactically valid CTL formulae, $s \in S$, $a \in AP$, $\Phi, \Psi \in \mathcal{L}$. Then, the satisfaction relation $s \models \phi$ with the intended meaning that ϕ holds in a state s is defined as follow:

1. $s \models a$ iff $a \in L(s)$
2. $s \models \neg\phi$ iff $s \not\models \phi$.
3. $s \models \phi \wedge \psi$ iff $(s \models \phi)$ and $(s \models \psi)$
4. $s \models \mathsf{E}\,\mathsf{X}\,\phi$ iff there exists a path π with $\pi(0) = s$ such that $\mathit{length}(\pi) > 0$ and $\pi(1) \models \phi$.
5. $s \models \mathsf{A}\,\mathsf{X}\,\phi$ iff for all paths π with $\pi(0) = s$, the conditions $\mathit{length}(\pi) > 0$ and $\pi(1) \models \phi$ hold.
6. $s \models \mathsf{E}\ \phi\ \mathsf{U}\ \psi$ iff there exists a path π with $\pi(0) = s$ such that $\exists j \in \mathbb{N}, \pi(0) = \mathit{length}(\pi) \geqslant j \geqslant 0 : (\pi(j) \models \psi \wedge (\forall 0 \leqslant k < j : \pi(k) \models \phi))$
7. $s \models \mathsf{A}\ \phi\ \mathsf{U}\ \psi$ iff for all paths with $\pi(0) = s$, the condition $\exists j \in \mathbb{N}, \pi(0) = \mathit{length}(\pi) \geqslant j \geqslant 0 : (\pi(j) \models \psi \wedge (\forall 0 \leqslant k < j : \pi(k) \models \phi))$ holds.

□

Note that because $s \models \mathsf{A}\ \mathsf{G}\ \phi \Leftrightarrow s \models \mathsf{A}\ \phi\ \mathsf{W}\ \bot$, $s \models \mathsf{A}\ \mathsf{F}\ \phi \Leftrightarrow \mathsf{A}\ \top\ \mathsf{U}\ \phi$ and $s \models \mathsf{A}\ \phi\ \mathsf{W}\ \psi \Leftrightarrow \neg\mathsf{A}\ (\phi \wedge \neg\psi)\ \mathsf{U}\ (\neg\psi \wedge \neg\psi)$ and all these equivalences hold with E instead of A as well, it is sufficient to provide semantics for one of these expression types.

Definition 21 (Satisfiability)**.** Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. Then, a CTL expression ϕ is called *satisfiable* iff $\forall\ s_0 \in I : s_0 \models \phi$, i.e. ϕ holds in all initial states. □

Expression	Intended Meaning
$s \models \mathsf{A}\ \mathsf{G}\ \phi$	In all paths starting with s, ϕ holds.
$s \models \mathsf{E}\ \mathsf{G}\ \phi$	In at least one path starting with s, ϕ holds.
$s \models \mathsf{A}\ \mathsf{F}\ \phi$	In all paths starting with s, ϕ holds at some point in the future.
$s \models \mathsf{E}\ \mathsf{F}\ \phi$	In at least one path starting with s, ϕ holds at some point in the future.
$s \models \mathsf{A}\ \mathsf{X}\ \phi$	In all states having a transition from s, ϕ holds.
$s \models \mathsf{E}\ \mathsf{X}\ \phi$	In at least one state having a transition from s, ϕ holds.
$s \models \mathsf{A}\ \phi_1\ \mathsf{U}\ \phi_2$	In all paths starting with s, ϕ_2 holds at some point in the future and ϕ_1 before.
$s \models \mathsf{E}\ \phi_1\ \mathsf{U}\ \phi_2$	In at least one path starting with s, ϕ_2 holds at some point in the future and ϕ_1 before.
$s \models \mathsf{A}\ \phi_1\ \mathsf{W}\ \phi_2$	In all paths starting with s, either ϕ_1 holds forever or ϕ_2 holds at some point in the future and ϕ_1 before.
$s \models \mathsf{E}\ \phi_1\ \mathsf{W}\ \phi_2$	In at least one path starting with s, either ϕ_1 holds forever or ϕ_2 holds at some point in the future and ϕ_1 before.

Table 2: Meaning of CTL expressions in natural language

Table 2 shows the 10 basic CTL expression types. All CTL expressions have CTL path expressions directly enclosed by a CTL state operator. Table 3 shows CTL expressions which are satisfiable w.r.t. the weather forecast of Figure 16.

Example 26 (CTL expression formulation and evaluation).

The expression *Is it ok to go for picnic tomorrow?* as defined as that there is maybe sunshine and no rain, can be formulated as $(\mathsf{E}\ \mathsf{X}\ \mathit{sunshine}) \wedge (\mathsf{A}\ \mathsf{X}\ \neg \mathit{raining})$

To evaluate this expression, in principle we have to check all paths starting with s_{init}, for example the path $s_{init}, s_{mo1}, s_{tu1}, s_{we2}, s_{th2}, s_{fr2}$, but also $s_{init}, s_{mo1}, s_{tu1}, s_{we1}, s_{th2}, s_{fr2}$ and other paths. We, however, can conclude that after the states s_{mo1}, s_{mo2} or s_{mo3} the remainder of the path does not matter any longer and thus can be ignored. If all next states fulfill the condition, then nothing may happen in the remainders to invalidate the formula. If there exists one next state fulfilling the condition, then there might be many, many paths all starting with the initial state and this one next state fulfilling the condition, but there at least is one. If, how ever, no next state fulfills the condition, then there is surely no path fulfilling the condition. Thus, we only have to check the next states as depicted in Figure 17 where a denotes $\neg \mathit{raining}$ and s denotes *sunshine*. We see that the statement is false because there is a next state where it is raining and additionally there is no next state where there is sunshine.

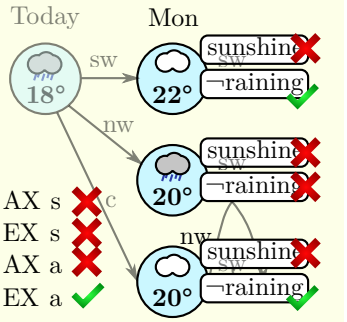


Figure 17: Next states of the initial state

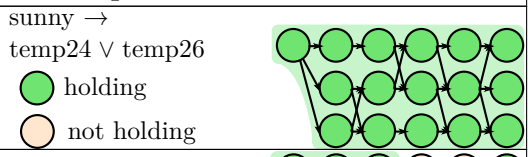
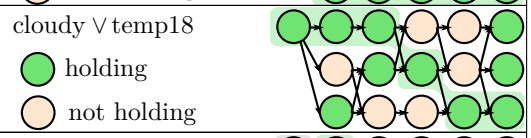
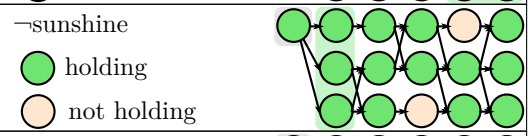
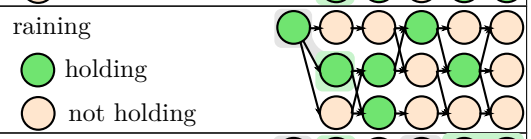
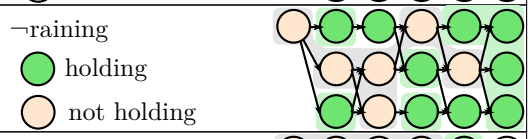
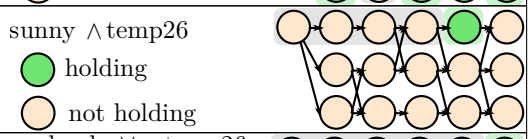
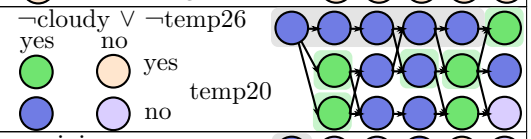
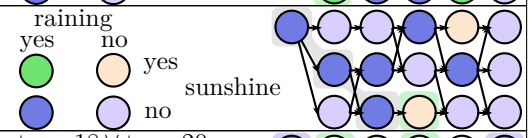
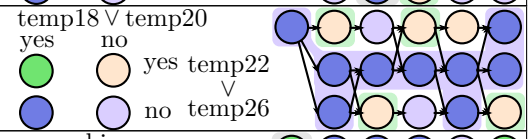
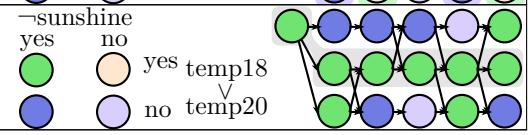
Expression	State space
A G (sunny → ¬(temp24 ∨ temp26))	
E G cloudy ∨ temp18	
A X (¬sunshine)	
E X (raining)	
A F ¬raining	
E F (sunny ∧ temp26)	
A (¬cloudy ∨ ¬ temp26) U (temp20)	
E raining U sunshine	
A (temp18 ∨ temp20) W (temp22 ∨ temp26)	
E ¬sunhine W (temp18 ∨ temp20)	

Table 3: Examples of CTL expressions

Example 27 (CTL expression formulation and evaluation II). The expression *Is it always raining before it gets sunny* can be formulated in different ways based on what is exactly meant. If formulated as A *raining* W *sunshine* if we do not need to know that it will get sunny or using U if it must get sunny at some point in time. Note that even if using weak until, it is required that it is raining all the time if there is no hope for sun. If we want to specify that it has only to be raining before it is sunny but nothing is required if it does not get sunny, we can express this as A (E F *sunshine*) → *raining*) W *sunshine*. This expression is a bit more complicated since the thing we want to express is no CTL base expression. It means that unless we have hit a state where it is sunny, it has to rain if it might be sunny in the future because if it would not rain, we would have found a path where it does not always rain, but gets sunny later on.

In our state space, neither expression is true.

Example 28 (CTL expression formulation and evaluation III). The expression *If it is sunny, it will not rain for the rest any longer* can be formulated as A G (*sunshine* → (A G ¬*raining*)). It is true because we have only two states where it is sunny, namely s_{we3} and s_{th1}. There are only two paths starting at these states, namely, $s_{we3}, s_{th3}, s_{fr3}$ and s_{th1}, s_{fr1} and it is always cloudy there.

Extensions and Alternatives to CTL

Commonly used alternatives to CTL are the Linear Temporal Logic (LTL) [58] and CTL* [25]. While CTL is a branching–time logic, i.e. it considers all possible paths, LTL is a linear–time logic, i.e., it considers the execution of a specific single path. An LTL expression defines that all possible execution traces must fulfill some condition. This condition is specified by the quantifiers X, F, G, U and W in arbitrary combination. Since all paths must fulfill this condition, a single A is prepended to the formula, and E is not allowed. Since there are both CTL expressions not expressible in LTL and vice versa, a superset of CTL and LTL, CTL* has been defined. It allows all operators to be used in arbitrary order. Another language, even more expressive than CTL*, is the μ–calculus [26]. It operates on a set of states fulfilling a condition including the greatest and smallest fixpoint operator. Fixpoint operators in fact are used for the implementation of CTL in this thesis. CTL is used in this thesis because it is an easy to implement branching–time logic suitable for expressing many interesting verification tasks.

3.5 Summary

While testing is an easy and fast method for finding bugs, the absence of bugs cannot be guaranteed. Manual proof methods are applicable to infinite state systems but are cumbersome while model checking allows to give fully automated guarantees of finite–state programs. Model checking consists of formulating the system under consideration in a concise way, running the analysis and interpreting the result. Quite a lot of research has been conducted to combat the state space explosion problem to be able to

handle real–world problems. Nowadays, model checking is successfully applied in industry. However, model checking requires to transform the models used in MDE to a representation suitable for model checkers. Further, the properties to be checked have to be formulated upon these transformed models, requiring to change techonological space. In the following, we present an approach to reduce this gap by integrating MDE and model checking technologies.

CHAPTER 4

Related Work

There are several works in the context of integrating MDE with model checking. We survey related literature and (1) give an overview on approaches which specify a clear semantics for languages for model checking of software models, and (2) we review existing tools implementing model checking facilities for MDE. Finally, we define certain criteria for comparing and evaluating related approaches and select a suitable candidate for integrating our ideas.

4.1 Languages

A first requirement for model checking is a temporal language with clearly defined semantics. Thus, OCL based model checking extensions have been proposed using different kinds of techniques for balancing usability and expressiveness. Some of these languages which seemed to be most promising are described here. Additionally, mCRL2 [34] is discussed because of its great expressiveness, even though it is not an OCL extension.

TOCL

Gogolla and Ziemann [69, 71] proposed a formal foundation for an LTL extension of OCL called *Temporal OCL* (TOCL) using linear state sequences of UML object models. The basic idea is that while OCL can be used to express global invariants and properties regarding a single transition, it cannot be used for reasoning over multiple transitions. LTL allows reasoning over multiple transitions, so it is used to extend OCL.

The TOCL syntax extends the OCL syntax by $Expr_{Boolean} \cup =$ next e | always e | sometime e | always e_1 until e_2| sometime e_1 before e_2 and $Expr_t \cup = \omega@\mathsf{next}(e_1, ..., e_n)$ with $\omega \in Expr_t$.

TOCL expressions are evaluated in environments $\tau = (\hat{\sigma}, i, \beta)$ with $\hat{\sigma}$ being a state sequence, i being the state reference index and β being the variable assignment. Unusually for LTL, all state sequences have finite length $|\hat{\sigma}|$.

The semantics of TOCL as given in Definition 2 in [71] and [70] is as follows. The first rules are used from Definition 2 in [62] and can be also found as first rules of our cOCL extension in Section 30.

vii. $I[\![\mathsf{next}\ e]\!](\tau) \Leftrightarrow (i = |\hat{\sigma}| - 1) \vee I[\![e]\!](\hat{\sigma}, i+1, \beta)$. The operator next evaluates to true if there is no next state or e evaluates to true in the next state. This operator corresponds to the next (X) operator in LTL.

viii. $I[\![\mathsf{always}\ e]\!](\tau) \Leftrightarrow \forall i \leqslant j < |\hat{\sigma}| : I[\![e]\!](\hat{\sigma}, j, \beta)$. The operator always evaluates to true if e evaluates to true for all states of the path starting with a given reference state. This operator corresponds to the globally (G) operator in LTL.

ix. $I[\![\mathsf{sometime}\ e]\!](\tau) \Leftrightarrow \exists i \leqslant j < |\hat{\sigma}| : I[\![e]\!](\hat{\sigma}, j, \beta)$. The operator sometime evaluates to true if e evaluates to true for at least one state of the path starting with the reference state. This operator corresponds to the finally (F) operator in LTL.

x. $I[\![\mathsf{always}\ e_1\ \mathsf{until}\ e_2]\!](\tau) \Leftrightarrow \forall i \leqslant j < min\{\{k \geqslant i \mid I[\![e_2]\!](\hat{\sigma}, k, \beta)\} \cup \{|\hat{\sigma}|\}\} : I[\![e]\!](\hat{\sigma}, j, \beta)$. The operator always ... until evaluates to true if, as long as e_2 does not evaluate to true, e_1 does evaluate to true. This operator corresponds to the weak until (W) operator in LTL.

xi. $I[\![\mathsf{sometime}\ e_1\ \mathsf{before}\ e_2]\!](\tau) \Leftrightarrow \exists i \leqslant j < min\{\{k \geqslant i \mid I[\![e_2]\!](\hat{\sigma}, k, \beta)\} \cup \{|\hat{\sigma}|\}\} : I[\![e]\!](\hat{\sigma}, j, \beta)$. The operator sometime ... before evaluates to true if e_1 evaluates to true somewhere before e_2 evaluates to true for the first time. This operator has no direct translation to an LTL operator, but is equivalent to $\neg((\neg e_1)\ W\ e_2)$ in LTL.

xii. $I[\![w@\mathsf{next}(e_1, ..., e_n)]\!](\tau) = \begin{cases} I(\hat{\sigma}, i+1)(\omega)(I[\![e_1]\!](\tau), ..., I[\![e_n]\!](\tau)) & \text{if} |\hat{\sigma}| > i+1 \\ \bot & \text{otherwise} \end{cases}$
This operator has the intended meaning of evaluating the parameter values e_1, e_2, ..., e_n in the current state, but then evaluating ω in the next state.

Additionally, semantics for the past temporal operators `previous`, `always ... since` are defined analogously.

The variable assignment is not changed when switching between different states, but the current state affects object operations like attribute access or navigation. Further, invariants and pre– and postconditions can be specified using TOCL.

Temporal OCL

Kanso and Taha [42] extend OCL by an adaption of patterns introduced by Dwyer [21] which are translated into other formalisms like CTL, LTL or the μ–calculus. Their idea is that temporal logic is too complicated to use in practice and they wanted to provide a user friendly OCL extension usable in automated testing.

A temporal property is specified by a pattern `<pattern>` and a scope `<scope>`, where one of eight predefined occurrence patterns `Absence`, `Existence`, `BoundedExistence`, `Universality`, `Precedence`, `Response`, `ChainPrecedence` and `ChainResponse` may be combined

with one of the five scopes `Globally`, `Before` Q, `After` Q, `Between` Q `and` R and `After` Q `until` R

These patterns are evaluated on finite sequences of atomic events. These *atomic events* $e \in \Sigma = \mathbb{O} \times \mathbb{E} \times \mathbb{E}$ are defined over a set of operations $\mathbb{O}$ and a set of pre– and postcondition expressions $\mathbb{E}$. The events *isCalled* and *becomesTrue* are subsets of the set of all atomic events:

- $isCalled(op, pre, post) = \{(op, pre', post') \in \Sigma \mid pre' \Rightarrow pre, post' \Rightarrow post\}$
- $becomesTrue(P) = \{(op, pre, post) \in \Sigma \mid op \in \mathbb{O}, pre \Rightarrow \neg P, post \Rightarrow P\}$

The event *isCalled* occurs on operation calls or state changes where *op* specifies the operation of the event and *anyOp* specifies that no specific operation is required. The expressions *pre* and *post* are pre– and postconditions necessary to make the event fire. Additionally, the disjunction operator | and the exclusion operator \ are operators on events.

A scenario $\sigma = (\sigma(0), ...\sigma(n-1)) \in \Sigma^*$ is defined as sequence of length n of atomic events describing a temporal order of events. $\sigma(i)$ denotes the atomic event at index i and $\sigma(i:j)$ identifies the subsequence of atomic events between indices i and j.

Scopes $s \in \mathbb{S}$ specify the part of the scenario where the pattern should hold and are defined as follows.

- $[\![\mathsf{globally}]\!]^s(\sigma) = \{\sigma\}$. This operator evaluates to the set containing the full scenario.
- $[\![\mathsf{before}\ E]\!]^s(\sigma)\{\sigma(0:i-1) \mid \sigma(i) \in E \text{ and } \forall k, 0 \leqslant k < i, \sigma(k) \notin E\}$. This operator evaluates to the set containing the largest subscenario where the event E does not yet occur.
- $[\![\mathsf{After}\ E]\!]^s(\sigma) = \{\sigma(i+1:n-1) \mid \sigma(i) \in E \text{ and } \forall k, 0 \leqslant k < i, \sigma(k) \notin E\}$. This operator evaluates to the set containing the subscenario after the first occurrence of the event E.
- $[\![\mathsf{between}\ E_1\ \mathsf{and}\ E_2]\!]^s(\sigma) = \{\sigma(i_k + 1 : j_k - 1) \mid \forall k \geqslant 0, i_k < j_k < i_{k+1}, \sigma(i_k) \in E_1, \sigma(j_k) \in E_2, \forall m, i_k \leqslant m < j_k, \sigma(m) \in E_2 \text{ and } \forall l, j_k < l < i_{k+1}, \sigma(l) \notin E_1\}$. This operator evaluates to the set containing all subscenarios starting after E_1 occurred and ending just before the next occurrence of E_2.
- $[\![\mathsf{after}\ E_1\ \mathsf{unless}\ E_2]\!]^s(\sigma) = \{\sigma(i_k + 1 : j_k - 1) \mid \forall k \geqslant 0, i_k < j_k < i_{k+1}, \sigma(i_k) \in E_1, \sigma(j_k) \in E_2, \forall m, i_k \leqslant m < j_k, \sigma(m) \notin E_2 \text{ and } \forall l, j_k < l < i_{k+1}, \sigma(l) \notin E_1\} \cup \{\sigma(i : n-1) \mid \sigma(i) \in E_1, \forall m \geqslant i, \sigma(m) \notin E_2\}$. This operator evaluates to the set containing all subscenarios starting after E_1 and ending before the next occurrence of E_2 or the scenario end.

The concrete syntax of TOCL extends the afore mentioned scopes by an additional `when` P scope selecting subsequences where an OCL expression holds. Ranges can be specified in– or exclusive borders and for `between` and `after`, it is possible to select the last event only, but no exact semantics is given.

Patterns $p \in \mathbb{P}$ describe properties which have to hold at the scope s.

- $[\![\mathsf{never}\ E]\!]^p(\sigma) \Leftrightarrow \forall i \geqslant 0, \sigma(i) \notin E$. This operator evaluates to true if and only if E does not occur in the scenario.
- $[\![\mathsf{always}\ P]\!]^p(\sigma) \Leftrightarrow [\![\mathsf{never}(\mathit{isCalled}(\mathit{anyOp}, \_, \neg P))]\!]^p(\sigma)$. This operator evaluates to true if and only if P holds as postcondition in every atomic event.
- $[\![E_1\ \mathsf{preceding}\ E_2]\!]^p(\sigma) \Leftrightarrow \forall i \geqslant 0, (\sigma(i) \in E_2 \Rightarrow \exists k \leqslant i, \sigma(k) \in E_1)$. This operator evaluates to true if and only if there is an occurrence of E_1 before (or at) every occurrence of E_2.
- $[\![E_1\ \mathsf{following}\ E_2]\!]^p(\sigma) \Leftrightarrow \forall i \geqslant 0, (\sigma(i) \in E_2 \Rightarrow \exists k \geqslant i, \sigma(k) \in E_1)$. This operator evaluates to true if and only if there is an occurrence of E_2 after (or at) every occurrence of E_1.
- $[\![\mathsf{eventually}\ E\ \alpha\ \mathsf{times}]\!]^p(\sigma) \Leftrightarrow |\{i \mid \sigma(i) \in E\}| = \begin{cases} = k \text{ if } \alpha = k \\ \geqslant k \text{ if } \alpha = \text{at least } k \\ \leqslant k \text{ if } \alpha = \text{at most } k \end{cases}$
  This operator evaluates to true if and only if E occurs as often as specified.

Kanso and Taha provide a tool for their language extension based on the Eclipse OCL plugin. However, instead of using their temporal OCL expressions for model checking, they translate them into regular expressions, which are used to generate test cases.

EOCL

Extended OCL (EOCL) [53] is an extension of OCL with CTL. EOCL is one of the first OCL extensions implemented in a tool for evaluating temporal properties on UML models. The EOCL syntax is shown in Figure 18.

It consists of two strictly separated parts: On the top level, there is CTL with OCL being on the level below. It is not possible to directly transfer information from one state to another. The semantics for the EOCL constructs are the usual semantics as defined for OCL and CTL. The semantics of EOCL expressions is defined over an Object–Oriented Transition System ($OOTS_{UML}$) which is a structure $\mathcal{OT} = \langle S, EOCL_R, s_0 \rangle$. S is a set of states with associated functions $\rho_s, \sigma_s, \gamma_s$ and h_s. The function $\rho_s : \mathcal{V} \rightarrow \mathit{Val}$ associates values to attributes and method parameters (variables) in a state s. The function $\sigma_s : C \rightarrow [\mathcal{V} \rightarrow \mathit{Val}]$ associates a collection of attribute/method associations to each class, i.e., the object instances for a class. The function $\gamma_s : \mathcal{M} \rightarrow [[\mathcal{V} \rightarrow \mathit{Val}] \rightarrow$

$e(\in P_{exp}) ::= x \mid v\ e.a \mid \omega(e_1, ..., e_n) \mid e_1 \rightarrow \mathbf{iterate}\ \{x_1; x_2 = e_3 \mid e_3\ \} \mid e) \mid$
$e@\mathbf{pre} \mid e.\mathbf{owner} \mid \mathbf{act}(e)$
$\phi(\in F_{exp}) ::= e \mid \neg\phi \mid \phi \wedge \psi \mid \forall z \vdash \tau\ :\ \phi \mid \mathbf{EX}\phi \mid \mathbf{E}[\phi \mathbf{U} \psi] \mid \mathbf{A}[\phi \mathbf{U} \psi]$

Figure 18: EOCL syntax [53]

$Val_\perp]$ associates a collection of method instances to the methods defined for each class including the method parameters. The function $h_s : \mathcal{M} \to S \times \{\circ, \bullet\}$ associates to each method the state where this method has been called together with $\bullet$ indicating that s is the last state for the instance before being returned and $\circ$ otherwise. The set $\mathcal{R} \subseteq S \times S$ is a transition relation and $s_0 \in S$ is the initial state. A class instance $c_i = (c, i) \in \mathcal{C} \subseteq \Sigma_c \times \mathcal{N}$ is defined by a class name $c \in \Sigma_c$ and an index $i \in \mathcal{N}$. A method instance $m = (c, i, m, j) \in \mathcal{M} \subseteq \mathcal{C} \times \Sigma_m \times \mathcal{N}$ is defined by a class instance (c, i), a method name $m \in \Sigma_m$ and a method index $j \in \mathcal{N}$.

The function $[\![\_]\!] : P_{exp} \to [S \to Val_\perp]$ is defined as follows.

- $[\![v]\!]_s = v$. The evaluation of a constant value returns the same value.
- $[\![x]\!]_s = \rho_s(x)$. The evaluation of an attribute or method parameter in a state is the corresponding value defined in the transition system.
- $[\![\omega(e_1, ..., e_n)]\!]_s = \omega([\![e_1]\!]_s, ..., [\![e_n]\!]_s)$. An operation is evaluated by first evaluating its parameters and then evaluating the operation.
- $e.f_s = \begin{cases} \sigma_s((c,i))(f) & \text{if} [\![e]\!]_s = (c, i) \\ \gamma_s((c, i, j, m, j)).f & \text{if} [\![e]\!]_s = (c, i, m, j) \end{cases}$
  The evaluation of an attribute access is the value of the attribute function for object number i of type c for a class instance or or the value of the method function for a method of object number i of class c named m with parameter number j, i.e., the value of parameter j of this method instance.
- $[\![e.\mathbf{owner}]\!]_s = (c, I) \text{for} [\![e]\!]_s = (c, i, m, j)$. The owner of a method instance is the corresponding class.
- $[\![\mathbf{act}(e)]\!]_s = \text{true iff } [\![e]\!]_s \in C_s + \mathcal{M}_s$. An expression is considered to be active if the object or method instance it evaluates to is active in the state s.
- $[\![e_1 \to \mathbf{iterate}\{x_1; x_2 = e_2 | e_3\}]\!]_s = [\![\mathbf{for}\ x_1 \in [\![e_1]\!]_s\ \mathbf{do}\ x_2 := e_3]\!]_{\rho_s[x_2 \mapsto [\![e_2]\!]_s]}$. For this, the evaluation of $[\![\mathbf{for}\ x_1 \in []\ \mathbf{do}\ x_2 := e]\!]_s = [\![x_2]\!]_s$ and the evaluation of $[\![\mathbf{for}\ x_1 \in h :: w\ \mathbf{do}\ x_2 := e]\!]_s = [\![\mathbf{for}\ x_1 \in w\ \mathbf{do}\ x_2 := e]\!]_{\rho_s[x_2 \mapsto [\![e]\!]_{\rho_s[x_1 \mapsto h]}]}$. Thus, the evaluation of an iteration is done by first evaluating the expression e_1 to get the source set. Initially, the accumulator variable x_2 contains the evaluation of e_2. The evaluation of an iteration for the empty list is the value of the accumulator variable. The evaluation of a non–empty list is provided by first updating the accumulator variable with the expression e_3 where x_1 is the first element of the list and then evaluating the iteration for the rest of the list.
- $[\![e@\mathbf{pre}]\!] = \begin{cases} [\![e]\!]_{s'} & \text{if } (c, i, m, j) \in dom(\gamma_s) \text{ and } h_s(c, i, m, j) = (s', \bullet) \\ \perp & \text{otherwise} \end{cases}$
  The value of an expression at the beginning of a method instance is the value of the expression in state s' which is stored in the history of method instance.

EOCL expressions are evaluated in a context as invariant or, in the case of methods, as pre– and postcondition. EOCL is heavily inspired by BOTL [19], a CTL–based logic for model checking object–oriented systems. BOTL, however, does not extend OCL by temporal operators. Yet, OCL invariants and pre– and postconditions are translated to BOTL. The syntax of a context in EOCL is defined as follows:

$$\kappa(\in C_{exp}) ::= \textbf{context}\ C\ \textbf{inv}\ e \mid \textbf{context}\ C\ ::\ M\ \textbf{pre}\ e_1\ \textbf{post}\ e_2$$

Invariants have to be fulfilled whenever an instance of or a class inherited from the context is active and no method of *self* is executing. Postconditions have to hold for every method M instance of the class C if the preconditions held. Both of these operations can be expressed using EOCL constraints.

The semantics of temporal formula is given by the relation $\Vdash \subseteq S \times F_{exp}$.

- $s \Vdash e \Leftrightarrow [\![e]\!]_s = \text{true}$. An OCL expression holds in a state if the OCL expression evaluates to true in this state.
- $s \Vdash \neg\phi_1 \Leftrightarrow s \nVdash \phi_1$. The negation of an expression ϕ_1 holds in a state if the expression ϕ_1 does not hold in the state.
- $s \Vdash \phi_1 \wedge \phi_2 \Leftrightarrow (s \Vdash \phi_1)$ and $(s \Vdash \phi_2)$. An expression combined with and holds in a state if both subexpressions hold in that state.
- $s \Vdash \forall z \vdash \tau : \phi_1 \Leftrightarrow s \vDash \phi_1[z \mapsto v]$ for all $v \in Val^{\tau}$. A forall expression holds in a state if the subexpression ϕ_1 holds in the state no matter which value the variable z has.
- $s \vDash \mathsf{E\ X}\ \phi_1 \Leftrightarrow \exists_{r \in \mathcal{R}un_s(\mathcal{OT})} r[1] \vDash \phi_1$. A next expression holds in a state if there is a run in the transition system starting with s where in the second state of the run the expression ϕ_1 holds.
- $s \vDash \mathsf{E}[\phi_1\ \mathsf{U}\ \phi_2] \Leftrightarrow \exists_{r \in \mathcal{R}un_s(\mathcal{OT})} \exists_{j \geqslant 0} r[j] \vDash \phi_2 \wedge \forall_{0 \leqslant k < j} r[k] \vDash \phi_1$. An exists ... until expression holds if there exists a run in the transition system starting with s where ϕ_2 holds in some state and ϕ_1 holds in all states before.
- $s \vDash \mathsf{A}[\phi_1\ \mathsf{U}\ \phi_2] \Leftrightarrow \forall_{r \in \mathcal{R}un_s(\mathcal{OT})} \exists_{j \geqslant 0} r[j] \vDash \phi_2 \wedge \forall_{0 \leqslant k < j} r[k] \vDash \phi_1$. An always ... until expression holds if for all runs in the transition system starting with s ϕ_2 holds in some state and ϕ_1 holds in all states before.

Thus, the semantics of EOCL corresponds to CTL semantics with OCL expressions usable as propositional formulas with the addition of a forall operator on the CTL level.

mCRL2

The Micro Common Representation Language 2 (mCRL2) [34] is a specification language for distributed systems and protocols. The idea of mCRL2 is to design a process algebra containing the machine independent data types one expects when writing specifications including sets and functional data types. Model checking in mCRL2 uses an extension

of the μ–calculus. mCRL2 is a large and feature rich language, so only a small subset considered as most important to model checking combined with MDE is presented. While mCRL2 does not extend OCL, it is considered because it is the only language which supports temporal operators and object–oriented features at the same level. All other languages only allow to use OCL expressions to calculate the value of properties in the temporal language. The language reasons over so–called sorts which behave like types and are defined using constructors and equations.

State formulas of mCRL2 have the following form [23]:

StateFrm := *DataValExpr* | (*StateFrm*) | `true` | `false` | (`mu`|`nu`) *Id*((*Id*:*SortExpr*=*DataExpr*(,*Id*:*SortExpr*=*DataExpr*)∗))? . *StateFrm* | (*StateFrm* (=> | || | &&) *StateFrm*) | [*RegFrm*] *StateFrm* | <*RegFrm*> *StateFrm* | ! *StateFrm* | *Id* (*DataExprList*)| (`forall` | `exists`) *VarsDeclList* . *StateFrm*

The elements *RegFrm* are regular expressions, including `nil`, of *Action formulas*. These have the following form:

ActFrm := (`tau` | *Action*(| *Action*)∗) |(*ActFrm*) | `val`(*DataExpr*) | `true` | `false` | ! *ActFrm* | (`forall` | `exists`) *VarsDeclList* . *ActFrm* | *ActFrm* (&& | || | =>) *ActFrm*

The elements `DataExpr` are any kind of expressions of sort elements. The elements `DataExprList` contain a variable number of DataExpr. The construct `VarDeclList` contains a list of variable names of the form `Id` together with the sort type and a value. Single actions are transition names. Variables can be put into parameters of the transition to restrict the possible transitions. It is also possible to access model properties [46].

The expression `val` is used to get the value of a boolean expression. The logical operators `||`, `&&` and `!` have their usual meaning, `=>` is the implication. `forall` and `exists` specify that the relevant `ActForm` must hold for all or for some of the elements declared in the `VarDeclList`.

The operator `nu X.<ExprInX>` returns true if and only if the current state is in the greatest fixpoint of `<ExprInX>`, the operator `mu X.<ExprInX>` returns true if and only if the current state is in the least fixpoint of `<ExprInX>`.

The *must* operator `[`*Expr1*`]`*Expr2* returns true if all paths that start in the current state and satisfy `Expr1` do satisfy `Expr2` as well. The *may* operator `<`*Expr1*`>`*Expr2* is true if there exists a path starting in the current state which satisfies `Expr1` and where `Expr2` is valid.

$\mathcal{O}\mu$

$\mathcal{O}\mu$ [9] is an extension of OCL with the observational μ–calculus. Their idea is that while OCL allows to specify contracts between system parts designed in UML, more feature–rich contracts should be provided. It should be possible to define the behavior of a system in dynamic interaction. Temporal logics provide a possibility for describing dynamic properties only, OCL provides a possibility for describing static properties only, the combination allows to describe both static and dynamic properties. $\mathcal{O}\mu$ is defined

$$\Phi = \psi \mid \mathbf{T} \mid \mathbf{F} \mid X \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid \langle\, l, C, \phi \rangle \Phi \mid [l, C, \phi] \Phi \mid \nu\, X.\Phi \mid \mu\, X.\Phi$$

Figure 19: $\mathcal{O}\mu$ syntax

on a transition systems where transitions are not only labeled by a name, but also by structured data. These transitions contain OCL messages with an identifier and source and target of the message.

The syntax of $\mathcal{O}\mu$ is shown in Figure 19. The operators μ and ν denote the least and greatest fixpoints. The *must* operator $[l, C, \phi]\Phi$ and *may* operator $\langle l, C, \phi \rangle \Phi$ differ slightly from the corresponding operators in mCRL2. There are no regular expressions for path specification allowed, just a single transition. C specifies a set of mutable OCL variables, l is a pattern matching the transition and ϕ is an OCL constraint holding at the target state of the transition referring to variables in C. The formal semantics of the may operator is as follows. Let $\rho : C \to V$ be a function assigning variables to their values, then $A \vdash \langle l, C, \phi \rangle \Phi$ if and only if $\exists \rho' : c \notin C \Rightarrow \rho'(c) = \rho(c)$, so the new cell values are equal to the old ones except for the mutable ones and $\exists a, A'$ with $A \stackrel{a}{\Rightarrow} A'$, with a matching l in the context ρ' and $\rho, \rho' \vdash \Phi$ and $A' \vdash_{V,\rho'} \Phi$, so there is a transition matching the requirement and ϕ is fulfilled using both the values of the variables at source and target state of the transition and the target state fulfills Φ. The must operator is defined in a dual way, i.e., $\forall$ instead of $\exists$.

Comparison

Table 4 compares the different approaches with respect to various language features. The logics differ in their expressibility and understandability. The Dwyer patterns used by Temporal OCL are specifically designed to be easy to understand [21], but are at the same time the least expressive. LTL and CTL have incomparable expressiveness, but in general, LTL is preferred because a single program execution trace is linear, CTL is thought to be unintuitive and most CTL formulas used in practice can be expressed using LTL [66]. CTL is typically found in model checkers because it is simpler to verify. The μ–calculus can express both CTL and LTL–formulas, but the fixpoint semantics is even harder to understand than temporal logics. Except for mCRL2, which is completely independent from OCL, every language extends OCL is some way. While BOTL is just "inspired" by OCL, it rather acts as extension of a subset of OCL. All OCL extensions

	TOCL	**Temp. OCL**	**EOCL**	**mCRL2**	$\mathcal{O}\mu$
Underlying logic	LTL	Dwyer pat.	CTL	μ–calc.	μ–calc.
Relation to OCL	Ext.	Ext.	Ext.	None	Ext.
Support of mixture of concepts[a]	No	No	No	Yes	No
Implementation	No	Yes	Yes	Yes	No

Table 4: Features of various languages

[a] Whether it is possible to trace objects in time

however do not allow mixing temporal logics and OCL expressions in arbitrary way: There is a temporal logic layer on top and the OCL layer below. The problem of state space generation is left open for all languages, but for the languages EOCL and mCRL2, the problem has obviously been solved. TOCLs severe restriction is that the language can only be applied to finite paths.

None of the described languages are found to be directly usable because they either do not offer a direct way of implementation like TOCL and $\mathcal{O}\mu$, have no publicly available implementation like EOCL or use the hard to understand μ–calculus.

4.2 Tools

Since the idea of combining MDE with model checking is not new, there exist some tools in this area.

The tools were evaluated in a way to figure out a suitable candidate for extension. A modeler should be able to model with his/her usual modeling tools and knowledge. Thus, an off the shelf modeling tool should be usable. It is uncomfortable to have to switch between multiple tools, so the model checker should be integrated into the modeling environment. Learning new languages is difficult, so the modeling language should be similar or equal to common modeling languages like UML. Likewise, the modeler's property specification knowledge should be reusable. Thus, the language used in model checking should be as similar as possible to common property specification languages like OCL. In the process of modeling, verification is used as debugging. In order to help debugging, the verification output has to help the modeler. Thus, the result should be readable and point to errors in the model.

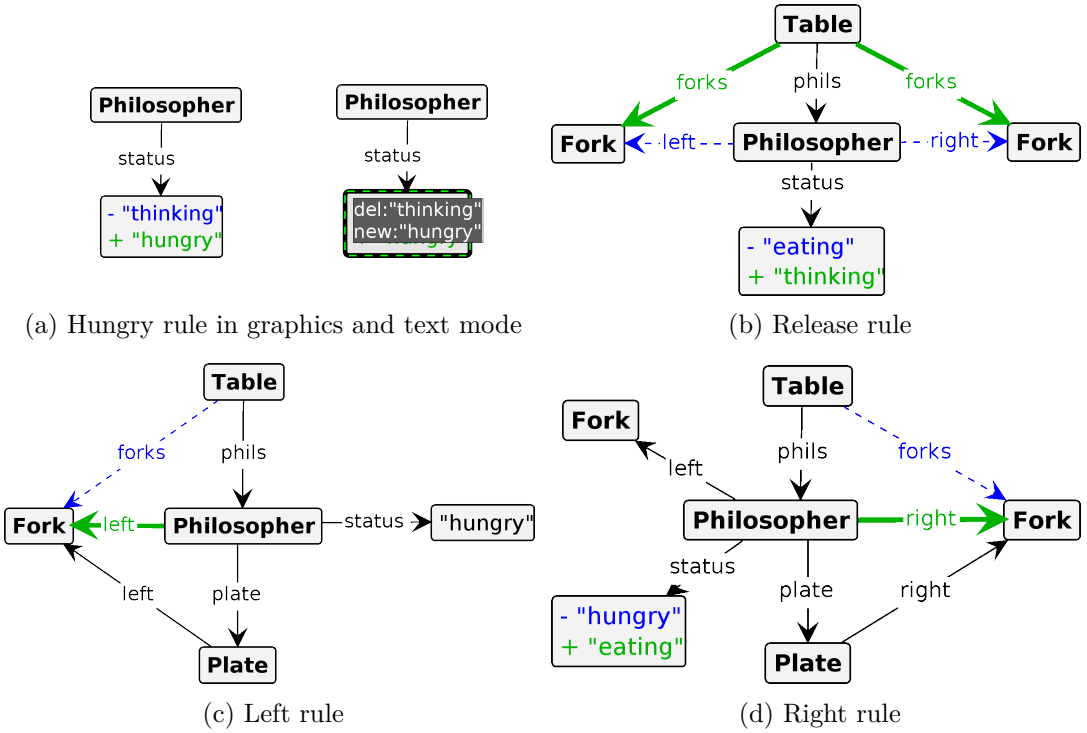
There are also a few nonfunctional requirements. To allow debugging well, the result should not take too long to compute so performance is important. Adaptability might have some advantage in general, but for this thesis it is absolutely required because we aim to select a candidate tool for integrating extensions.

Five tools are selected for different reasons. Groove and Henshin are open source tools for visual graph model checking. SOCLe and Hugo/RT on the other hand promise to do model checking on UML models and CheckVML is able to handle arbitrary models.

Groove

Groove [60] is an open source model checker written in Java allowing to specify graph transformations and to evaluate CTL and LTL expressions.

Graph transformations may be entered interactively using the integrated editor which combines graphical and textual input where editing is done mostly in textual fashion but then is visualized. The graph transformation is displayed in storyboard notation (see Section 2.3) and many operators can be used for matching nodes and edges including exists and forall operators which also allow to specify the number of matching elements, regular expressions for paths with multiple edges and matching node values with arbitrary expressions.



(a) Hungry rule in graphics and text mode

(b) Release rule

(c) Left rule

(d) Right rule

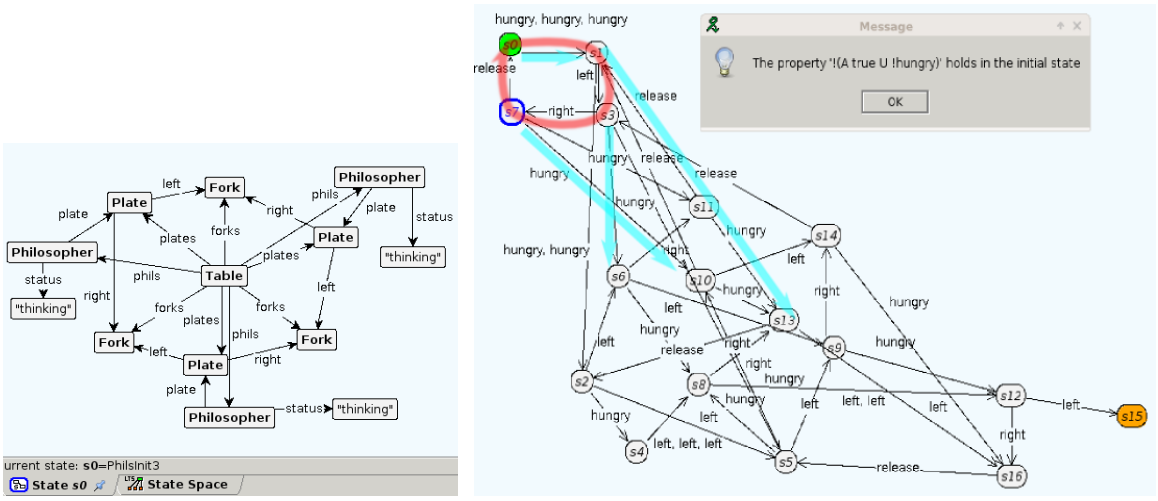
Figure 20: Dining philosopher rules as implemented in GROOVE.

Figure 20 shows the graph transformations for our dining philosophers example as implemented in GROOVE.

CTL and LTL properties here correspond to possible graph transformation applications (`true` and `false` can be used as special properties). Graph transformations which do not change the graph can thus always be used as properties without influencing the state space. For example, the CTL expression `E G hungry` specifies that there is an infinite path where the rule hungry can be applied all the time. This expression is true because there is a cycle (hungry, left, right, release) in which for every state the hungry rule may be applied as shown in Figure 21b. The red cycle path and cyan paths were added to the GROOVE screenshot.

Thus, `E G hungry` does not refer to actual rule executions, i.e., a path where only the rule hungry is executed, but to potential rule applications. Note how GROOVE automatically normalizes entered expressions as seen in the dialogue box.

The state space can be created stepwise or in a single step and for each state, the graph found in this state can be viewed. Isomorphic states are automatically collapsed. The graph layout used there resembles the layout used for the initial graph. Figure 21a shows the initial state of the state space as generated by GROOVE, it exactly matches the initial model in its layout.



(a) Initial state (b) Evaluation of the property E G hungry

Figure 21: Initial state and full statespace of a property in GROOVE.

CheckVML

CheckVML [64] is another tool for verifying models whose behavior is specified using graph transformations. In contrast to GROOVE, its intended application is different. While in GROOVE, graphs and graph transformations are directly used as models, in CheckVML, only the dynamic behavior is modeled as graph transformations in the metamodel layer; there is no direct necessity to use graph transformations for the modeler.

At first, CheckVML derives a transition system out of the provided instance model, the metamodel and its graph transformation rules. Then, a PROMELA description for SPIN is generated and formal verification is done using LTL as provided by SPIN.

Henshin

Henshin [2] is an open source Ecore model transformation language implementing the double pushout approach (see Section A) including verification. There is a visual editor for graph transformations for Eclipse using the storyboard notation. Like in GROOVE, some of the editing is done textually and then represented in a different form, but the differences are smaller than in GROOVE; e.g., quotes are automatically added to association stereotypes. The visual editor's graph matching functionality is more restricted than in GROOVE, some not commonly used functions are only available using the tree editor.

Figure 23 shows graph transformations for the dining philosophers problem.

The state space shown in Figure 22 can be fully or partially created with interactive layouting. While inspecting states is possible, more clicks are needed to display states in Henshin than in GROOVE. Verification is possible using various external model checkers like CADP and mCRL2 providing support for the μ–calculus, and PRISM, a probabilistic

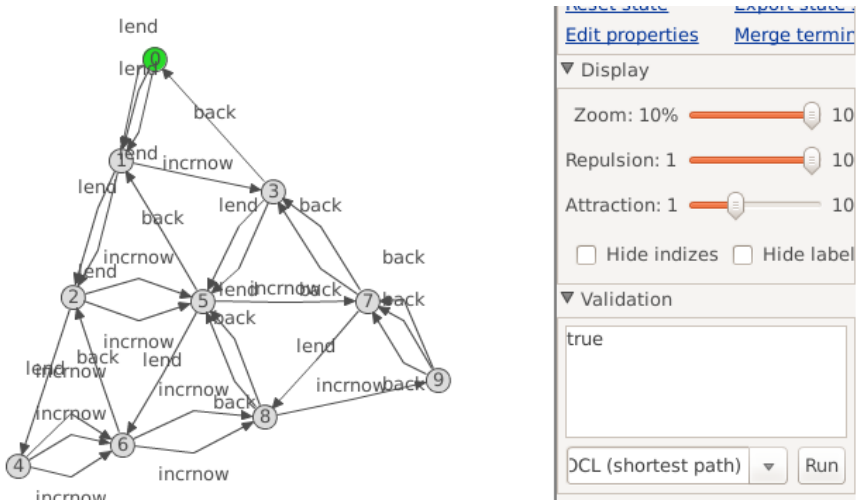


Figure 22: State space in Henshin

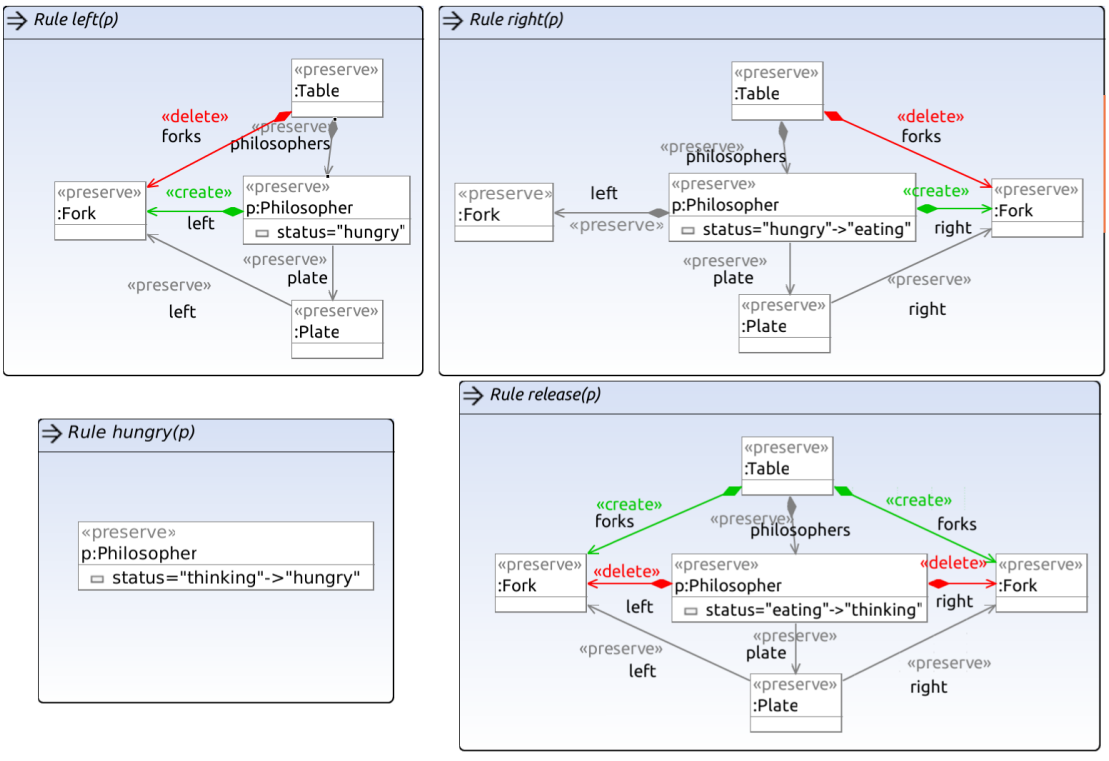


Figure 23: Dining philosopher rules in Henshin

model checker. OCL invariants may be evaluated on each state of the state space. However, no temporal properties may be checked therewith.

SOCLe

SOCLe [53] is a model checking tool for checking EOCL constraints on UML models. Unfortunately, it seems to be not available to the public any longer. EOCL (see the section above) extends OCL by CTL operations and some first–order logic features. Abstract State Machine (ASM) semantics are used to define semantics for class, statechart and object diagrams.

The verification of UML models is done by first translating the UML model into the ASM specification and then generating an execution graph from the ASM specification where the OCL constraints are verified on the fly. An ArgoUML integration provides some text window including verification results and an execution diagram.

Hugo/RT

Hugo/RT [44] is a model checking tool for a subset of UML. Properties can then be expressed using interaction diagrams where it is checked whether some interaction may happen or not or using assertions in the form of CTL and LTL expressions.

Classes of the class diagram subset have attributes of type integer, boolean, clock or can refer to other classes. Arrays are supported, but there are no other collection types. Operations and signals with input parameters can be declared. Their behavior is expressed using state diagrams. Simple states can be connected using transitions reacting on triggers equivalent to method calls with boolean guards allowing to use mathematical functions, the ternary operator and access to object attributes. Transition effects include sending signals or method invocations, assertions, conditions and both sequential, parallel and non–deterministic behavior. Behavior executed on state entry and exit can be defined. Hugo/RT allows as well the use of composite and concurrent states and supports pseudo–states like fork and join nodes.

Collaborations specify potential behavior using initial objects and their interactions. Many types of interaction fragments are supported including loop, not, seq, strict, par, alt, opt, ignore and loop. The maximum time between two messages can be specified. Assertions are defined either as LTL or CTL expressions without next operator. Like in EOCL, CTL and LTL can only be used at the top of an expression, e.g., it cannot be used inside the ternary operator.

The tool accepts either the proprietary UTE format as input or ArgoUML files, as it does not provide any GUI integration itself. It then transforms the input model for the use of multiple model checkers like UPPAAL, SPIN, KIV etc.

Comparison

As a first note, there are some threats to a fair evaluation regarding CheckVML, SOCLe and Hugo/RT. The first two tools could not be evaluated as they are not publicly

	Groove	CheckVML	Henshin	SOCLe	Hugo/RT
Entry barrier	Medium	Medium?	High	Medium?	High
Visual Editor	Yes	No	Partially[a]	External	External
Model	Graphs	Graphs	Ecore	UML	UML
Property	CTL, LTL	LTL	mCRL2, OCL inv., PRISM, CADP	EOCL	CTL, LTL, Model consistency
Tool output	Boolean	SPIN outp.	Different[b]	Eval. tree	SPIN outp.
Platform	Java	Java, SPIN	Java, MCs	?, ASM	J., MCs
Extensibility	Medium[c]	—	High	—	Low[d]

Table 5: Comparison of selected tools

[a] Model generation is not integrated with verification, but both are integrated into Eclipse
[b] Counterexamples are supported for OCL invariants and CADP
[c] No API, but open source java
[d] Not open source

available any more. Hugo/RT has a rather high entry barrier and the available documentation does not help to fully understand the usage of the tool. Thus, some evaluation criteria are assessed based on literature. An overview of the tool comparison is given in Table 5.

Entry Barrier SOCLe allows the user to model using UML and to specify the properties in an extension of OCL, thus the entry barrier should be low. CheckVML is able to handle any kinds of models as long as their metamodel is specified in a certain way, so it should be possible to use UML or UML–like models, thus the entry barrier is also assumed to be low. While UML is common to software modelers, the documentation of Hugo/RT is lacking. The description of the UTE language format on their homepage[1] seems to be outdated making it difficult to write the models in their format; also there is no description on how to write ArgoUML models Hugo/RT accepts as input. Groove needs some training to be used efficiently, but there is a nice documentation [60] available.

Visual Editor While models are represented visually in Groove, they quite strongly differ from models used in software modeling. Sometimes the error output when using wrong syntax for specifying nodes or edges is not helpful, e.g., if `del:string:"eating" new:string:"eating"` is used instead of `del:"eating" new:"eating"` the error output includes `Conflicting node aspects`

[1] `http://www.pst.informatik.uni-muenchen.de/projekte/hugo/`. Note that there is a newer Hugo/RT version on `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/sse/hugort/usage/` available, but without description of the UTE format

`del: and new:` which might lead to the wrong conclusion that `del:` and `new:` cannot be used in the same node. If the expression is written on one instead of two lines, the error output is `Unable to parse expression "thinking" new:"hungry" as regular expression`, again not catching the real problem. This might be partially pushed by the huge amount of possible constructs for specifying nodes, edges and graph transformations. Syntax errors in CTL or LTL formulas are usually helpful for correcting the formula. Henshin's visual representation of graph transformations matches the user's expectations, but there are some bugs and non–intuitive behavior, e.g., for specifying multiple NACs and their definition of injective matching which could increase the entry barrier for modelers. It should also be considered that graph transformations are not typically used by modelers. GROOVE allows an import of Ecore models while Henshin allows to use Ecore models directly. Our experiments showed that GROOVE was easier to use than Henshin. Hugo/RT was not usable at all.

Model While SOCLe and Hugo/RT are restricted to UML models, GROOVE, CheckVML and Henshin allow to define custom modeling languages based on their integrated metamodel. GROOVE supports typed attributed graphs, CheckVML uses the terms classes, associations and attributes and Henshin is based on Ecore.

Property specification The properties are typically specified in some sort of temporal logic. GROOVE uses CTL or LTL and properties specified using graph transformations. The graph rule operations are powerful, but some kinds of contstraints are hard to specify, for example constraints using sets. Some languages supported by Henshin are more powerful than CTL or LTL. OCL constraints can be integrated in the expression in PRISM, albeit in a non–intuitive way. mCRL2 (see Section 4.1) can be used for checking paths of parameterized actions and is the only logic able to propagate variables from one state to another. In SOCLe, EOCL (see section 4.1) is used. OCL constraints might be familiar to software modelers, but in EOCL, the concepts of model checking have to be understood as well. In Hugo/RT, CTL and LTL without next operator is supported for assertions. Sets seem to be not usable, but traversal of associations should be possible. Henshin together with mCRL2 clearly has the most powerful language available, but it is also hard to use. EOCL seems to be a better compromise between expressibility and usability because it is the only language with direct integration of OCL and model checking.

Tool output The tool output seems to be a problem for most tools. GROOVE presents the full state space to the user. When an expression is evaluated, only the evaluation result (whether it is true or false) as shown in Figure 21b is returned, but there is no hint on why the evaluation result is the way it is. Thus, the tool is helpful in verifying that the software model fulfills some standard but does not help fixing the error. Henshin provides some information about the result dependent on the language used. The states where an OCL invariant fails to hold are shown. CADP, a language similar to mCRL2 in the sense that it also uses the μ–calculus provides counterexamples. mCRL2 itself,

however, does not provide counterexamples. It is difficult to judge about SOCLe's tool output from the papers alone. It might be very helpful with clickable states in the execution diagram where states correspond to state chart execution steps, but not that helpful, if the execution diagram results cannot be easily mapped to the state chart. The tool output of Hugo/RT seems to be hard to understand. The evaluation result is printed, but there is little information available to interpret the result, especially when locating the error source. However, it should be possible to retranslate trails from UPPAAL or the SPIN model checker to UML runs. Beside the tool output itself, the time for generating the output also has to be considered. Since Hugo/RT transforms to SPIN and SPIN is fast, Hugo/RT should be fairly fast as well. In a simple example, the transformation time for Hugo/RT to SPIN exceeded the time needed for SPIN checking the model. Still, it can be assumed that for larger models which are more complicated to check, the time SPIN needs for model checking is dominant. A performance evaluation of GROOVE against CheckVML and SPIN was done by Rensink et al. [61] showing that GROOVE has comparable performance to CheckVML and SPIN, its performance could then be considered as quite good.[2] The version of Henshin used in the course of the thesis tends to be somewhat slower than GROOVE, but recent versions include performance improvements. While SOCLe's performance is not directly compared to other tools, it seems to be at least sufficient for a case study [56] where the evaluation took only 50ms.

Platform Most if not all are implemented in Java. GROOVE, CheckVML, Henshin and Hugo/RT are surely implemented in Java while the platform for SOCLe is unknown. CheckVML, Henshin and Hugo/RT use model checkers as backend. SOCLe uses a custom on–the–fly model checker whose semantics is based on ASMs and GROOVE uses a custom model checker as well.

Extensibility The most adaptable tools are GROOVE and Henshin. GROOVE is completely open source as well as Henshin, which additionally offers an easy–to–use API. The CADP model checker used by Henshin is not open source, but mCRL2, PRISM and the OCL invariant check are. Hugo/RT is not open source, but the source code can be obtained by asking the developer. SOCLe and CheckVML seem to be not publicly available and thus cannot be adapted and used for further extensions.

Possible extensions Possible extensions for Henshin include a language which is more simple to use then mCRL2 and CADP yet more expressive than OCL constraints alone and improved tool output. GROOVE could be extended to allow OCL constraints in CTL and LTL formulas since some properties are not easily expressible using graph transfor-

[2]Note, however, that the numbers in this paper are at least partially wrong for the dining philosophers problem. While the exact rules of the GROOVE dining philosopher example were not given, they could be reconstructed by the number of states and transitions; in fact, the row which should display 12 entities in GROOVE displays the result for 11 entities.

mations. The tool output could be improved as well by showing a (counter)example–path if available, which could be given for every LTL formula and at least some CTL formulas.

Decision The decision fell to extend Henshin. SOCLe and CheckVML are not available and thus are no candidate for extension. Hugo/RT was too difficult to understand ruling it out as extension candidate as well. Groove has a fully integrated GUI, but a completely new OCL evaluation engine on the graph structure used by Groove would have to be written. Henshin is less integrated, but the provided OCL engine proved to be easily extensible. While originally it was planned to extend the existing state space visualisation, this turned out to be more difficult than expected, so a web interface was generated.

4.3 Summary

Many model checking approaches use ad–hoc formalizations of models. Different kinds of model checking paradigms have been brought to the modeling world trading off expressibility for complexity. The simplest approach is OCL invariant checking requiring no model checking knowledge, the most expressive approach does not use any OCL features. Many languages have been defined but never implemented into tools, many promising tools are not publicly available. Currently the most usable integrations of model checking in MDE are the least sophisticated ones using graph transformations for state space generation. All approaches able to use UML transform the used model into representations suitable for other model checkers. The most relevant issue seems to be helpful output. Henshin has been selected as most suitable candidate for extension.

CHAPTER 5

MoCOCL: A Framework for Model Checking OCL

In this chapter, MoCOCL, a framework for OCL model checking, is presented. At first a rationale behind the design is given, then the temporal OCL language extensions are described in terms of formal syntax and semantics. Finally, an overview of the actual implementation of our model checker is given and the visualization of the result returned by the model checker implementation is discussed.

5.1 Design Rationale

The main requirement for the language and tool integrating MDE and model checking is usability in terms of property specification and the presentation of the result returned by the model checker. Performance issues were not considered when designing the framework. Existing knowledge should be reusable and the approach should be suitable for software models.

Many approaches, as described in Chapter 4, regard OCL as suitable base for a language extension with temporal operators because OCL is likely to be known by modelers. Thus, the approach presented in this chapter extends Essential OCL as well. Pure Essential OCL formulas are valid formulas in the presented language. This allows to separate learning a new language for defining constraints on models and learning a new environment for verification.

We chose CTL over other temporal logics because it is both understandable and easy to implement. The μ–calculus is thought to be hard to understand [9], so it was not regarded as a candidate. While LTL has the advantage of giving a single linear counterexample, which can be reported back easily, its operators would spread throughout the OCL formula making it either necessary to allow LTL only on the top level and OCL on the bottom level or to quite strongly modify existing OCL engines like the Eclipse

OCL Engine.[1] CTL, as discussed in Chapter 3, on the other hand, is easy to implement with no great hassles occurring when mixing temporal operators with OCL constructs. The second OCL extension allows to specify an OCL expression evaluated in certain states. It was tried out what can be implemented easily and straightforwardly and what can be expressed with it.

A clear requirement of our tool is that the modeler should be able to see a counterexample which helps debugging if a property is violated. Thus, the used search algorithm must concentrate on the relevant model parts. Relevant parts are parts of the model affecting the evaluation result. Thus, all relevant information is calculated in the formula evaluation. The output should also be integrated into the usual modeler's workflow. Thus, it was aimed for an integration into Eclipse. The graph transformation framework Henshin (see Chapter 4) provides a state space visualisation, so the first idea was to just extend this visualisation. Unfortunately, there were problems doing this, so an own visualization using web–technology was implemented. This has the advantage of being easily accessible from other computers.

5.2 OCL Semantics

The syntax and semantics defined for this framework are an extension of the formal semantics of OCL [36, 62] with only slight adaptions needed for the existing definitions. At first, a formal definition of Essential OCL syntax and semantics from Richter and Gogolla [62] is reproduced which serves as basis for our OCL extension.

The object model M used in OCL contains classes, attributes, associations, operations and a subtype hierarchy. The set N denotes the set of names.

Definition 22 (Object model [62])**.** The structure of an object model is defined as $M = (\text{CLASS}, \text{ATT}_C, \text{OP}_C, \text{ASSOC}, \mathsf{associates}, \mathsf{roles}, \mathsf{multiplicities}, \prec)$.

- CLASS is a set of class names
- ATT_C is a set of operation signatures for functions mapping an object of class c to an associated attribute value
- OP_C is a set of signatures for user–defined operations of a class c without side effects
- ASSOC is a set of association names
  - assoicates is a function mapping each association name to a list of participating classes
  - roles is a function assigning each end of an association a role name
  - multiplicities is a function assigning each end of an association a multiplicity specification
- $\prec$ is a partial order on CLASS reflecting the generalization hierarchy of classes

[1] `http://wiki.eclipse.org/MDT/OCL/Plugins_and_Features`

□

Example 29 (Object model). An object model for a slightly adapted dining philosophers problem could be specified as follows:
$M = (\text{CLASS}, \text{ATT}_C, \text{ASSOC}, \mathsf{associates}, \mathsf{roles}, \mathsf{multiplicities}, \prec)$.

- $\text{CLASS} = \{\text{Philosopher}, \text{Fork}, \text{Plate}, \text{Table}, \text{Status}, \text{Thinking}, \text{Hungry}, \text{Eating}, \text{String}\}$
- $\text{ATT}_C = \{\text{status} : \text{Philosopher} \to \text{Status}\}$
- $\text{OP}_C = \{\}$
- $\text{ASSOC} = \{\text{phils}, \text{forks}, \text{plates}, \text{leftF}, \text{rightF}, \text{left}, \text{right}\}$
- $\mathsf{associates} = \{\text{phils} \mapsto \langle \text{Philosopher}, \text{Table}\rangle, \text{forks} \mapsto \langle \text{Fork}, \text{Table}\rangle, \text{plates} \mapsto \langle \text{Plate}, \text{Table}\rangle, \text{leftF} \mapsto \langle \text{Plate}, \text{Fork}\rangle, \text{rightF} \mapsto \langle \text{Plate}, \text{Fork}\rangle, \text{left} \mapsto \langle \text{Philosopher}, \text{Fork}\rangle, \text{right} \mapsto \langle \text{Philosopher}, \text{Fork}\rangle\}$
- $\mathsf{roles} = \{\text{phils} \mapsto \langle \text{phil}, \text{tabl}\rangle, \text{forks} \mapsto \langle \text{forks}, \text{tabl2}\rangle, \text{plates} \mapsto \langle \text{plates}, \text{tabl3}\rangle, \text{leftF} \mapsto \langle \text{plate}, \text{leftfork}\rangle, \text{rightF} \mapsto \langle \text{plate2}, \text{rightfork}\rangle, \text{left} \mapsto \langle \text{lfphil}, \text{left}\rangle, \text{right} \mapsto \langle \text{rfphil}, \text{right}\rangle\}$
- $\mathsf{multiplicities} = \{\text{phils} \mapsto \langle \mathbb{N}^+, \{1\}\rangle, \text{forks} \mapsto \langle \mathbb{N}_0, \{1\}\rangle, \text{plates} \mapsto \langle \mathbb{N}^+, \{1\}\rangle, \text{leftF} \mapsto \langle \{1\}, \{1\}\rangle, \text{rightF} \mapsto \langle \{1\}, \{1\}\rangle, \text{left} \mapsto \langle \{1\}, \{0, 1\}\rangle, \text{right} \mapsto \langle \{1\}, \{0, 1\}\rangle\}$
- $\prec = \{(\text{Thinking}, \text{Status}), (\text{Hungry}, \text{Status}), (\text{Eating}, \text{Status})\}$

For building OCL expressions, a data signature is defined over object models. It includes types, their subtype relationship and methods for accessing attributes and navigation.

Definition 23 (Data signature [62])**.** A data signature $\Sigma_M = (T_M, \leqslant, \Omega_M)$ for an object model consists of a set of types T_M, a relation $\leqslant$ on types reflecting the type hierarchy and a set of operations Ω_M. □

Example 30 (Signature). A data signature $\Sigma_M = (T_M, \leqslant, \Omega_M)$ for the dining philosophers problem is defined with the type set $T_M = \{\mathit{Philosopher}, \mathit{Fork}, \mathit{Table}, \mathit{Plate}, \mathit{Thinking}, \mathit{Hungry}, \mathit{Eating}, \mathit{Status}, \mathit{String}\}$ consisting of the primitive type `String` and the object types `Philosopher`, `Fork`, `Table` and `Plate` (see Figure 1 in Section 1.2). The philosopher's status is a `Status`, which can be either `Thinking`, `Hungry` or `Eating`. Thus, the subtype relation is $\leqslant = \{(\mathit{Thinking}, \mathit{Status}), (\mathit{Hungry}, \mathit{Status}), (\mathit{Eating}, \mathit{Status})\}$. The operation set $\Omega_M = \{\text{status} : \mathit{Philosopher} \to \mathit{Status}, \text{phils} : \mathit{Table} \to \mathit{Set}(\mathit{Philosopher}), \ldots\}$ includes an operation for accessing the `Status` of a `Philosopher`, an operation for retrieving all `Philosophers` of a table and some more.

This data signature is used to define the syntax of OCL expressions. The set of free variables of an expression denotes the variables that are assigned a value during the evaluation of the OCL expression. Collection operations are denoted using an arrow (`->`) instead of the usual dot (`.`).

Definition 24 (OCL Syntax [62])**.** The syntax of base OCL expressions is defined as follows:

1. If $v \in \text{Var}_t$, then $v \in \text{Expr}_t$ and $\text{free}(v) := \{v\}$.
2. If $v \in \text{Var}_t, e_1 \in \text{Expr}_{t_1}, e_2 \in \text{Expr}_{t_2}$ then `let` $v = e_1$ `in` $e_2 \in \text{Expr}_{t_2}$ and $\text{free}(\text{let } v = e_1 \text{ in } e_2) := \text{free}(e_2) - \{v\}$.
3. If $\omega : t_1 \times \ldots \times t_n \to t \in \Omega_M$ and $e_i \in \text{Expr}_{t_i}$ for all $i = 1, \ldots, n$ then $\omega(e_1, \ldots, e_n) \in \text{Expr}_t$ and $\text{free}(\omega(e_1, \ldots, e_n)) := \text{free}(e_1) \cup \ldots \cup \text{free}(e_n)$.
4. If $e_1 \in \text{Expr}_{Boolean}$ and $e_2, e_3 \in \text{Expr}_t$ then `if` e_1 `then` e_2 `else` e_3 `endif` $\in \text{Expr}_t$ and $\text{free}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif}) := \text{free}(e_1) \cup \text{free}(e_2) \cup \text{free}(e_3)$.
5. If $e \in \text{Expr}_t$ and $t' \leqslant t$ or $t \leqslant t'$ then `(`e `asType` t'`)` $\in \text{Expr}_{t'}$, `(`e `isTypeOf` t'`)` $\in \text{Expr}_{\text{Boolean}}$, `(`$e$ `isKindOf` t'`)` $\in \text{Expr}_{\text{Boolean}}$ and $\text{free}((e \text{ asType } t')) := \text{free}(e), \text{free}((e \text{ isTypeOf } t')) := \text{free}(e)$, $\text{free}((e \text{ isKindOf } t')) := \text{free}(e)$.
6. If $e_1 \in \text{Expr}_{Collection(t_1)}, v_1 \in \text{Var}_{t_1}, v_2 \in \text{Var}_{t_2}$, and $e_2, e_3 \in \text{Expr}_{t_2}$ then $e_1 \to$ `iterate(`v_1`;` $v_2 = e_2$ `|` e_3`)` $\in \text{Expr}_{t_2}$ and $\text{free}(e_1 \to \text{iterate}(v_1; v_2 = e_2 | e_3)) := (\text{free}(e_1) \cup \text{free}(e_2) \cup \text{free}(e_3)) - \{v_1, v_2\}$.

□

A state $\sigma \in \mathcal{S}$ is used to retrieve object, associations and attributes of an object model M via the projection functions $\sigma|_{\text{CLASS}}$, $\sigma|_{\text{ATT}}$, and $\sigma|_{\text{ASSOC}}$ which retrieve all objects, attributes and associations for a system state [62], repsectively.

Definition 25 (System state [67])**.** The structure of a system state over an object model M is defined as $\sigma(M) = (\sigma|_{\text{CLASS}}, \sigma|_{\text{ATT}}, \sigma|_{\text{ASSOC}})$:

- $\sigma|_{\text{CLASS}}(c)$ contains all objects o of the class c.
- $\sigma|_{\text{ATT},a}$ returns a function for the value of attribute a of class c. $o_{\text{ATT},a}(o)$ retrieves the value of the attribute a for the object o.
- $\sigma|_{\text{ASSOC}}(as)$ assigns a list of participating objects $\langle o_1, ..., o_n \rangle$ to each association as.

□

A variable assignment is a function $\beta : Var_t \to Val_t$ that, given a variable name, returns the current value of the associated variable, where t is the type of the associated variable. An environment $\tau = (\sigma, \beta)$ consists of a system state σ and a variable assignment β.
Now, we have all components for defining the semantics of OCL.

Definition 26 (OCL semantics [62])**.** Let Env be the set of environments $\tau = (\sigma, \beta)$. The semantics of an expression $e \in \text{Expr}_t$ is a function $I[\![\ e\ ]\!] : \text{Env} \to I(t)$ that is defined as follows.

i. $I[\![v]\!](\tau) = \beta(v)$

ii. $I[\![\text{let } v = e_1 \text{ in } e_2]\!](\tau) = I[\![e_2]\!](\sigma, \beta\{v/I[\![e_1]\!](\tau)\})$.

iii. $I[\![\omega(e_1, \ldots, e_n)]\!](\tau) = I(\omega)(\tau)(I[\![e_1]\!](\tau), \ldots, I[\![e_n]\!](\tau))$.

iv. $I[\![\text{ if } e_1 \text{ then } e_2 \text{ else e}_3 \text{ endif}]\!](\tau) = \begin{cases} I[\![e_2]\!](\tau) & \text{if} I[\![e_1]\!](\tau) = \mathit{true} \\ I[\![e_3]\!](\tau) & \text{if} I[\![e_1]\!](\tau) = \mathit{false} \\ \bot & \text{otherwise} \end{cases}$

v. $I[\![(e \text{ asType } t')]\!](\tau) = \begin{cases} I[\![e]\!](\tau) & \text{if} I[\![e]\!](\tau) \in I(t'), \\ \bot & \text{otherwise.} \end{cases}$

$I[\![(e \text{ isTypeOf } t')]\!](\tau) = \begin{cases} \mathit{true} & \text{if} I[\![e]\!](\tau) \in I(t') - \cup_{t'' < t'} I(t''), \\ \mathit{false} & \text{otherwise} \end{cases}$

$I[\![(e \text{ isKindOf } t')]\!](\tau) = \begin{cases} \mathit{true} & \text{if} I[\![e]\!](\tau) \in I(t'), \\ \mathit{false} & \text{otherwise} \end{cases}$

vi. $I[\![e_1 \to \text{iterate}(v_1; v_2 = e_2 | e_3)]\!](\tau) = I[\![e_1 \to \text{iterate}'(v_1 | e_3)]\!](\tau')$ where $\tau' = (\sigma, \beta')$ and $\tau'' = (\sigma, \beta'')$ are environments with modified variable assignments: $\beta' := \beta\{v_2/I[\![e_2]\!](\tau)\}$, $\beta'' := \beta'\{v_2/I[\![e_3]\!](\sigma, \beta'\{v_1/x_1\})\}$ □

5.3 CTL Extension of OCL

CTL, as described in Section 3.4, is a temporal logic using branching–time semantics and provides the temporal concepts which we will integrate in OCL. In this thesis, this language extension of OCL is called cOCL.

Definition 27 (Syntax of cOCL)**.** Let $\Sigma_M = (T_M, \leqslant, \Sigma_M)$ be a data signature over an object model M, $\text{Var} = \{\text{Var}_t\}_{t \in T_M}$ be a family of variable sets where each variable set is indexed by a type t. The syntax over the signature Σ_M is given by a set $\text{Expr} = \text{Expr}_{t \in T_M}$ of state expressions, a temporary set PExpr of path expressions and a function $\text{free} : \text{Expr} \to \mathcal{F}(\text{Var})$ denoting the set of free variables. It is defined as follows:

1. Each OCL expression of Definition 24 is in cOCL;

2. if $\phi \in \text{Expr}_{\text{Boolean}}$ then A X ϕ, E X ϕ, A G ϕ, E G ϕ, A F ϕ, E F ϕ in cOCL, where $\text{Expr}_{\text{Boolean}}$ are expressions of type Boolean. Let e denote this expression, then $\text{free}(e) = \text{free}(\phi)$.

3. if $\phi, \psi \in \text{Expr}_{\text{Boolean}}$ then A ϕ W ψ, E ϕ W ψ, A ϕ U ψ, E ϕ U $\psi \in \text{Expr}_{\text{Boolean}}$ in cOCL, where $\text{Expr}_{\text{Boolean}}$ are expressions of type Boolean. Let e denote this expression, then $\text{free}(e) = \text{free}(\phi) \cup \text{free}(\psi)$. □

Because CTL is used, A and E must be directly be followed by some X, G, F, W or U. Path formulas alone are not valid cOCL expressions. To enhance readability, the concrete syntax of cOCL uses the expanded form of the operators. The operator A is written as `Always`, E as `Exists`, X as `Next`, G as `Globally`, F as `Finally`, W as `Unless` and U as `Until`.

Example 31 (Syntactically valid and invalid cOCL formulas). This example illustrates some syntactically valid and invalid formulas.

```
let s = self.philosophers->size() in Always Globally self.philosophers->size()
= s
```

A syntactically valid cOCL formula with the intention to specify that the number of philosophers should always remain constant.

```
Always Globally Exists Next true
```

A syntactically valid cOCL formula with the intention to specify that there should be no deadlock.

```
Always self = null or Finally self.philosophers->size() > 0
```

A syntactically invalid cOCL formula with the intention to specify that unless there is no root object in the model, there will be a philosopher sometimes in the future. This expression is invalid because `Always` is not followed by a temporal operator and `Finally` is not preceded by `Always` or `Exists`. This would be a formula valid in an LTL extension of OCL.

To define the semantics of cOCL, we first define the term state space. An approach for generating such a state space is discussed in Section 5.5.

Definition 28 (State space)**.** The state space $\mathcal{K}_M = (\mathcal{S}, \iota, \mathcal{T}, \mathcal{B}, \mathcal{E})$ of a model M consists of a set of states $\mathcal{S}$, a single initial state $\iota \in \mathcal{S}$, a transition relation $T \subseteq \mathcal{E} \times \mathcal{E}$, a set of variable assignments $\mathcal{B}$, and the environment relation $\mathcal{E} \subseteq \mathcal{S} \times \mathcal{B}$. An environment $\tau \in \mathcal{E}$ is a pair (σ, β) with state $\sigma \in \mathcal{S}$ and variable assignment $\beta \in \mathcal{B}$. □

Due to the richness of OCL, its state space is more complex than typical statespaces used by CTL model checking. Typically, the transition relation connects states, not environments. An equivalent version of a typical CTL model checking state space compliant with the state space generation described in the implementation section would consist of just a single object containing boolean attributes equivalent to the atomic propositions of that state. The transition relation collapses to the relation transforming the single object of the current state into the single object of the next state and thus equals the state transition relation found usually in CTL model checking.

Example 32 (State space). Consider the state space $\mathcal{K}_M = (\mathcal{S}, \iota\mathcal{T}, \mathcal{B}, \mathcal{E})$ of the simple dining philosophers problem with one philosopher. Assume the data signature Σ_M to be the data signature defined in Example 30. It contains the states $\mathcal{S} = (\sigma_{thinking}, \sigma_{hungrynofork}, \sigma_{hungryleft})$. These states correspond to the states where the philosopher is thinking, is hungry, but has no fork and the state where he/she is hungry and has a left fork in his/her hands. The thinking state $\sigma_{thinking}|_{\text{CLASS}} = \{p_1, f_1, d_1, t_1, think\}$ has five objects: A philosopher p_1, a fork f_1 a plate d_1, a table t_1 and a status object *think*. The only attribute `status` associates p_1 with *think*. $\sigma_{thinking}|_{\text{ATT}} = \{\texttt{status} : p_1 \mapsto think\}$. There are associations between table and forks, plates and philosophers. Additionally, there are two associations from the plate to the fork. $\sigma_{thinking}|_{\text{ASSOC}} = \{\text{phils} \mapsto \{\langle p_1, t_1\rangle\}, \text{forks} \mapsto \{\langle f_1, t_1\rangle\}, \text{plates} \mapsto \{\langle d_1, t_1\rangle\}, \text{leftF} \mapsto \{\langle d_1, f_1\rangle\}, \text{rightF} \mapsto \{\langle d_1, f_1\rangle\}, \text{left} \mapsto \{\}, \text{right} \mapsto \{\}\}$. The other states have similar objects and associations with the only exception that the status of p_2 (resp. p_3) refers to $hungry_2$ (resp. $hungry_3$) and, in the last state, there is an association between philosopher and fork instead of table and fork. $\sigma_{hungrynofork}|_{\text{CLASS}} = \{p_2, f_2, d_2, t_2, hungry_2\}$ and $\sigma_{hungryleftfork}|_{\text{CLASS}} = \{p_3, f_3, d_3, t_3, hungry_3\}$. Note that there is no state σ_{eating}, since it is not possible to eat with only one fork. The initial state ι is $\sigma_{thinking}$. Assume the only variable possible is p of type *Philosopher* corresponding to the philosopher or `null`. then, $\mathcal{B} = \{b_0 : p \mapsto \texttt{null}, b_1 : p \mapsto p_1, b_2 : p \mapsto p_2, b_3 : p \mapsto p_3\}$. There are six possible environments. $\mathcal{E} = \{\epsilon_0 = (\sigma_{thinking}, b_0), \epsilon_1 = (\sigma_{thinking}, b_1), \epsilon_2 = (\sigma_{hungrynofork}, b_0), \epsilon_3 = (\sigma_{hungrynofork}, b_2), \epsilon_4 = (\sigma_{hungryleftfork}, b_0), \epsilon_5 = (\sigma_{hungryleftfork}, b_3)\}$. Other environments are not possible because we restrict the environments of a specific state to only allow variable assignments assigning to objects of the same state. The transition relation is given by $\mathcal{T} = \{\tau_0 = (\epsilon_0, \epsilon_2), \tau_1 = (\epsilon_2, \epsilon_4), \tau_2 = (\epsilon_1, \epsilon_3), \tau_3 = (\epsilon_3, \epsilon_5)\}$.

The notation of a state space introduced above allows us to define paths on the state space.

Definition 29 (Path)**.** Let $\mathcal{K}_M = (\mathcal{S}, \iota, \mathcal{T}, \mathcal{B}, \mathcal{E})$ be the state space of a model M. A *path* π is a finite or infinite sequence of environments $(\tau_1, \tau_2, \ldots)$ with $\tau_i \in \mathcal{E}$ such that $(\tau_i, \tau_{i+1}) \in \mathcal{T}$. For a path $\pi = (\tau_1, \tau_2, \ldots)$, we define the projection function $\pi(i) = \tau_i$ and $\pi^i = (\tau_i, \tau_{i+1}, \ldots)$. The length of a path $|\pi| = n$ for finite paths $\pi = (\tau_1, \ldots, \tau_n)$, and $|\pi| = \infty$ for infinite paths $\pi = (\tau_1, \tau_2, \ldots)$. A path π is called *maxpath*, if $|\pi| = \infty$ or $\nexists \tilde{\tau} : (\pi(|\pi|), \tilde{\tau}) \in \mathcal{T}$ □

Example 33 (Path). Consider the state space of Example 32. There are only four possible paths starting in the initial state in this statespace. These paths are $\pi_1 = (\tau_0, \tau_1)$ moving from the state thinking to hungry with a left fork where the variable is assigned to `null`, $\pi_2 = (\tau_2, \tau_3)$ doing the same with the variable p being assigned to the philosopher, $\pi_3 = (\tau_0)$ and $\pi_4 = (\tau_2)$ moving from state thinking to hungry with no fork. Path π_1 and π_2 are maxpaths. There are no paths of infinite length because the state space does not include a cycle in $\mathcal{T}$.

Definition 30 (Semantics of cOCL)**.** Let $\mathcal{K}_M$ be a state space of model M. The semantics of a cOCL expression is defined by the rules i.–vi. of Definition 2 from [62] plus the following rules for the temporal extension.

vii. $I[\![\mathsf{A}\,\phi\,\mathsf{U}\,\psi]\!](\tau) = \text{true} \Leftrightarrow \forall\,\text{paths}\ \pi \text{ with } \pi(0) = \tau : \exists n \in \mathbb{N}, n \leqslant |\pi| : I[\![\psi]\!](\pi(n)) = \text{true} \wedge \forall\, 0 \leqslant i < n : I[\![\phi]\!](\tau_i) = \text{true}$

viii. $I[\![\mathsf{E}\,\phi\,\mathsf{U}\,\psi]\!](\tau) = \text{true} \Leftrightarrow \exists\,\text{path}\ \pi \text{ with } \pi(0) = \tau : \exists n \in \mathbb{N}, n \leqslant |\pi| : I[\![\psi]\!](\pi(n)) = \text{true} \wedge \forall\, 0 \leqslant i < n : I[\![\phi]\!](\tau_i) = \text{true}$

ix. $I[\![\mathsf{A}\,\phi\,\mathsf{W}\,\psi]\!](\tau) = \text{true} \Leftrightarrow \forall\,\text{paths}\ \pi \text{ with } \pi(0) = \tau : \forall n \in \mathbb{N}, n \leqslant |\pi| : I[\![\phi]\!](\pi(n)) = \text{false} \rightarrow \exists i \in \mathbb{N}, i \leqslant n : I[\![\psi]\!](\pi(i)) = \text{true}$

x. $I[\![\mathsf{E}\,\phi\,\mathsf{W}\,\psi]\!](\tau) = \text{true} \Leftrightarrow \exists\,\text{path}\ \pi \text{ with } \pi(0) = \tau : \forall n \in \mathbb{N}, n \leqslant |\pi| : I[\![\phi]\!](\pi(n)) = \text{false} \rightarrow \exists i \in \mathbb{N}, i \leqslant n : I[\![\psi]\!](\pi(i)) = \text{true}$

xi. $I[\![\mathsf{E}\,\mathsf{X}\,\phi]\!](\tau) = \text{true} \Leftrightarrow \exists\,\text{path}\ \pi \text{ with } \pi(0) = \tau, |\pi| \geqslant 1 : I[\![\phi]\!](\pi(1)) = \text{true}$

xii. $I[\![\mathsf{A}\,\mathsf{X}\,\phi]\!](\tau) = \text{true} \Leftrightarrow \forall\,\text{paths}\ \pi \text{ with } \pi(0) = \tau, |\pi| \geqslant 1 : I[\![\phi]\!](\pi(1)) = \text{true}$

□

The expression $\perp$ refers to the invalid value. The intuition of the path formulas E and A is that E is true, if there is a path starting from the current state where the remaining formula is true, otherwise false. Likewise, A is true if the remaining formula is true for all paths starting at the current state. The X operator evaluates the following subexpression in the next state.

The expression ϕ U ψ means that when considering a single path, ψ must hold at some state, the n–th state in the path, in the future and before that (excluding the state where ψ holds), ϕ has to hold. If one only wants to ensure ψ holding somewhere in the future, ϕ can be replaced with *true*. This is abbreviated as F ψ.

The expression ϕ W ψ is true if whenever ϕ evaluates to false, ψ must have held at some previous point. In other words, as soon as ψ holds, ϕ needs not to hold any longer, but before that, ϕ must evaluate to true. The difference to U lies in that ψ is not required to hold eventually. If one just wants that ϕ holds forever, one could express this with ϕ W *false*. This is abbreviated as G ϕ.

Example 34 (Semantics of cOCL). Now let us consider the applications of this semantics to a simple starvation check on the statespace depicted in Figure 24 with the environment transformation rule that the first/second philosopher always stays the first/second philosopher in the according state. The expression `philosophers->exists(p | Exists Finally Exists Globally p.status = PState::hungry)` checks the possibility that a philosopher may at some point start getting hungry but stays hungry all the time, i.e., is not able to eat anything in the future. The initial state is the top left state with both philosophers thinking.

For evaluating the `exists` collection operation, we need to check whether for any element in the philosophers set $\{P1,P2\}$, the body condition is true. We start with

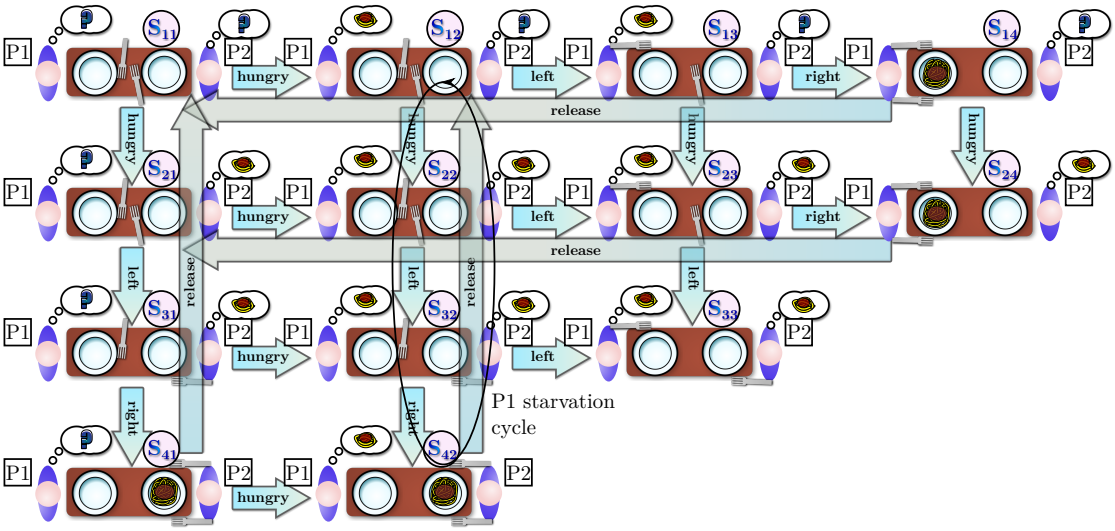


Figure 24: Full state space of the dining philosopher problem with 2 philosophers

$P1$ and now check the condition that `Exists Finally Exists Globally p.status = PState::hungry`. For the `exists` collection operation to hold, we need a path starting at the current state where the remaining formula holds. This is, we need a path, where somewhere in the future `Exists Globally p.status = PState::hungry` holds. Thus, we might start looking for a path where $P1$'s status is always hungry. We can find a cycle leading to the path in column 2: $(S_{12}, S_{22}, S_{32}, S_{42})$ gives a cycle where $P1$ is always hungry. Thus, when being in any of these states, we can find an infinite path where $P1$ is always hungry by traversing this cycle. Now we need to find a path leading to some of these states. S_{12} can be reached by the hungry transition for the first philosopher. Thus, we have found a philosopher with a path leading to a possibly infinite sequence of being hungry and the result of this cOCL expression is true.

We define a cOCL expression *satisfiable* as follows.

Definition 31 (Satisfiability)**.** A cOCL expression ϕ is satisfiable w.r.t. a state space $\mathcal{K}_M$ iff $I[\![\phi]\!](\iota)$ is *true* w.r.t. $\mathcal{K}_M$. □

Example 35 (Satisfiability). The expression of Example 34 is satisfiable.

5.4 OCL Extension Using Selectors

While CTL expressions allow to verify behavioral properties, they can only give information about whether the property is fulfilled or not. Still, often in modeling, one wants to know some information about the model. Thus, the selectors have been developed allowing the evaluation of OCL expressions in a specified set of states. More specifically,

selectors evaluate expressions on the last state of matching paths. For example, it is possible to retrieve all deadlock states and look at certain properties of them.

Definition 32 (Syntax of selector expressions)**.** Let $\Sigma_M = (T_M, \leqslant, \Sigma_m)$ be a data signature (see Definition 29). The syntax for selector expressions is given by:

i. If $e \in \text{Expr}_t$ and $r \in \text{RExpr}$ then $\texttt{(}e\texttt{)@(}r\texttt{)} \in \text{Expr}_t$

□

Definition 33 (Syntax of range expressions)**.** The syntax for range expressions *RExpr* is as follows:

i. If $e \in \text{Expr}_{Boolean}$, then while e, where e, before e and from $e \in \text{RExpr}$.

ii. If $e_b, e_u \in \text{Expr}_{Int}, e_h \in \text{Expr}_{Boolean}$ then next, next between e_b upto e_u, next e_b, next having e_h, next e_b having e_h, next between e_b upto e_u having $e_h \in \text{RExpr}$

iii. If $e_h \in \text{Expr}_{Boolean}$ then current, current having $e_h \in \text{RExpr}$.

iv. If $e_c, e_h \in \text{Expr}_{Boolean}$ then range e_c upto $e_h \in \text{RExpr}$.

v. If $r_1, r_2 \in \text{RExpr}$ then (r_1) with r_2, (r_1) intersect r_2, (r_1) without r_2 and (r_1) then $r_2 \in \text{RExpr}$.

□

For simplicity, the abstract syntax exactly matches the concrete syntax for the selector expressions.

The intuition of the semantics is that the result of a selector expression $\texttt{(}e\texttt{)@(}r\texttt{)}$ is the value of e evaluated in the last states of all paths selected by r. All range expressions themselves select a set of paths. The intention of the expression while e is that paths are selected where e is fulfilled in all states of the path while where e selects paths where e is fulfilled in at least the last state of the path. The expression before is the counterpart of while and selects paths where e is not fulfilled in any state of the path. The expression from e selects paths where e is fulfilled in at least one state of the path. The expression range e_b upto e_u combines both operators mentioned before. It selects paths where e_b is fulfilled in some state, and e_u is never fulfilled after the first fulfillment of e_b. The expression current selects the path consisting of the current state, next selects all paths with specified length, possibly meeting some condition. The expressions (r_1) with r_2, (r_1) intersect r_2 and (r_1) without r_2 perform the corresponding set operations on the selected paths. The expression (r_1) then r_2 selects paths matching r_2 starting from any end state of a path selected by r_1.

For defining the semantics, we introduce a helper function $pfs(\tau)$ generating all paths starting from an environment. This function is defined as $pfs(\tau) = \{\pi | \pi \text{ path}, \pi(1) = \tau\}$.

Definition 34 (Selector Semantics)**.** Let $\mathcal{K}_M$ = be a state space of model M. The semantics of a selector expression is defined by the following rules.

i. $I[\![e@r]\!](\tau) = \bigcup_{\pi_i \in I[\![r]\!]} I[\![e]\!](\pi(length(\pi_i))$

ii. $I[\![\textsf{next between } e_1 \textsf{ upto } e_2 \textsf{ having } e_3]\!](\tau) = \{\pi | \pi \in pfs(\tau) : \exists i \in \mathbb{N}, I[\![e_1]\!](\tau) < i = length(\pi) \leqslant I[\![e_2]\!](\tau) + 1 : I[\![e_3]\!](\pi(i)) = true\}$ with $e_3 = true$ if the having clause is not specified.

iii. $I[\![\textsf{next } e_1 \textsf{ having } e_2]\!](\tau) = I[\![\textsf{next between } 1 \textsf{ upto } e_1 \textsf{ having } e_2]\!](\tau)$

iv. $I[\![\textsf{current having } e]\!](\tau) = I[\![\textsf{next between } 0 \textsf{ upto } 0 \textsf{ having } e]\!](\tau)$

v. $I[\![\textsf{while } e]\!](\tau) = \{\pi | \pi \in pfs(\tau), \forall 1 \leqslant j \leqslant length(\pi) : I[\![e]\!](\pi(j)) = true\}$

vi. $I[\![\textsf{where } e]\!](\tau) = \{\pi | \pi \in pfs(\tau), I[\![e]\!](\pi(length(\pi)) = true\}$

vii. $I[\![\textsf{range } e_1 \textsf{ upto } e_2]\!](\tau) = \{\pi | \pi \in pfs(\tau), i = length(\pi), \exists j \in \mathbb{N} : 1 \leqslant j \leqslant i : I[\![e_1]\!](\pi(j)) = true, \forall k \in \mathbb{N} : min(\{j | I[\![e_1]\!](\pi(j)) = true\}) \leqslant k \leqslant i : I[\![e_2]\!](\pi(k)) = false\}$

viii. $I[\![\textsf{before } e]\!](\tau) = I[\![\textsf{range } true \textsf{ upto } e]\!](\tau)$

ix. $I[\![\textsf{from } e]\!](\tau) = I[\![\textsf{range } e \textsf{ upto } false]\!](\tau)$

x. $I[\![r_1 \textsf{ with } r_2]\!](\tau) = I[\![r_1]\!](\tau) \cup I[\![r_2]\!](\tau)$

xi. $I[\![r_1 \textsf{ without } r_2]\!](\tau) = I[\![r_1]\!](\tau) \setminus I[\![r_2]\!](\tau)$

xii. $I[\![r_1 \textsf{ intersect } r_2]\!](\tau) = I[\![r_1]\!](\tau) \cap I[\![r_2]\!](\tau)$

xiii. $I[\![r_1 \textsf{ then } r_2]\!](\tau) = \bigcup_{\pi_i \in I[\![r_1]\!](\tau)} I[\![r_2]\!](\pi_i(length(\pi_i))$

The paths π used here may be finite and not ending in a deadlock, i.e., not maxpaths.□

In the following, we clarify the selectors' syntax and semantics with the help of some examples using the state space shown in Figure 24. Unless otherwise noted, the expressions are evaluated in the State S_{11}.

Current The `current` range selects the path consisting only of the current state. The expression `current having` e selects the current state only if e evaluates to true.

Example 36 (Current). This example illustrates the usage of the current expression.

```
(self.philosophers)@(current having self.philosophers->exists(p | p.left <>
null))
```

If this expression is evaluated in S_{11}, it will return the empty set because there is no philosopher having a fork in its left hand. If it is, however, evaluated in state S_{13}, it will return the set of both philosophers.

Next The `next` range covers paths with certain length as specified using the `between` and `upto`, inclusively, attributes. These paths then are filtered for having a last state fulfilling the `having` attribute.

Example 37 (Next). This example illustrates the usage of the `next` expression.

```
(self.philosophers.status)@(next between 1 upto (self.philosophers->size())
having self.philosophers->exists(p | p.status = PState::hungry and p.left =
null))
```

This expression returns the status of philosophers in the next n states with n being the number of philosophers, but only if there is at least one philosopher hungry who also has no fork. When evaluated in the state S_{11}, the paths (S_{11}, S_{12}) and (S_{11}, S_{21}) of length 2 are returned all having a philosopher hungry who has no fork leading to the result set $\{thinking, hungry\}$ and $\{hungry, thinking\}$. Additionally, the paths (S_{11}, S_{12}, S_{13}) and (S_{11}, S_{21}, S_{31}) are considered, but since the philosophers are either thinking or having a fork in S_{13} or S_{31}, the states are not returned. The paths (S_{11}, S_{12}, S_{22}) and (S_{11}, S_{21}, S_{22}) both have both philosophers being hungry and having no fork, but have the same last state S_{22}; thus the result set includes the element $\{hungry, hungry\}$ for this state only once.

While The `while` range covers all paths where the specified expression is true all the time. In other words, `expr@(while` e_2`)` returns expressions evaluated in a state reachable by a path with the expression e_2 being true all the time.

Example 38 (While). This example illustrates the usage of the `while` expression.

```
self@(while (self.philosophers->exists(p,p2 | p.status = PState::thinking or
(p <> p2 and p.status = p2.status and p.left = p2.left))))
```

This expressions returns the tables of the states where one philosopher is thinking, which are $S_{11}, S_{12}, S_{13}, S_{14}, S_{21}, S_{31}, S_{41}$ and the state where both philosophers are hungry and have no fork, which is S_{22}. The table of the state S_{33} where both philosophers have a fork in the left hand, however, is not returned because there are only paths from S_{32} and S_{23} to this state which both do not fulfill the condition specified in the range.

Where The `where` range covers all paths with the last state fulfilling the expression. In other words, e`@(where` e_2`)` includes the results of e evaluated in all states where e_2 holds.

Example 39 (Where). This example illustrates the usage of the `where` expression.

```
self@(where (self.philosophers->exists(p,p2 | p.status = PState::thinking or
(p <> p2 and p.status = p2.status and p.left = p2.left))))
```

The result of the expression includes the table of all states of the while expression above and additionally the table of the state S_{33} because it matches the condition specified.

Range Upto The expression `range` e_1 `upto` e_2 covers all paths where e_1 evaluates to true for some state in the path but e_2 does not evaluate to true in that path after the first evaluation of e_1 to true.

Example 40 (Range Upto). This example illustrates the usage of the range ... upto expression.

```
self@(range (self.philosophers->exists(p | p.status = PState::hungry))
upto (self.philosophers->forAll(p | p.status = PState::hungry or p.status =
PState::eating)))
```

This expression returns the tables of the states $S_{12}, S_{13}, S_{14}, S_{21}, S_{31}, S_{41}$ and S_{11}. S_{11} is returned because at state S_{14} or S_{41}, after the release action is executed, state S_{11} is reached.

Before The expression `before` e_1 covers all paths where e_1 does never evaluate to true. Thus, e`@(before` e_1`)` includes expressions in states reachable by a path without e_1 evaluating to true.

Example 41 (Before). This example illustrates the usage of the before expression.

```
self@(before self.philosophers->exists(p | p.left <> null))
```

This expression returns the tables of the states S_{11}, S_{21}, S_{12} and S_{22}.

From The expression `from` e covers all paths where e has evaluated to true before.

Example 42 (From). This example illustrates the usage of the from expression.

```
self@(from (self.philosophers->exists(p | p.left <> null)))
```

This expression returns the tables of all states because all states are reachable by a path starting from S_{11} which is reachable by the path S_{11}, S_{12}, S_{13}, S_{14}, S_{11}.

With The expression r_1 `with` r_2 covers all paths which are in at least one of the ranges specified in both expressions.

Example 43 (With). This example illustrates the usage of the with expression.

```
self@((next 1) with (current))
```

This expression returns the tables of the states S_{11}, S_{12} and S_{21}.

Without The expression r_1 `without` r_2 covers all paths which are in the range r_1 but not in r_2.

Example 44 (Without). This example illustrates the usage of the without expression.

```
self@((while true) without (current))
```

This expression returns the tables of all states. This is because only the path (S_{11}) is excluded, but not e.g. the path $(S_{11}, S_{12}, S_{13}, S_{14}, S_{11})$ leading to the same state.

Intersect r_1 `intersect` r_2 covers all paths with are in both the ranges r_1 and r_2.

Example 45 (Intersect). This example illustrates the usage of the intersect expression.

```
self@((while self.philosophers->exists(p | p.status = PState::thinking))
intersect (next 2))
```

This expression returns the tables of the states S_{12}, S_{13}, S_{21} and S_{31}. These are all states having a path of length two or three where there is a path of arbitrary length where in each state at least one philosopher is thinking.

Then The expression r_1 `then` r_2 returns all paths starting with a path in r_1 and continuing then with a path in r_2

Example 46 (Then). This example illustrates the usage of the then expression.

```
self@((next 1) then (before self.philosophers->forAll(p | p.status =
PState::thinking)))
```

This expression returns the tables of all states except S_{11}. Initially, the two states S_{12} and S_{21} are selected by `next 1`. Starting from these states, all states from paths starting from any of these states are returned as long as there are not all philosophers thinking in the path.

5.5 Implementation

We call the implemented framework MocOCL, the short name for *Model Checking OCL*. Its implementation is based on the existing Eclipse OCL Plugin.[2] The basic architecture is shown in Figure 25. It consists of an XText[3] based OCL parser used to convert input strings into the concrete syntax tree. The pivot parser then builds the abstract syntax tree resolving references and types, the pivot evaluator then traverses this tree to evaluate the result providing additional information about the evaluation for easier understanding of what caused the result.

The syntactical augmentation of OCL by CTL operators was relatively straightforward. A new `CTLOperationExp` was introduced and the generated parsers were updated to handle this new operation as well.

[2] `http://wiki.eclipse.org/MDT/OCL/Plugins_and_Features`
[3] `http://www.eclipse.org/Xtext/`

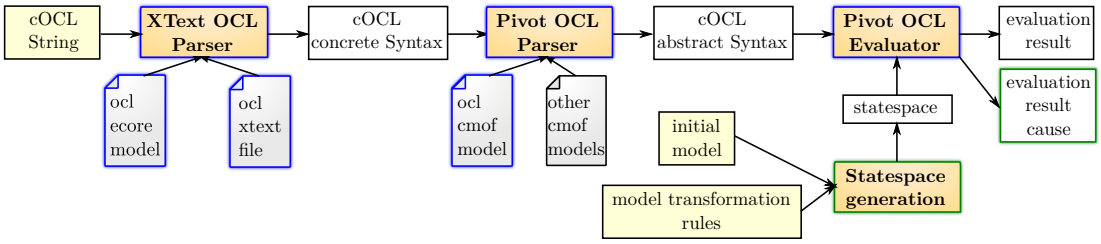


Figure 25: Architecture of the XText OCL plugin. Modified parts are highlighted blue, added parts green

The evaluation of CTL formulas in the pivot evaluator requires the ability to traverse the state space. This ability is provided by the `StateTraverser`. It allows to calculate all next states for a state and provide information about some state. A `StateTraverser` implementation is provided using Henshin. For generating all next states, it executes all applicable rules to the current state graph, checks if the generated state exists already by doing an isomorphism check to existing states and calculates the mapping between the old and the new state. The pivot evaluator uses the state space generator to iteratively retrieve the next states and required evaluation environment transformations.

The type definitions $Sequence(t)$, $Set(t)$, and $Bag(t)$ and the function definitions $mkSequence_t$, $mkSet_t$, and $mkBag_t$ that we use in the following definitions are those introduced by Richters and Gogolla [62]. For example, $I(Sequence(t))$ defines the set of all possible sequences of type t. We define $I(Collection(t)) = (I(Sequence(t)) \cup I(Set(t)) \cup I(Bag(t)))$.

In MocOCL, the state space consists of a set of graphs. Each graph corresponds to an instance of the system and thus represents a system's state at a discrete point in time. Given a graph transformation system $KG = (\mathcal{R}, \iota)$ with graph rewrite rules $\mathcal{R}$ and an initial state ι, the function $\boldsymbol{stategen}_{\mathcal{R}}\colon \mathcal{S} \to P(\mathcal{S} \times \mathcal{M})^4$ handles the generation of the state space. It expects as input a state σ_s and returns a set of pairs (σ_t, m) where σ_t denotes the successor state of σ_s and $m : \sigma_{Class} \to \sigma_{Class} \cup \{\bot\}$ is a morphism that maps objects in σ_s to corresponding objects in σ_t, or to $\bot$ if no such object exists. The successor state σ_t is obtained from σ_s by applying a rewrite rule $r \in \mathcal{R}$ to the graph represented by σ_s. We write $\sigma_s \overset{r}{\Rightarrow} \sigma_t$ to denote that σ_s is rewritten to σ_t by rule $r \in \mathcal{R}$. The state space generation function is then defined as

$$\boldsymbol{stategen}_{\mathcal{R}}(\sigma_s) = \bigcup_{r \in \mathcal{R}} \{(\sigma_t, m) | \sigma_s \overset{r}{\Rightarrow} \sigma_t \wedge \exists m \in \mathcal{M} : \forall c \in \sigma_s|_{\text{CLASS}} : m(c) \in \sigma_t|_{\text{CLASS}} \vee m(c) = \bot\}$$

The helper function $\boldsymbol{succ}\colon \mathcal{E} \to P(\mathcal{E})$ returns all environments reachable by a transition from the source environment $\tau_s = (\sigma_s, \beta_s)$ and is defined by

[4] $P(X)$ is the set of all finite subsets of X.

$$\textbf{\textit{succ}}((\sigma_s, \beta_s)) := \{(\sigma_t, \beta_t) | (\sigma_t, m) \in \text{stategen}_{\mathcal{R}}(\sigma_s), \beta_t = \mathit{mapvar}(\beta_s, m)\}$$

The mapping m corresponds to the vector mapping function f_V of the graph morphism s for the single pushout approach and $m(v) := f^{-1}(v)$ for the double pushout approach using the definitions of Section A.

The $\textbf{\textit{mapvar}} : \mathcal{B} \times \mathcal{M} \to \mathcal{B}$ function takes a variable assignment β_s of state σ_s and a mapping $m \in \mathcal{M}$ and updates β_s with respect to m resulting in a variable assignment β_t for the successor state σ_t. It is defined by

$$\mathit{mapvar}(\beta(v), m) : v \mapsto \begin{cases} \mathit{mapcol}(\beta(v), m) & \text{if } \exists t : \beta(v) \in I(\mathit{Collection}(t)) \\ m(\beta(v)) & \text{if } \beta(v) \in \mathit{Dom}(m), \text{i.e., } \beta(v) \in \sigma_{\mathit{class}} \\ \beta(v) & \text{otherwise.} \end{cases}$$

A collection is mapped by the $\textbf{\textit{mapcol}} : \mathit{Collection}(t) \times \mathcal{M} \to \mathit{Collection}(t)$ function, which applies ***mapvar*** recursively to all elements of the collection:

$$\mathit{mapcol}(X, m) = \begin{cases} \mathit{mkSequence}_t(\mathit{mapvar}(x, m) | x \in X) & X \in \mathit{Sequence}(t) \\ \mathit{mkSet}_t(\mathit{mapvar}(x, m) | e \in X) & X \in \mathit{Set}(t) \\ \mathit{mkBag}_t(\mathit{mapvar}(x, m) | e \in X) & X \in \mathit{Bag}(t) \end{cases}$$

This implementation gives us a state space $\mathcal{K}_M = (\mathcal{S}, \iota, \mathcal{T}, \mathcal{B}, \mathcal{E})$ with initial state $\iota \in \mathcal{G}$ and $(\tau_s, \tau_t) \in \mathcal{T} \Leftrightarrow \tau_t \in \text{succ}(\tau_s)$, $\mathcal{E}$ being the transitive closure of applying the **succ** function to the initial environment (ι, β_ι), and $\mathcal{S}$ and $\mathcal{B}$ being all states and variable assignments occurring in an environment.

Currently, the implementation of the $\mathit{stategen}_R$ function uses Henshin's graph rewrite engine [2]. Also note that the context object (self) is used to determine the current state making it impossible to use CTL or selector formulas for contexts not being objects of a certain state, e.g., the formula `Set{self.philosophers->at(1).left, self.philosophers->at(1).right}->forAll(x | Always Next self = x)` is currently not evaluable since the left and right fork of any philosopher in the initial state are `null`. This could be solved by adding a special context variable storing the current state.

In the following, the algorithms used for the CTL extension and the selector extension will be described. To simplify algorithms, the implementation uses a cache to store already evaluated function values where it is used that $I[\![\phi]\!]((\sigma_1, \beta_1)) = I[\![\phi]\!]((\sigma_2, \beta_2))$ if $\sigma_1 = \sigma_2$ and $\forall u \in \mathit{usedvar}(\phi) : \beta_1(u) = \beta_2(u)$ with **usedvar** being the set of variables used in the expression ϕ. The cache stores the value for each σ, ϕ and $\beta|_{\mathit{usedvar}(\phi)}$. While this cache does not completely remove the need for efficient algorithms, it is simple to use, especially regarding the need of integrating the CTL expressions and the selector expressions with arbitrary cOCL expressions as sub– and superexpressions.

CTL Extension Implementation

The *evaluation* of cOCL expressions of the form (`Exists`|`Always`) φ (`Until`|`Unless`) ψ is shown in Figure 26. The algorithm constructs the sets Φ and Ψ that contain all environments where φ and ψ hold, and a third set η that contains all environments reachable

```
/*Evaluates the given CTL expression on the start environment τ_ι.
Returns true if the expression holds, otherwise false.*/
function evaluateCTL(τ_ι, (t = Always|Exists) φ (r = Until|Unless) ψ)) : Bool
ω = {τ_ι}; /*Initialize the set of unclassified environments to the initial environment*/
Ψ = ∅; /*Environments known to fulfill ψ, i.e. being border nodes*/
Φ = ∅; /*Environments known to fulfill φ, but not ψ, i.e. being inner nodes*/
η = ∅; /*Environments known to fulfill neither φ nor ψ, i.e. being outer nodes*/
while ω ≠ ∅ /*Stop as soon as there are no more unclassified environments*/
    pick τ = (σ, β) ∈ ω; /*Select an unclassified environment*/
    ω := ω\{τ}; /*and unmark it as unclassified*/
    if I[[φ]](τ) or I[[ψ]](τ) then
        if I[[ψ]](τ) then
            /*All paths to border environments passing only inner environments fulfill the
            expression so there is no more evaluation necessary*/
            Ψ := Ψ ∪ {τ};
        else
            Φ := Φ ∪ {τ}; /*Environments fulfilling only ψ are inner environments, so*/
            ω := ω ∪ succ(τ)\(Φ ∪ Ψ ∪ η); /*add all not yet classified successor*/
        end if/*environments to the set of unclassified environments */
    else
        η := η ∪ {τ}; /*Environments fulfilling neither ψ nor φ are outer environments*/
    end if
end while
Δ = Δ_l = ∅; /*Set of inner environments not being in inner environment cycles*/
repeat
    Δ_l := Δ;
    /*An inner environment is not in an inner environment cycle, if all its inner envi-
    ronment successors are not in inner environment cycles*/
    Δ := {τ ∈ Φ | succ(τ) ∩ (Φ\Δ_l) = ∅};
until Δ = Δ_l /*Stop if there is no more change*/
Z := {τ ∈ Φ | succ(τ) = ∅}; /*Deadlock environments have no successor*/
if t = Always and r = Until then
    /*Inner environments contain neither deadlocks nor cycles and border environments
    must block reaching outer environments*/
    return φ = Δ and Z = ∅ and η = ∅;
elseif t = Always and r = Unless then
    return η = ∅; /*Border environments must block reaching outer environments*/
elseif t = Exists and r = Until then
    /*Some border environments must be reachable through inner environments only*/
    return ψ ≠ ∅;
elseif t = Exists and r = Unless then
    /*Some border environments must be reachable through inner environments only or
    there are inner environment cycles or deadlocks*/
    return ψ ≠ ∅ or Z ≠ ∅ or φ ≠ Δ;
end if
```

Figure 26: `Until`/`Unless` algorithm pseudo code

	Exists	Always
ϕ Until ψ	$\Psi \neq \varnothing$	η empty, $\Delta = \Phi$, no deadlock in Φ
ϕ Unless ψ	$\Psi \neq \varnothing$, $\Delta \neq \Phi$ or a deadlock in Φ	η empty

Table 6: Result criteria for different cOCL expressions

from φ but where neither φ nor ψ hold. The worklist ω contains all environments that need to be processed. The algorithm sets the worklist to the initial environment τ_ι and uses the **succ** function to iteratively expand the set of reachable environments. It evaluates φ and ψ in each environment τ and assigns τ to the corresponding sets Φ and Ψ, or to η if neither φ or ψ hold.

The set Δ contains environments of Φ not being usable for constructing a cycle in Φ. Thus, initially it contains only those environments of Φ having no next environments in Φ. Then, iteratively environments of Φ containing no next states in Φ or being unusable for constructing a cycle, thus being in Δ, are added until the set does not change any longer. There is thus a cycle in Φ if and only if there is $\Phi \neq \Delta$. A deadlock (in Φ) occurs if there is an environment in Φ having no successor environment (no matter whether the successor environment is in Φ or not).

Considering only subformulas that return true and false (not invalid), the satisfiability conditions of each CTL expression is provided in Table 6. Since by construction all states of Ψ are reachable by a finite series of environments in Φ, a nonempty Ψ gives us the knowledge that at least one path with φ U ψ holds. In practice, this path can be retrieved by storing a predecessor environment for every environment. Such a path is sufficient for `Exists Until` and `Exists Unless` expressions. For the CTL expression `Unless`, it is also sufficient to find an maxpath of Φ. Such a maxpath can be found iff there either is a cycle in Φ by visiting this cycle infinitely often and thus get an infinite path or if there is no successor state of an environment in Φ because then the path ending in this deadlock environment is a maxpath. Likewise, a path not fulfilling ϕ W ψ can be found by looking at an environment in η because by construction such an environment is reachable by a finite series of environments in Φ and thus gives a counterexample of the `Always` quantifier. In the case of ϕ U ψ, a path not ending in a Ψ node also is a counterexample. These paths are the cycle and deadlock paths.

Example 47 (CTL extension expression I).

```
Always philosophers->exists(status = PState::
thinking) Until philosophers->exists(status =
PState::eating) ⟹ret false
```

The state space generated for this expression is shown in Figure 27. The expression ϕ equals `philosophers->exists(status = PState::thinking)`, the expression ψ is `philosophers->exists(status = PState::eating)`. The set Φ contains all expressions where the expression ϕ, but not ψ, evaluates to `true`, starting from the initial state. The set Ψ contains all

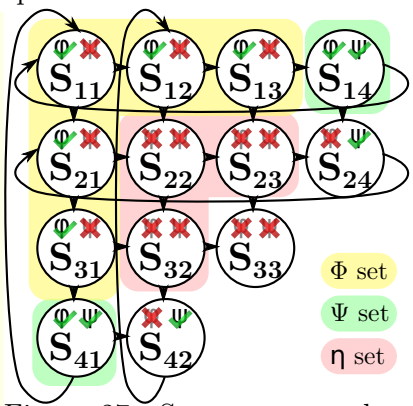


Figure 27: State space evaluated by the algorithm

expressions where the expression ψ evaluates to true. The set η contains all expressions where neither ϕ nor ψ evaluates to true. Some states are not in any set, these are the states which do not get generated by the algorithm. In the end, since η is not empty, the expression evaluates to `false`. Any path from the initial state S_{11} to any state in η, for example (S_{11}, S_{12}, S_{22}) gives a counter example.

The implementation of the **`Always Next`** ϕ expression is simpler. It is evaluated as $I[\![\mathsf{A}\ \mathsf{X}\ \phi]\!]((\sigma,\beta)) = \forall n \in \mathrm{succ}(\sigma,\beta) : I[\![\phi]\!](n) \in \{true, \mathrm{null}\}$ and **`Exists Next`** ϕ is evaluated as $I[\![\mathsf{E}\ \mathsf{X}\ \phi]\!]((\sigma,\beta)) = \exists n \in \mathrm{succ}(\sigma,\beta) : I[\![\phi]\!](n) = true$

Example 48 (CTL extension expression II).

```
Always Next philosophers->exists(status = PState::thinking)  ⟹ (ret) true
                                  φ
```

If we look again at the state space in Figure 27, we see that all states reachable from S_{11}, which are S_{12} and S_{21}, have ϕ evaluating to `true`. Thus, the expression evaluates to `true` as well.

Selector Implementation

The implementation of the selectors is depicted in Figure 28. The algorithm constructs three sets from the initial evaluation environment τ_ι, the initial selector state *init* and the expression to be evaluated for the result *ev*: The worklist set ω includes the open evaluation informations, the set η includes finished evaluation informations and the set Φ includes the results. Each evaluation info consists of the evaluation environment τ and a selector state s. A selector state in an evaluation info combines all paths leading to equivalent results in terms of the selector used. For a certain state, a selector state decides, based on the evaluation result of the expressions in the selector, whether the current state is the end of an accepted path (a), whether there cannot be any further accepted path (f) and the next selector state x. If the current state is the end of an accepted path, it is added to the result set Φ. If there might be further accepted paths, all the evaluation environments directly reachable from the current one are added to the set of remaining evaluation informations if they have not been evaluated before. At the end, a list of all evaluation results for matching end states of accepted paths is returned.

Each selector can be described by a state diagram where node transitions may emit ADD or FINAL to indicate that the current state should be added or that no further evaluation is necessary. Since the selectors are put in a map, every selector may only contain a finite number of states. Figure 29 shows the state diagrams for some basic selector range expressions.

The current selector shown in Figure 29a selects only the current state if the expression evaluates to true, otherwise it selects nothing. In both cases, since no states beside the first one can be selected, the selector emits FINAL in both cases. It contains two states.

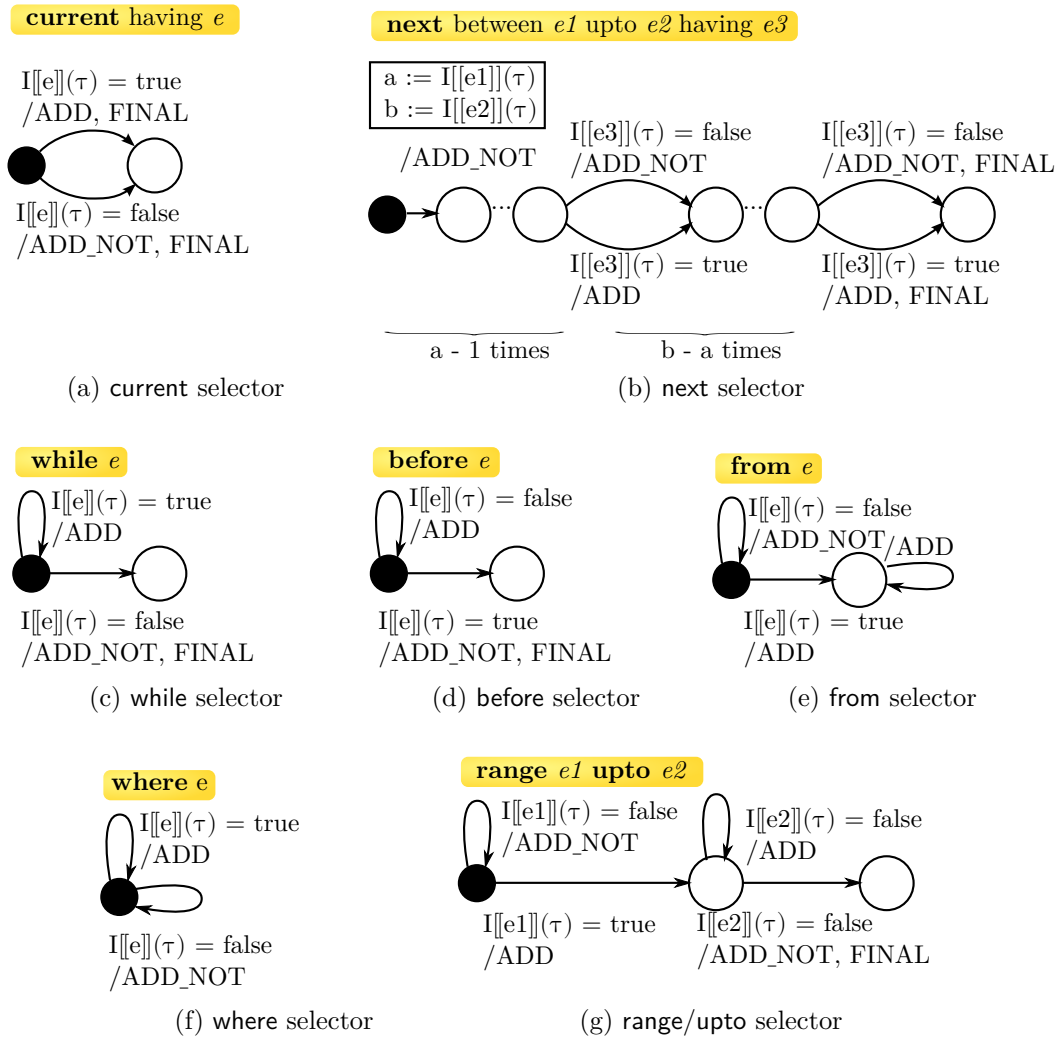
/*Evaluates the given selector expression e_t@r on the start environment τ_ι.
Returns a collection of the evaluation results of e in the last state of the paths matched by r.*/
function evaluateSelector(τ_ι, e_t@r) : Collection(t)
$\omega = \{(\tau_\iota, init)\}$ /*Initialize the worklist ω with τ_ι and an initial selector state *init* for r*/
$\eta = \varnothing$ /*Initialize the set of processed worklist elements with the empty set.*/
$\Phi = \varnothing$ /*Initialize the set of worklist elements with matching paths with the empty set.*/
while $\omega \neq \varnothing$ /*If the worklist is empty, we are finished*/
  pick $\rho = (\tau, s) \in \omega$ /*Pick some element from the worklist ... */
  $\omega := \omega \backslash \{\rho\}$ /*..., remove it from there ...*/
  $\eta := \eta \cup \{\rho\}$ /* ... and add it to the set of processed elements*/
  $l := [I[\![e]\!](\tau) | e \in s.\text{expr}()]$ /*Evaluate the selector parameters*/
  /*And then decide what to do next. f is true iff the evaluation of any paths starting with the current path is not sensible any longer. a is true iff this path matches the range r. x is a collection of successor selector states.*/
  $(f, a, x) := s.\text{eval}(l)$
  if a **then** /*Add the current worklist element to to the set of returned ... */
    $\Phi := \Phi \cup \rho$ /*... worklist elements if the path matches*/
  end if
  if not f **then**
    /*If it is sensible to process paths starting with the current path, add all successor environments with their respective successor states to the worklist*/
    $\omega := \omega \cup \bigcup_{x_i \in x}(\{(\tau_i, x_i) | \tau_i \in \text{succ}(\tau)\} \backslash \eta)$
  end if
end while
/*Return the evaluation of e_t for every matching worklist element*/
return $[I[\![e_t]\!](\tau_i) | (\tau_i, s) \in \Phi]$

Figure 28: Main selector algorithm pseudocode

The next selector shown in Figure 29b initially calculates the minimum path length of acceptance a and the maximum path length of acceptance b. During the first $a - 1$ transitions, no state will be added so the given expression $e3$ does not matter. Then, during the next $b - a$ transitions, a path is accepted if $e3$ evaluates to true, otherwise it is not accepted. After the last transition, no more nodes can be added so FINAL is emitted. It contains $b + 1$ states.

The while selector shown in Figure 29c selects states as long as the given expression is fulfilled. As soon as the expression is not fulfilled, it emits FINAL and does not add the state any longer. The before selector shown in Figure 29d is a similar selector selecting states as long as the given expression is not fulfilled. It contains two states.

The from selector shown in Figure 29e starts selecting states as soon as the specified



(a) current selector (b) next selector

(c) while selector (d) before selector (e) from selector

(f) where selector (g) range/upto selector

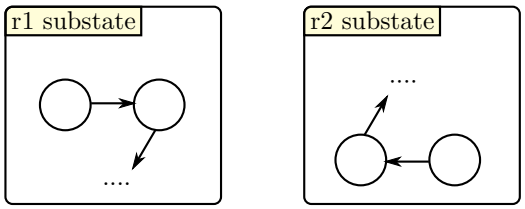
Figure 29: State diagrams for different primitive selector operations

condition is fulfilled. In contrast to the while and before selectors before it however cannot stop as soon as the condition is (not) fulfilled but has to continue emitting values. It contains two states.

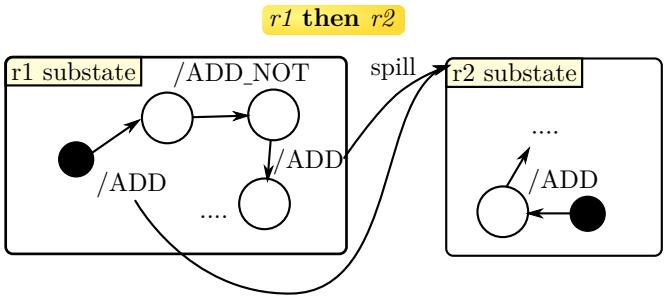
The range ... upto selector shown in Figure 29g starts selecting states as soon as the first condition is fulfilled and stops as soon as the second condition is fulfilled, then FINAL is emitted. It contains three states.

Set operation selectors combine two subselectors each as shown in Figure 30. The selector with contains paths of both subselectors so it emits ADD whenever at least one subselector emits ADD. Likewise, it only knows that no further paths will be accepted if this is known for both subselectors and it emits FINAL only if both subselectors emit it. The selector intersect contains only paths found in both subselectors so it emits ADD

	r1 **with** *r2*		*r1* **intersect** *r2*		*r1* **without** *r2*	
r1 \ r2	ADD	ADD_NOT	ADD	ADD_NOT	ADD	ADD_NOT
ADD	ADD	ADD	ADD	ADD_NOT	ADD_NOT	ADD
ADD_NOT	ADD	ADD_NOT	ADD_NOT	ADD_NOT	ADD_NOT	ADD_NOT
r1 \ r2	FINAL	NOT_FIN	FINAL	NOT_FIN	FINAL	NOT_FIN
FINAL	FINAL	NOT_FIN	FINAL	FINAL	FINAL	FINAL
NOT_FIN	NOT_FIN	NOT_FIN	FINAL	NOT_FIN	NOT_FIN	NOT_FIN



(a) With, intersect and without selectors



(b) Then operator

Figure 30: State diagrams for different compound selector operations

when both subselectors emit ADD. As soon as one subselector is FINAL intersect knows it is finished and emits FINAL as well. The selector without contains paths found in the first, but not the second subselector so it emits ADD when the first, but not the second subselector emits ADD. As soon as the first subselector is FINAL it knows that no further paths will be accepted and emits FINAL.

The then operator accepts path starting with an accepted path for r_1 and continuing with an accepted path for r_2. Whenever the first subselector r_1 emits ADD, a new fresh subselector r_2 is returned for the state which would otherwise have been added. Thus, for each ADD in fact the returned x contains two values: A then selector with updated first subselector and a new r_2 subselector. If r_1 does not emit ADD, only the updated subselector r_1 is returned. FINAL is emitted as soon as r_1 emits FINAL.

Example 49 (Selector operation).

```
self@((while self.philosophers->exists(p | p.status = PState::thinking))
intersect (next 2))
```

This operation selects the tables of all states where there is a path of arbitrary length with at least one philosopher thinking which are also reachable by a path of length two

or three. Figure 31 shows the automata for the `while` and `next` subexpressions as well as the combined automata for the whole expression. The automaton for the `next` range has four nodes. The first state is not added, the second and third state are added. Since there are no optimizations done, the next automaton and thus the combined automaton as well have redundant information. Because the `having` attribute is not specified, it is assumed to be `true`. This expression can obviously not evaluate to false, thus the second and third transition with ADD_NOT of the `next 2` automaton can never be taken. This, however, does not hurt so much since it only increases the size of the generated automaton a bit. It does not increase the number of evaluated states.

The actually taken transitions are marked with the state leading to the transition with red states not being added to the result set and green states being added to the result set. The first state S_{11} has at least one philosopher thinking, so the transition to R_{21} is taken and the successor states of S_{11}, which are S_{12} and S_{21} are added to this state. Each of these states has one philosopher thinking, so again the transition to R_{31} is taken for both states. Since this transition has an ADD marker, both states are added to the result state set. Again, the successor states of these three states are added to the state R_{31}. These are the states S_{31} and S_{22} for S_{21} and S_{13} and S_{22} for S_{12}. Due to the worklist being a set, there are three states, namely S_{13}, S_{22} and S_{31} in R_{31}. The state S_{22} has no philosopher thinking, so the transition to R_{32} is taken. Since this transition is marked as FINAL, the successor states of S_{22} are not added to the worklist. Since it is also marked as ADD_NOT, S_{22} is as well not added to the result set. S_{31} and S_{13} however have one philosopher thinking, so the transition to R_{41} is taken and they are added to the result set. Since this transition as well is marked as ADD_NOT, their successor states are not added to the worklist.

Thus, the result set contains the evaluation of `self` on the states S_{12}, S_{21}, S_{31} and S_{13}.

Reporting Information

Especially in case of model checking formulas we are not only interested in the actual result, but also in the reason for the result w.r.t. the model, e.g. an example or a counterexample. Thus, any evaluation of a cOCL expression yields a *report* that, besides returning the result of the evaluation, contains a *cause* or explanation for the result. A cause is like an evaluation tree in the sense that it stores results of partial expressions of the full cOCL expression, but different in the sense that it stores only the results for the expression parts it considers relevant. A sub–expression is *relevant* if it influences the result of its super–expression.

In general, the cause contains all information of subexpressions except for those cases in which it knows only a subset of the information is necessary. For expressions of the form `Exists` ϕ `Until/Unless` ψ, these are the states of a maxpath fulfilling the expression if available. For expressions of the form `Always` ϕ `Until/Unless` ψ, these are the states of a maxpath not fulfilling the expression. For boolean operations, this is any expression evaluating to true for an `or` or `exists` operation and any expression evaluating to false for an `and` or `forAll` operation. For selector expressions, the cause always returns all

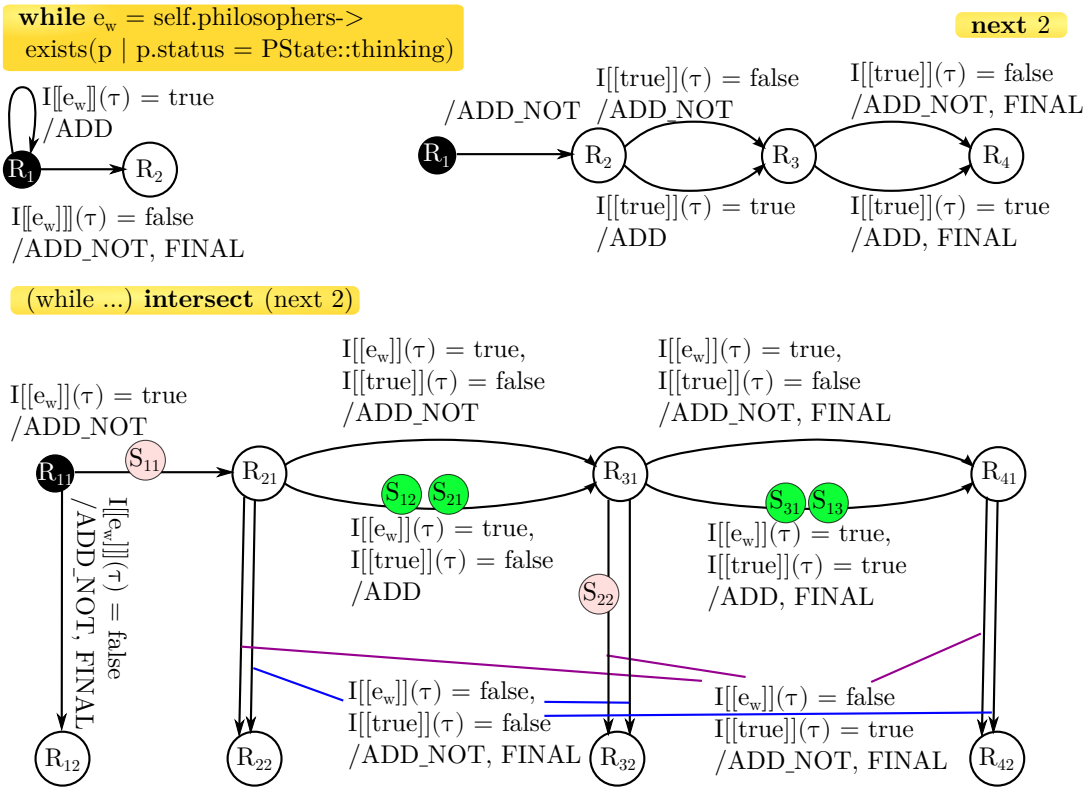


Figure 31: Automata of the example selector subexpressions and the combined automaton

evaluated values. Of course there often are multiple possible selections for the cause of an expression, e.g. multiple counter example paths. Currently, in those cases an arbitrary selection is done, but in principle some kind of optimization criterion like finding the smallest cause could be considered.

Conceptually, the cause for any expression is returned additionally to the usual return value. For every expression, the cause consists of the expression, its evaluated value and, as subcauses the causes of all subexpressions considered relevant. Except for specific implementations in the CTL algorithm, the boolean operations and some collection operations, all subexpressions are considered relevant. In order to not break the existing interface, the causes are stored on a separate stack and not passed as return values. The cause structure for standard OCL expressions directly matches the expression structure. The cause structure for CTL and selector expressions is more complicated, since their subexpressions are evaluated multiple times in different states. Thus, these expressions do not directly contain causes of their subexpressions, but special causes representing individual states. Each of this special cause contains information about the state name, incoming and outgoing transitions if calculated and the expression result of the CTL or selector subexpressions. For convenience, in the case of a single (counter)example

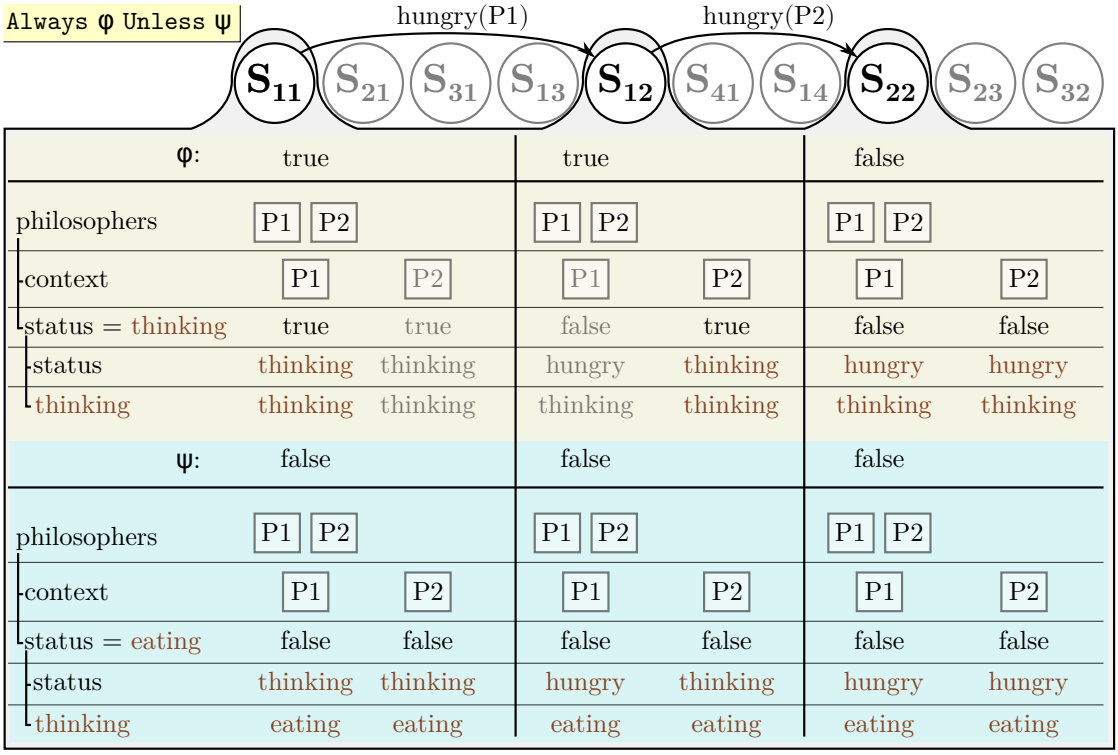


Figure 32: Cause result (transparent parts are not part of the cause)

path, only the end state of this path is directly returned as cause with the possibility to retrieve the previous states of this path.

In case of cOCL expressions the report generation becomes expensive fast. For example, the number of generated sub-causes for a counter-example trace of a E F φ formula, where φ is a propositional formula without set operations, has as upper bound $\mathcal{O}(|\mathcal{K}_M| \cdot |\varphi|)$, the size of the state space times the size of the formula φ.

Example 50 (Cause). This example illustrates the generation process of a cause.

```
Always philosophers->exists(p | p.status = PState::thinking) Unless
philosophers->exists(p | p.status = PState::eating) ⟹ret false
```

This expression asks whether if a philosopher is eating, there always is a philosopher thinking before as seen in Subsection CTL Extension Implementation. Figure 32 shows a representation of a possible cause for this expression. Only the fully opaque states are part of the cause even though all transparent states might have been generated when evaluating the expression. These three states show a clear trace to a state where no philosopher is thinking but no philosopher is eating as well because both philosophers are hungry. The root cause for this expression is the `Always` ϕ `Until` ψ expression itself. Since there is a single example showing a state leading to this expression evaluating to false, the cause for the single state S_{22} is returned as single subcause of this expression.

It contains information about the state name, but also information that this cause was reached via `hungry` and the previous state was S_{12} together with S_{12}'s cause. The causes for the expressions ϕ and ψ evaluated in S_{22} are as well subcauses of this cause. The source of the `exists` operation is `philosophers`, thus the set of the two philosophers $P1$ and $P2$ is a subcause of this exists operation. The evaluation of the expression `p.status = PState::thinking` is relevant for both philosophers, thus the cause for this expression is given for both philosophers. The subcauses for each philosopher look the same. The expression `p.status = PState::thinking` is false because `p.status` is `hungry` which is different to `thinking`. When considering the same expression ϕ in S_{12}, on the other hand, we can see that only a single subcause is relevant. The operation `exists` is true because the status for $P2$ is `thinking`. Thus, the status of $P1$ is not relevant. The same holds for the expression ϕ in S_{11}. Since `exists` is true because the the status of $P1$ or $P2$ is true, only one subcause is necessary. In this case, however, both subcauses are candidates for being included in the reported cause. For all states, the cause for the expression ψ is the full subcause of the `exists` operation since the operation evaluates to false.

5.6 User Interface

The user interface for the tool is written in HTML/CSS/JavaScript using a Tomcat 7 servlet. Figure 33 shows an example of the tool output for the cause also shown in Figure 32. Figure 34 shows the tool output for an expression selecting states potentially leading to a deadlock state. Initially, the user chooses his/her model by selecting the Ecore file, the Henshin file, and the initial model file in (1). Then, the rules he/she wants to use in the state space exploration can be chosen in the select field on the right or written in the text box in (6). The cOCL expression is written in the textbox in (5). Then, the user needs to choose whether to generate a detailed report or just evaluate the expression by (un)checking the checkbox in (3). Not generating a cause does not severely improve performance, but might decrease memory requirements. Then, the Evaluate–Button in (7) is pressed.

During this interaction, two AJAX requests are generated. One request is sent after the Ecore file and the Henshin file have been selected to let the server determine the rules in the Henshin file. The second, larger, request is sent when the evaluation button is pressed and includes all information entered in the form. The server then evaluates the expression using the given files and transforms the result including the cause, time and possible error information into a JSON representation. This JSON object then is parsed in the client and displayed.

If no cause is requested, then the only result displayed is the evaluation value and some timing information found near the evaluation button. For the special results `true` and `false`, the output `Property fulfilled` and `Property not fulfilled` is given, otherwise just a small string representation is given. Additionally, the time spent for the evaluation excluding parsing the Henshin and the Ecore files and the expression and transforming the cause to JSON is given. Generating the JSON cause can take signifi-

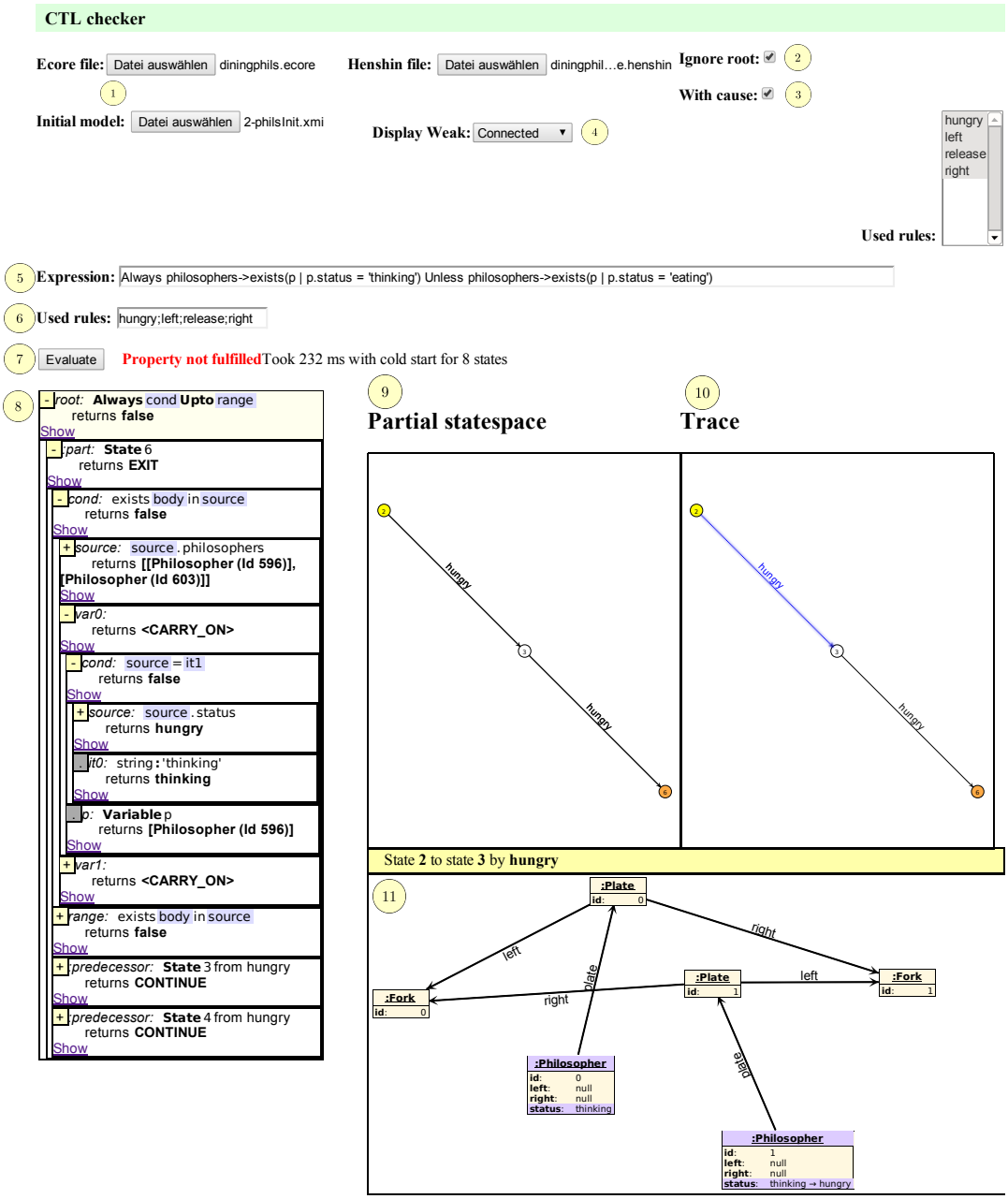


Figure 33: Sample tool output for the expression `Always philosophers->exists(p | p.status = PState::thinking) Unless philosophers->exists(p | p.status = PState::eating)`

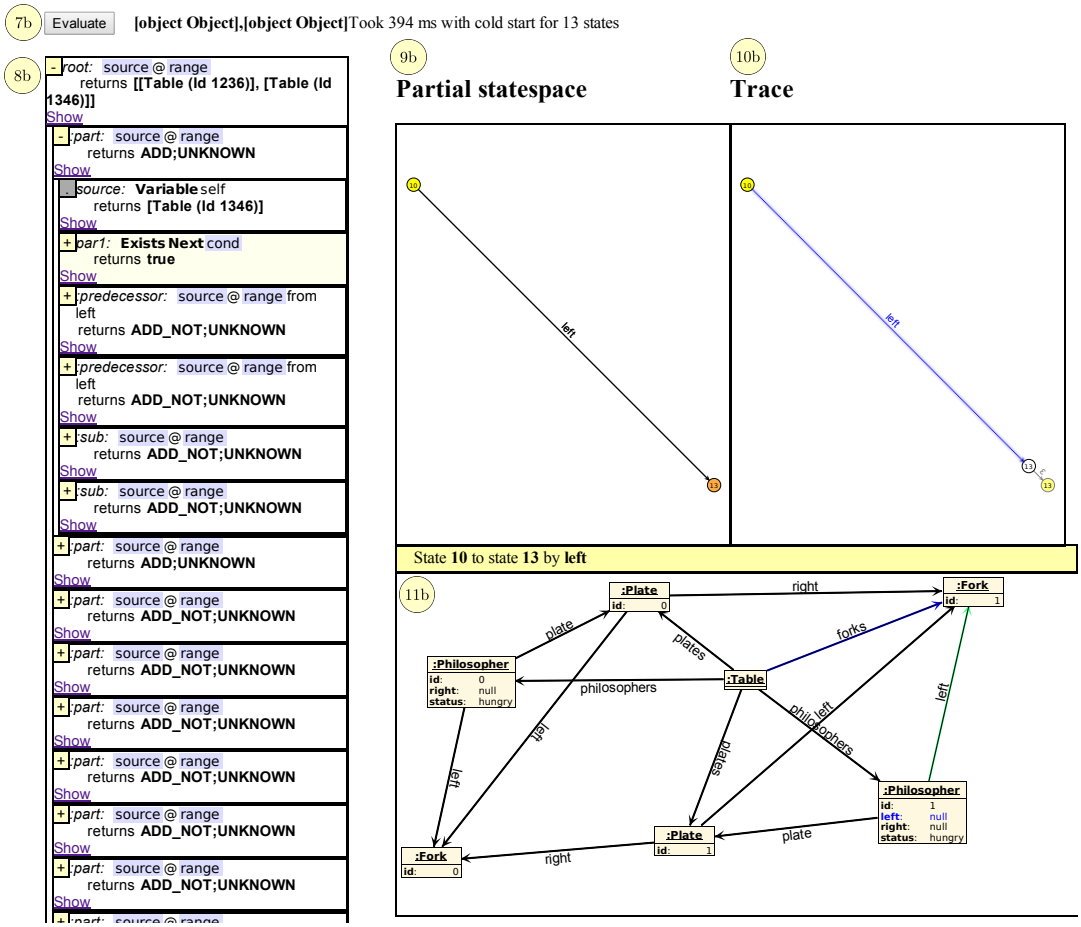


Figure 34: Sample tool output for the expression `self@(where (Exists Next (Always Next false)))`

cant time, especially if the generated result is large, e.g. if no single counterexample can be given.

Everything displayed is generated out of this JSON cause. The tree view seen in (8) is a direct translation of the cause into a tree view. Every field contains the evaluated (sub)expression on the top. In order to save space, not the full expression is given, but just the top–level expression, the sub–expression is condensed to a parameter name. For every expression, this parameter name is used in the top left corner. The full subexpression is stored as tooltip of this parameter. Note however that only the abstract syntax is available from the OCL Engine lacking some information like parenthesis. Thus, the expression in the tooltip does not fully equal the entered expression. Additionally, the return value is given. Subexpressions can be accessed with clicking on the +–sign at the top left. A gray dot indicates that this expression cannot be expanded any more.

For any CTL expression, there are special `part` subexpressions including relevant states. The return values of these states have three special values: `EXIT`, `STOP` and `CONTINUE`. `EXIT` indicates that the evaluation ended at this state and the evaluation result was known, `STOP` indicates that the mentioned state was not expanded any longer and

`CONTINUE` indicates that the mentioned state was expanded. All states have `predecessors`, which are states from which this state was reached. Additionally, states can have `substates`. A state is a substate if there is a transition to this state. Selector expressions are handled similarly with the only exception that the return values consist of `ADD`, `ADD_NOT` or `UNKNOWN` for the current and future states. `ADD_NOT` for future states is equivalent to `STOP` for CTL expressions. Examples for CTL part subexpressions can be seen in (8) of Figure 33. Examples for selector part subexpressions can be seen in (8b) of Figure 34.

These expressions can be visualized by clicking on the `Show` link. If this link is clicked, two state spaces are generated. The partial statespace is displayed on the left side (9). It includes all states occurring somewhere in the cause together with all available transitions for these states. Note that since the evaluation may abort early, some transitions from a potential full statespace are missing in this partial statespace because they have not been generated during the evaluation. The trace on the right side (10) contains the exact evaluation tree and only the relevant transitions. Initial states are marked in yellow, deadlock states in orange. The selection in (4) allows to choose whether to display sub–CTL–expressions as well. If the display is set to `Connected`, then all additional states and transitions are directly added to the partial statespace partially transparent. They are as well connected using an ϵ–edge in the trace. The transparent states in (10b) of Figure 34 correspond to the inner expression `Always Next false`. There is only one transparent state in the example shown because the expression `Always Next false` evaluated to `false`. If the display option is set to `None`, then these expressions are not shown. If a state is selected either on the left tree view or in one of the state spaces, this state is shown in the window below. It contains all model elements of this state. Transitions may be selected as well. In this case, the concrete model transition is displayed in storyboard notation ([27], see Section 2.3) with green arrows indicating added transitions, and blue arrows indicating removed transitions and attribute changes as well indicated by arrows as shown in (11b) of Figure 34. Attribute changes are represented by an arrow connecting old and new value as shown in (11) of Figure 33. All model elements occurring as result of subcauses of the selected expression — the *relevant* elements — are highlighted in violet as shown in (11) of Figure 33. Typically, the root element is not necessary for understanding the model, but makes the graph unnecessarily complex. Thus it can be omitted using the ignore root checkbox in (2).

The model layout is done iteratively using a force–directed algorithm similar to the one used by the Henshin state space visualisation. Objects repel each other and are attracted to other objects to which they have a transition except if these other objects are too close. The ϵ–transition has high attraction and states with the same name have no repelling. To ease interaction, the layout algorithm is suspended if the mouse is hovered over the corresponding state space.

As depicted in Figure 35, all objects are displayed without attributes and having only their initial letter as name together with a color coming from a hash function if the model gets too large. All attributes can be seen by hovering over the corresponding object.

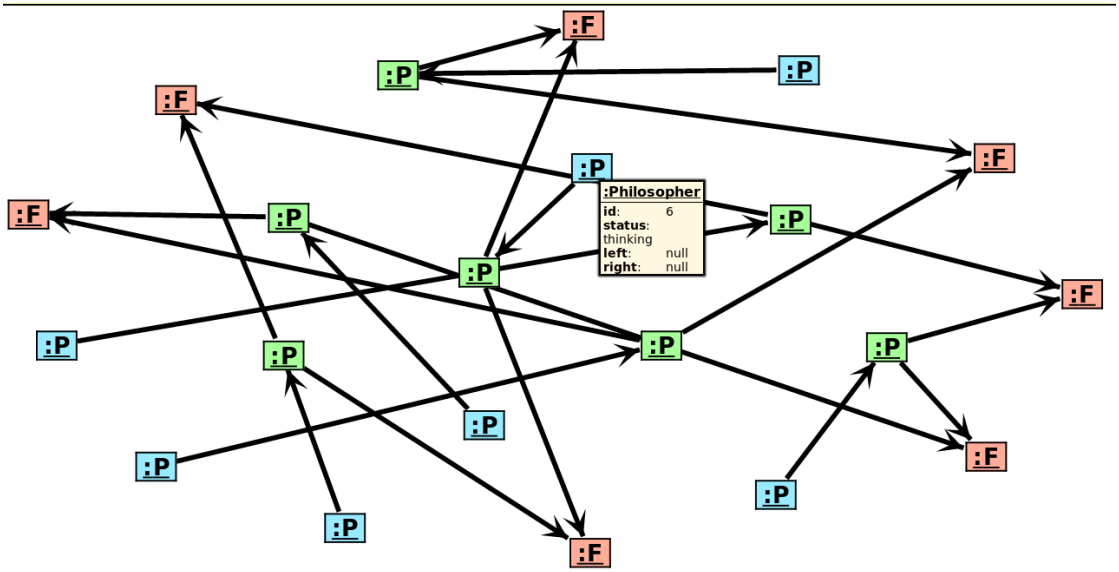


Figure 35: Model view of too many objects for detailed display

5.7 Summary

The MocOCL framework provides two extensions of OCL: On the one hand, a CTL extension of OCL allows to verify dynamic properties of software models. On the other hand, a custom selector extension of OCL allows to retrieve general information about the software by evaluating OCL queries on a specific set of states determined by the selector ranges. We formally introduced syntax and semantics of both extensions and showed how they are implemented in the MocOCL framework. With this extension, we have an OCL–based language for both static and dynamic property specification and with MocOCL the tool for evaluating these specifications in an interactive user–friendly manner.

CHAPTER 6

Evaluation

In this chapter, we evaluate the MoCOCL framework. A focus of this thesis was to improve the usability of a model checking language in the context of MDE and the understandability of the result returned by the model checker. For evaluating the usability, a qualitative user study was conducted using a custom Pacman–like setting. The Pacman example and the rest of the experimental setup is explained first. Then, the study results are given. For evaluating the performance, three different queries were used on multiple Pacman scenarios with varying number of ghosts and fields. Additionally, expressions for finding deadlocks and generating the whole state space were used on the dining philosophers problem with a varying number of philosophers. Finally, the evaluation results are discussed.

6.1 Pacman Evaluation Scenario

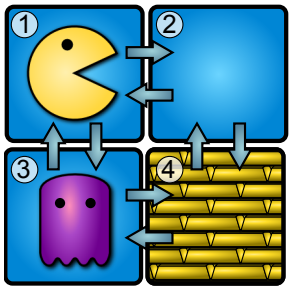


Figure 36: Initial field

Besides the dining philosophers problem presented in Section 1.1, we consider a variant of the game Pacman[1] in our evaluation. Pacman is one of most well-known games in video game history. In the original game, Pacman is on a board with dots, ghosts and power–ups. Pacman's task is to collect all dots in a level without getting touched by a ghost. Whenever Pacman eats a power–up, the ghosts reverse direction. The main advantage of Pacman in comparison to the dining philosophers problem is that it has many interesting properties and yet is commonly known and easy enough for being used in a user study. Even though it was not a selection criterion, it offers the possibility to easily influence the state space size making it a suitable candidate for performance evaluation. An upper bound for the state space size is n^{g+1} with n being the number of game

[1] http://en.wikipedia.org/wiki/Pac-Man

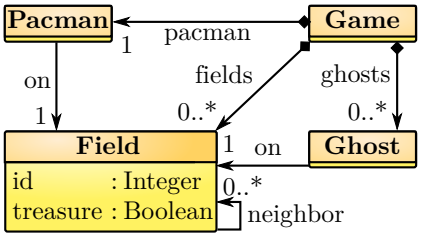


Figure 37: Pacman class diagram

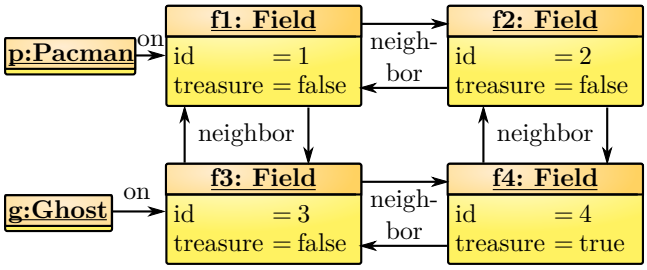


Figure 38: Partial object diagram of the initial field. All objects are contained in the game object

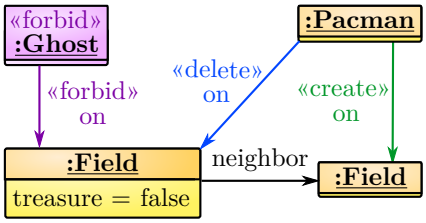


Figure 39: Pacman move rule

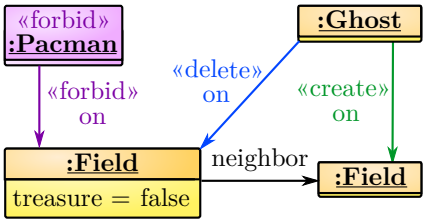


Figure 40: Ghost move rule used in the evaluation, with error

fields and g being the number of ghosts since any ghost and Pacman can potentially be on any field.

In the following, the game variant and its model representation are introduced shortly. The game is simplified and modeled using graph transformations inspired by Heckel [41]. In the simplified version used, there are no dots, but treasures on the board. Pacman does not need to collect all treasures, but just reach one treasure not occupied by a ghost. There are no power–ups. The board consists of multiple fields. While typically each field is a square, this restriction is lifted in this simplification. We will see below that in each round, either Pacman or a ghost may move to an adjacent field (diagonally adjacent fields are not considered adjacent). Thus, it might be the case that Pacman moves multiple times without the ghost moving in between.

The static structure of the game is shown in Figure 37. A game consists of multiple fields, but at least one, with an integer `id`. Some fields contain a treasure indicated by a boolean flag. Each field has up to four neighboring fields with the intention that Pacman or a ghost can move to neighboring fields in a single step. There is one Pacman and zero or more ghosts which are on a certain field each.

The dynamic behavior of Pacman as used in the user study is shown in Figure 39. A move occurs by changing the `on` reference of Pacman or a ghost to a field next to the original field. If the game is over, no more moves may happen. Thus, Pacman may only move if not currently on a treasure and there are no ghosts on Pacman's field. The first condition is expressed using the attribute `treasure = false` excluding that Pacman is on a field with a treasure. The second condition is formulated as NAC (see Section 2.3):

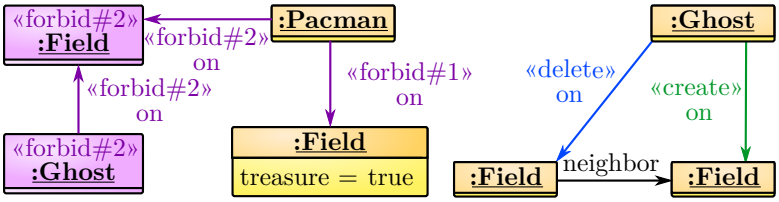


Figure 41: Correct ghost move rule

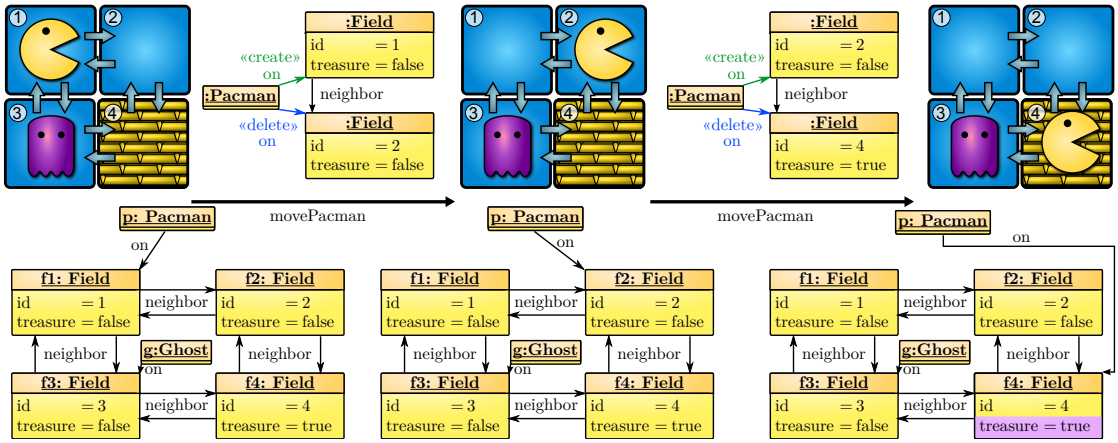


Figure 42: Example graph transformation

If a ghost is on the same field Pacman is, the rule cannot be executed.

The dynamic behavior of the ghost as used in the user study is shown in Figure 40. A ghost may move to an adjacent field as well, but only if there is neither Pacman on the field the ghost is nor Pacman has found the treasure. Like in the Pacman move rule, there is a NAC forbidding that the ghost moves if Pacman is on the same field as the the ghost. But for evaluation purposes, an error in our model is deliberately introduced: The rule has no NAC to forbid that the game is over because another ghost is on the same field as Pacman and no NAC to forbid that the game is over because Pacman has found the treasure. There were two main reasons for working with the erroneous rule. On the one hand, the error was intended to be found while using the tool. On the other hand, the correct rule is much more complicated and harder to understand. The correct dynamic behavior is shown in Figure 41. The right part just makes the ghost move to a neighboring field. The left part contains NACs. There are two different NACs, namely the one with ID `1` and another one with ID `2`. The rule is only executable if neither NAC matches. Thus, it may not be that Pacman is on a field with a treasure (NAC `#1`) and there may be no ghost on the field Pacman is (NAC `#2`).

An example also used in the evaluation setting for applying graph transformations is shown in Figure 42. A game is shown where Pacman finds the treasure in two steps from Field 1 to Field 2 to Field 4.

Expression type	**Natural language expr.**	**cOCL expression**
Simple boolean expression	It is possible that the game is over	`sometimes eventually (always next false)`[2]
Non–boolean expression	The set of all ghosts imposing a potential danger to Pacman	`self.ghosts->select(g | sometimes eventually g.on = self.pacman.on`
Complex expression	As long as the game is not over, every ghost may move to at least two different positions	`always self.ghosts->forAll(g | g.on.neighbour->select(field | sometimes next g.on = field)->size() >= 2 unless (always next false)`

Table 7: Some examples presented in the evaluation setting

6.2 Usability Study

The purpose of the usability study was to evaluate the usability of the CTL extension of OCL described in Section 5.3; especially how hard it is to read and write it and the usability of the tool. More specifically, it was evaluated how helpful the tool was to find out why an evaluation failed or succeeded.

Participants were invited to a single–person semi–structured interview using the questionnaire found in Appendix B. The participants included researchers from software engineering and modeling mostly with one single participant from the logics field. Some of them had used model checkers already, some others had not. An interview took about one hour, but up to two hours in some cases. Note that the questionnaire was adapted a bit during the course of the interviews for better introduction to the problem and clarification of the tasks.

At first, after a short introduction of the problem tackled with MoCOCL, each person had to provide a self–assessment of previous skills regarding their modeling, graph transformation and model checking skills. The assumption was that persons good in all of these areas could handle the language better than those without, but the model checking experience should not be absolutely necessary.

Then, the example was introduced with a short explanation of the class diagram, the transformations and the example provided. In the next step, the language was explained as an extension of OCL to be able to cover the system behavior. The language features were described using a mixture of CTL semantics and reference to the actual example, e.g. the term “game run” was partially used instead of “path”. Afterwards, several examples were provided to give a feeling of the language. Some of these examples are listed in Table 7, the others can be found in Appendix B.

	Task Nr.	Expression
read cOCL	2a	`sometimes next sometimes next self.pacman.on.treasure = true`
	2b	`sometimes eventually self.ghosts->exists(ghost | ghost.on = self.pacman.on)`
	2c	`self.ghosts->exists(g | g.on.treasure) and sometimes eventually (self.pacman.on.treasure and self.ghosts->forAll(ghost | ghost.on <> self.pacman.on))`
write cOCL	3a	After three turns, Pacman maybe has found the treasure.
	3b	A rule of the game: The game is over if Pacman has found the treasure.

Table 8: Reading and writing tasks of the user study

Some tasks were presented to the test persons regarding the understandability of cOCL expressions and the tool output as well as the easiness to write cOCL expressions. Task `1` was to find natural language expressions matching cOCL expressions which should serve primarily as further introduction to cOCL. The following tasks listed in Table 8 were dedicated to reading and writing cOCL expressions and using the tool. In Task `2`, the natural language meaning of the expressions presented in cOCL had to be given, in Task `3`, cOCL expressions had to be written. The tasks were sorted in ascending difficulty. The first expression in Task `2a` closely resembled the example `sometimes next self.pacman.on.treasure = true` used to introduce cOCL. The expression in Task `2b` contained a more complex OCL expression, but only a single, simple CTL part. The expression in Task `2c` combined two different temporal aspects, namely a property holding in the initial state only and a property holding in the future. In Task `3`, cOCL expressions for natural language expressions had to be constructed. Task `3a` again was rather easy: The expression that Pacman has maybe found the treasure in one turn was an cOCL introduction example, the expression that Pacman has maybe found the treasure in two turns was found in Task `2a` and now this task just consisted in having three instead of two turns. Task `3b` was more complicated. This expression cannot be directly inferred from previous expressions and a suitable temporal operator has to be found. Additionally, for each expression in the Tasks `2` and `3`, the result of this expression evaluated on the example had to be determined and an explanation, like a matching game, had to be given. The participants were encouraged to use the tool. Task `3a` revealed the expected result `true`, but with an unexpected explanation given by the tool: Pacman moves to the treasure first, then the ghost makes a move. This is a behavior not compatible with the game description. The expression in Task `3b` formalizes this violated property.

After these tasks, the test persons were asked to give their opinion on the language and the tool and provide further recommendations. That included a subjective evaluation whether reading and writing cOCL and interpreting the tool output was `easy`, `medium`, `difficult` or `infeasible`.

Preknowledge	Task result			Subjective Evaluation		
	Low	**Medium**	**High**	**Low**	**Medium**	**High**
Structural Models	12	8	10.5	8	7.5	7
Behavioral Models	8	10.1	11	7	7.7	6
OCL	12	9.6	11.5	8	6.9	8
Graph transform.	10	9.4	11.7	8	6.9	8
Standard Logics	9.5	10.3	10.4	5	7.4	8
Temporal Logics	10.4	9.9	—	6.8	8.3	—
Model checkers	10	–	10.4	6.5	–	6.7

Table 9: Evaluation results based on self–estimated proficiency

Table 9 shows the average points reached on the tasks on the left and the difficulty points in the subjective evaluation on the right. For each area, the participants were aggregated in groups with low, medium or high proficiency. These groups were created by forming the average of the proficiencies in the corresponding subareas with the proficiency `None` yielding 0 points, `Little` yielding 1 point, `Significant` yielding 2 points and `Expert` yielding 3 points. The question, whether model checkers were used before allowed only `Yes` and `No`, so `Yes` were awarded 3 points and `No` 0 points. A participant's proficiency is assumed to be low for at most 1 point, medium for at most 2 points and high else. A completely correct solution for a task yielded 2 points, a partially correct solution or a solution found with help yielded 1 point and a wrong or no solution yielded 0 points. In total, there were 12 points reachable. For the subjective evaluation, for each of the aspects "Reading cOCL", "Writing cOCL" and "Tool output interpretation", 0 points were given for the answer "infeasible", 1 for "difficult", 2 for "medium" and 3 for "easy". In total, there were 9 points reachable. The simple tasks (`1`, `2a`, `2b`, `3a`) could be solved by all the participants, even though there was one surprising answer for `3a` (`always next` instead of `sometimes next`) caused by an ambiguity of the question "After three turns, Pacman has found the treasure" which did not specify whether Pacman has surely or maybe found the treasure. To avoid this ambiguity, the question was changed to "After three turns, Pacman has maybe found the treasure".

Despite the small sample size, we got a first impression how personal experiences and background influence language and tool usage. However, a larger user study is out of scope of this thesis and subject to future work. For example, for modeling and OCL there is just one person each with low proficiency. If low and medium proficiency groups are put together, the results indicate that experience in modeling helps quite a lot for solving the cOCL tasks. It does, however, not help people feeling that using cOCL is easier. Graph transformation knowledge seems to have no influence on either task solving or subjective easiness.

Knowledge in logics seems to be helpful. The more knowledge in logic is available, the subjectively easier task solving is for the participants. People with previous experience in model checking also performed better than people without model checking

knowledge. Both groups are about equally big: There were six persons without model checker experience and five with, thus the results are explained in detail. Only one person without previous experience in model checking could solve all exercises while two persons completely failed at Task `3b` and two could solve it partially. In contrast, two persons with model checking experience could completly solve Task `3b`, two could solve it partially and only one person failed. Errors were quite mixed, often confusing the CTL operators, e.g. `Always Eventually` instead of `Always Globally` or boolean operators, e.g. `and` instead of `implies`. The distribution of Task `2b` was equal between persons with and without previous model checking usage, with three persons finding the correct and two a wrong solution. There was no one who could not solve Task `2b` but Task `3b` completely correct. Interestingly though, temporal logic knowledge seemed to help not at all. There were only three persons who considered themselves to have `significant` temporal logic knowledge, so it might be a statistical artifact, but since two of them failed at Task `2c`, it might be that they just are not used to expressions evaluated on a known, initial state.

People found using cOCL and the tool `easy` to `medium`. Reading cOCL expressions was found to be `easy` by eight persons, `medium` by two and `difficult` by one. Writing cOCL expressions was found to be `medium` by seven persons, `easy` by three and `difficult` by one. Interpreting the tool output was found to be `easy` and `medium` by five persons each, and `difficult` by one. Interestingly enough, there was no strong correlation between number of solved tasks and subjective evaluation. Out of four persons not being able to solve the reading Task `2c`, three found reading `easy` and one `medium`. Two people who could solve the writing Task `3c` found reading the tool output to be `medium` and only one found it to be `easy` while two of the people who could not solve this task at all found reading the tool output to be `easy` and one to be `difficult`.

The visualization of traces was generally reported as useful, even though the ϵ–transition connecting identical states in different internal evaluation stages was reported to be confusing for some. Some people would like to have a more detailed explanation and a comparison with other tools. The tree view, on the left, however, was considered as complicated to understand in the first place even though the meaning became clear after some tool usage for some people. Several suggestions for the tool were made, with some smaller changes implemented in between the interviews. One larger, yet unimplemented suggestion was to include different counterexamples. Currently, the tool provides only one counterexample.

6.3 Performance Evaluation

The goal of the performance evaluation was to find out about the model sizes which the evaluation engine could handle. Since there was no focus on performance as our implementation should serve as proof of concept, it was not expected that the results would be good. The evaluation setup consisted of evaluating various queries on different initial models for Pacman and the dining philosophers problem. These queries were evaluated using a simple program executing the query on the model multiple times and calculating the average. Only the time for building and evaluating the query was

measured, the time for transforming the cause into a JSON format suitable for the web interface was not taken into account.

Since the generated state space can be reused, all queries were executed two times in a row: In the first execution, the state space was generated and the expression was evaluated. Thus, the time needed for evaluating the query consisted of both state space generation and expression evaluation. On a subsequent run, the state space does not need to be generated any more, so the time needed equals the time for the expression evaluation alone. The difference between both times yield the state space generation time.

All performance results were evaluated on a Intel i5-2410M Machine with 2.30 GHz and 8 GB RAM.

Pacman

For Pacman, three queries and five different game boards varying in size and number of ghosts were used. At first, the game boards used will be explained, then the queries will be described.

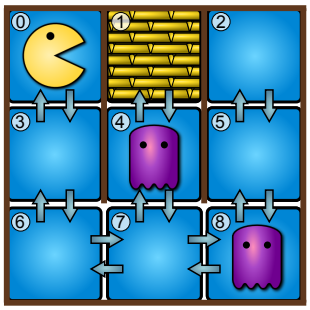


Figure 43: 3x3 test game field

The first example board is the board shown in Figure 36. It contains four fields, a ghost and a Pacman. All the 16 configurations of Pacman and the ghost being on a field can occur, so there are 16 states altogether. Pacman may simply move to the treasure in two steps.

The second example board as shown in Figure 43 contains nine instead of four fields. The challenge for the evaluation here is that the ghost has to perform a certain sequence of steps in order to allow Pacman to reach the treasure. Specifically, the ghost has to move away from the field next to the treasure to some field in the right corridor.

The third example board shown in Figure 44 is an abstraction of a real Pacman game board in a way that only crossing nodes are considered. While there are no problems for Pacman to reach the treasure here, the large state space might impose a problem to the evaluation engine. It was tested how many ghosts could be added to the game field without the evaluation engine not being able to finish any longer. For the one test case, no ghost was used, for another test case only the left ghost was used and finally, it was tried to use both ghosts. In order to be able to evaluate the large state space with two ghosts, more memory was given to the JVM.

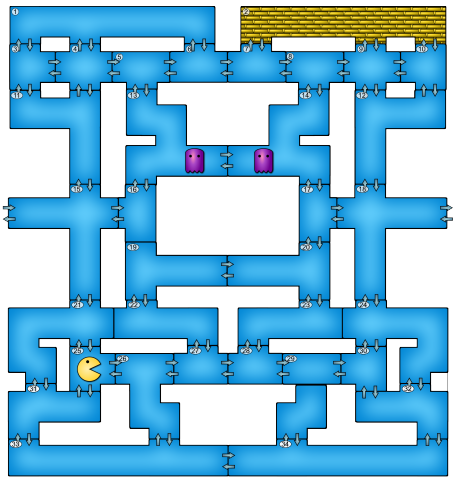


Figure 44: Large test game field

The performance evaluation was conducted using three queries, namely `Always Globally`

	Field	Ghosts	States	gen.time		eval. time		total	
				avg	std	avg	std	avg	std
state space generation	small	1	16	25	6.1	20	5.9	46	7
	medium	2	405	1051	623.3	114	42.6	1165	657.6
	large	0	34	128	63.8	20	5.1	148	68.9
	large	1	1156	7712	381.4	258	68.6	7970	437.4
	large	2	20230	213k	16.3k	5164	432.6	218k	16.4k
Pacman on treasure	small	1	10	19	21.3	29	2.4	48	22.4
	medium	2	120	124	18.3	63	19.9	188	36.2
	large	0	34	85	9.9	28	0.4	113	10
	large	1	631	1932	57.8	114	28.9	2046	38.9
	large	2	6920	30685	167.9	1819	34.5	32504	187.3
Pacman wins	small	1	10	15	19.7	65	9.7	80	19
	medium	2	176	128	115.4	266	94.8	393	128.3
	large	0	34	88	18.1	45	7.4	133	18.9
	large	1	631	2095	223.5	316	66.3	2411	224.6
	large	2	6920	22878	557.8	10772	16.2	33650	566.1

Table 10: Runtimes of MocOCL for the Pacman example (times are given in ms)

`true` for exploring the full state space with a simple OCL query giving a hint about state space generation performance, `Exists Eventually pacman.on.treasure`, the expression indicating that Pacman has found the treasure, and `Exists Eventually pacman.on.treasure and ghosts->forAll(g | g.on <> pacman.on)`, the expression that Pacman has won the game. The two similar expressions are used check the performance if only a part of the state space is needed.

Table 10 shows the benchmark results from five runs for running the evaluation without generating the cause. The evaluation time increases about linearly with the number of states if the model is not accessed from the OCL engine like in the `state space generation` query. There is a bit stronger increase if the model is accessed, like in the `Pacman on treasure` query. The even stronger increase for the `Pacman wins` query comes from the fact that the OCL evaluation engine used does no optimizations on boolean operations. In this case, it always evaluates both parameters for `and` even if the first is false. Thus, the time needed for evaluation of the part of the query checking that Pacman has won the game in a state increases with the number of ghosts. For some queries, the evaluation time differs quite strongly as indicated by the large standard deviation. This might be due to the JVM optimizing some of the evaluation code during the subsequent evaluation. In general, for small state spaces and complex OCL queries, the time for OCL expression evaluation is dominating. For larger state spaces and simple OCL queries, the time for state space generation is dominating. Realistic model sizes, for example including every possible position for Pacman and ghosts and four ghosts, could not be checked.

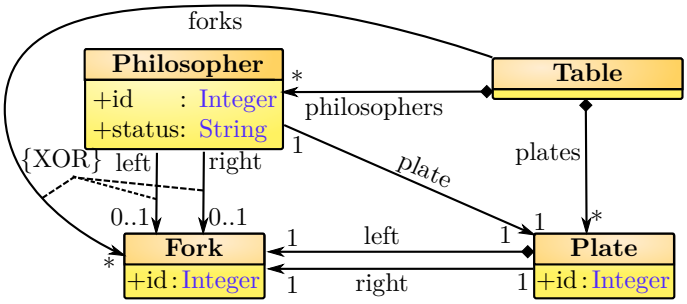


Figure 45: Dining philosophers benchmark metamodel

Dining Philosophers Problem

The dining philosophers problem is described in detail in Section 1.2. There are `n` philosophers sitting around in a table with a plate and two forks in front of them. The forks are all shared between neighbors. The most important property in the dining philosophers problem is a deadlock occurring if every philosopher takes one fork. As model for the dining philosophers scenario we use an adaption of the running example shown in Figure 45. Like in the running example, there are plates on the table with a plate having a left and a right fork. Philosophers are sitting in front of the table. A fork might be on the table or in the left or right hand of a philosopher. The difference to the running example is that IDs have been added for technical reasons and the philosopher status is a string to allow writing shorter queries.

Like the Pacman model's size increases linearly with the number of states, this model's size increases linearly with the number of philosophers. An additional philosopher induces three more objects necessary: a philosopher, a plate and a fork. The effect of the number of philosophers on the initial model size can be seen by looking at Figure 45, an object diagram of six dining philosophers. More philosophers just mean more model elements. Except from having more associations from the table object, the readability is not affected. The number of objects or associations does not change for different states in the state space since no rule changes the number of objects or associations. The state space size increases fast. There are four possible states for each philosopher, thus an upper bound for the state space size is 4^n. A simple lower bound for the state space size is 3^n, since philosophers may be `thinking`, `hungry with no fork` or `hungry with one fork` without interfering with each other in terms of these three states. Thus, in any case the state space grows exponentially with the number of philosophers.

There were two expressions evaluated in the dining philosophers setting. The first expression is `Always Globally true` and generates the whole state space. The second expression is `Always Globally (Exists Next true)`, an expression checking for deadlock–freeness.

Table 11 shows the results for both state space generation and deadlock freeness checking. In all cases, the time for state space generation is dominating since the evaluated queries are rather simple. A big problem is not the runtime, which is under a minute

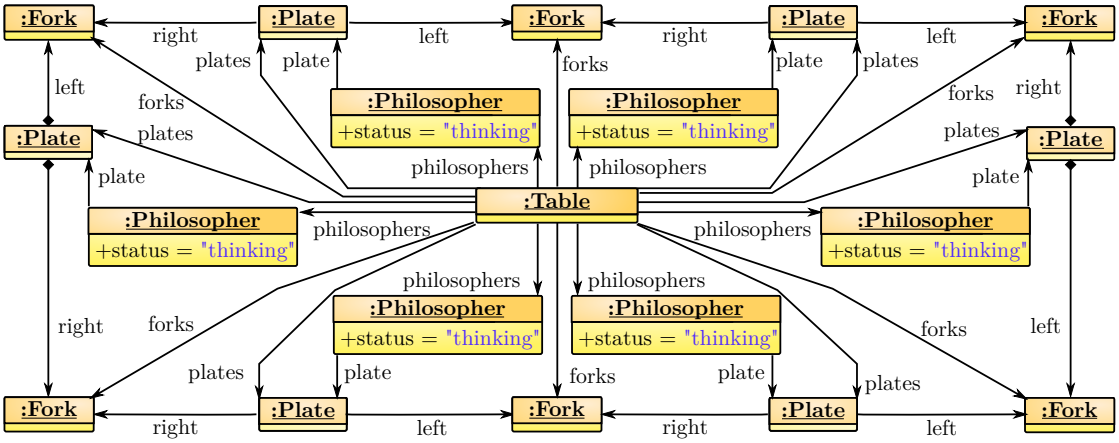


Figure 46: Object diagram of six dining philosophers

even for the largest state space handleable, but the memory requirements. Even though the benchmark was run on a machine with 8 GB of memory, the full state space could not be generated for eight philosophers. The cause generation did not affect runtime much because for many expressions, the cause was generated whether cause generation was enabled or not and later discarded in case of no cause requested. In fact, the results indicate that generating the cause lowers the required time which might be an artifact due to the JVM automatically optimizing code. Still, disabling cause significantly lowers memory requirements and thus made at least the deadlock check for eight philosophers possible.

6.4 Discussion

The results are a cause for cautious optimism. While the objective and subjective reports were somehow contradictory, there were some people interested in playing around with the tool. Positive comments about the tool mostly regarded the ability to see example traces including the state visually. Yet, there were little positive comments on the language alone and no one was interested to try out formulating custom properties. Thus, the language seems to be too difficult to understand to be able to try it out quickly. This, however, might be partly caused by the time constraints. Even though about one hour for a single interview seems to be much, a new language with concepts new to some people and the example itself had to be explained and some time was needed for asking the participants about their opinion on the tool. There was no comparison with other tools/languages like Groove done either. Thus, the question of whether this language might be more usable than existing, implemented, languages could not be answered. While most participants found out that the specified behavior did not match the actual behavior, it cannot be said for sure that this is an improvement over existing tools.

	Phils	Cause	States	gen.time		eval. time		total	
				avg	std	avg	std	avg	std
State space generation	2	no	13	51	13.7	26	7.0	77	14.8
	2	yes	13	22	3.5	17	4.0	39	7.0
	3	no	45	76	19.5	31	7.7	107	26.1
	3	yes	45	44	7.6	21	2.1	65	6.8
	4	no	161	183	3.0	32	.8	215	2.7
	4	yes	161	172	2.4	32	.2	204	2.6
	5	no	573	1115	38.9	93	2.8	1208	38.1
	5	yes	573	1084	2.8	97	2.5	1181	2.5
	6	no	2041	5610	265.8	355	20.0	5965	285.5
	6	yes	2041	5242	43.2	507	36.8	5749	8.9
	7	no	7269	25501	2664.6	4788	1304.7	30289	1374.7
	7	yes	7269	28264	1150.6	1526	8.4	29790	1142.3
	8	no	—	—	—	—	—	—	—
Deadlock freeness	2	no	13	49	16.2	36	5.3	85	19.6
	2	yes	13	22	2.8	21	5.0	43	4.2
	3	no	40	70	15.9	29	5.7	99	17.4
	3	yes	40	45	11.1	47	8.1	92	15.0
	4	no	154	165	7.9	39	1.3	204	7.7
	4	yes	154	142	9.4	47	3.2	189	8.2
	5	no	307	322	15.4	53	1.9	374	16
	5	yes	307	242	4.7	55	1.7	297	3.7
	6	no	713	896	29.4	106	13.4	1002	16.2
	6	yes	713	814	5.2	108	4.8	921	2.7
	7	no	2199	3661	9.4	282	4.4	3943	7.7
	7	yes	2199	3509	4.6	454	1.7	3962	4.4
	8	no	14415	45715	919.0	3048	97.0	48762	823.0

Table 11: Runtimes of MocOCL for the dining philosophers problem (times are given in ms)

Our tool's performance is of multiple magnitudes worse than the one of comparable solutions like Groove [61]. Yet, that was not the focus of this tool and sensible toy examples are possible for evaluating its usability.

6.5 Summary

Our tool MoCOCL was evaluated in terms of usability and performance. For evaluating the usability, a simple variant of the Pacman game using erroneous graph transformations was presented. Eleven test persons were given a short introduction into the language and had to solve tasks including understanding the meaning of cOCL expres-

sions, writing cOCL expressions and interpreting the tool output. Then, they had to give their subjective opinion on the language and the tool. While further evaluation would be necessary to give a definite answer to whether this tool is easier to use than existing solutions, the user study results give rise to some optimism about that. The approach scales quite acceptably. Like GROOVE [61], it runs in super–linear time in the number of states, but since an on–the–fly model checking approach is used, queries only needing a small subset of the state space can be answered fast.

CHAPTER 7

Conclusion

In this thesis, the formal syntax and semantics of cOCL consisting of two temporal OCL extensions are presented. We showed on the one hand, that our CTL extension allows a modeler to check various temporal properties on a system described using an initial model and graph transformations. The selector extension of OCL, on the other hand, provides means to evaluate OCL expressions on interesting system configurations. Further, both extensions were implemented including a web interface allowing to evaluate cOCL expressions and to explore (counter)examples. Two benchmarks demonstrate that our solution scales acceptably and provides sufficient performance for small toy examples. A small user study illustrates the practical usability of cOCL and the web interface.

The research prototype implemented in this thesis shows that integrating model checking with MDE is feasible and makes sense. First evaluation results indicate that our approach is a promising step for integrating model checking in MDE environments.

However, there are many ways of improving the tool's performance in the future.

- *OCL Evaluation*: The Eclipse OCL engine used is really slow. A switch to a faster OCL engine, e.g., the UML–based specification environment (USE) [32], would help in this regard.
- *CTL Evaluation*: A simple, naive, CTL algorithm is used. Consecutive CTL subexpressions could be combined and evaluated in an optimized way.
- *State Space Generation*: The isomorphism checks take a long time. This is worsened by the fact that currently even the mapping between source graph and target graph of a graph transformation has to be retrieved using an isomorphism check. Thus, up to half of the time spent in isomorphism checks could be avoided by having some way to directly get this mapping.
- *State Space Storage*: Currently, every graph of a state is stored explicitly leading to huge memory requirements. This could be avoided by using differential states like in Groove.

- *Evaluation State Storage*: For every transition, the whole evaluation environment is copied resulting in low performance and high memory requirements. There might be ways to reduce the necessity of explicit evaluation environment transformation.

The tool's usability can be improved as well.

- *Eclipse Integration*: Currently, there is only a web interface for the tool making it necessary to specify the transformations and to view the result in completely different environments.
- *Layout Algorithm*: The layout algorithm often leads to suboptimal results. It can be improved, manual intervention might be sensible.
- *Expression Input*: The cOCL expression has to be specified without any automated aid like syntax highlighting and term completion.
- *Relevant Object Display*: For large models, too much unnecessary information is presented. The part of the model displayed could be restricted to the parts relevant for the expression.
- *Path Exploration*: For counterexamples, it might be sensible to explore different paths.

APPENDIX A

Graph Transformations

Graph transformations are used for some definitions and as mechanism to describe the behavior of models throughout this thesis. While a basic understanding of graph transformations is sufficient to get the general idea, we summarize the used concepts of formal graph transformation executions for the interested reader in the following.

Graph transformation is a technique to transform a graph A into a graph B. As models are built upon the same basic concepts, graphs are often used to formalize their structure. EMF models are thus formally described as attributed graphs with inheritance and containment [7]. Similarly, graph transformation is considered as formal counterpart to model transformation. While there are many approaches for graph transformations, in the following we consider the algebraic approaches, namely the single–pushout approach [49] and the double–pushout approach [22]. The general idea of graph transformation is that a transformation rule, called production, is specified by graphs. The left–hand side graph (LHS) describes the search pattern, i.e. the part of the graph which should be changed and the right–hand side graph (RHS) describes the replacement pattern, i.e., the actual change. The mapping of elements of both pattern graphs might also be specified by a graph.

The following definitions are based on Ehrig et al. [22].

Definition 35 (Graph)**.** A *graph* $G = (V, E)$ consists of a set of vertices $v \in V$ and a set of edges $e = (e_s, e_t) \in E$. The functions $s, t : E \to V$ with $s(e) = e_s$, $t(e) = e_t$ return source and target vertices of each edge.

A graph $G_2 = (V_2, E_2)$ is called *subgraph* of a graph $G_1 = (V_1, E_1, s_2, t_2)$ if $V_2 \subseteq V_1$ and $E_2 \subseteq E_1$. □

A simple way to describe the mapping between two graphs is a function relating vertices and edges from one graph to vertices and edges from another graph. This is called a *graph morphism*.

Definition 36 (Graph morphism)**.** Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. Let $G_A = (V_A, E_A)$ be a subgraph of G_1. A (partial) *graph morphism* $m : G_A \to G_2$ consists of a pair $(m_V : V_A \to V_2, m_E : E_A \to E_2)$ of mapping functions mapping vertices and edges from G_1 to G_2. The *domain* of a graph morphism is the subgraph it is defined on, i.e. G_A. These functions are structure preserving, i.e. $f_V(s(e)) = s(f_E(e))$ and $f_V(t(e)) = t(f_E(e))$. A graph morphism is called *total* if its domain is the whole graph, i.e. $G_A = G_1$.

Let $m_1 = (m_{V,1}, m_{E,1}), m_2 = (m_{V,2}, m_{E,2})$ be two graph morphisms. The graph morphism composition $m = m_1 \circ m_2$ is defined as $m = (m_{V_1} \circ m_{V_2}, m_{E_1} \circ m_{E_2})$ and is a graph morphism. □

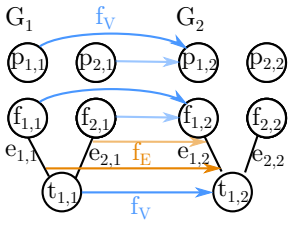


Figure 47: Example graph

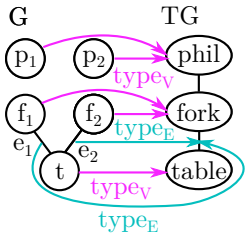


Figure 48: Typed graph G and type graph TG

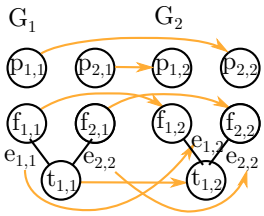


Figure 49: Isomorphism changing philosophers

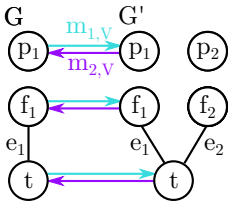


Figure 50: Mono– and epimorphisms m_1, m_2

Example 51 (Graph)**.** The graph $G_{i=\{1,2\}} = (V_{i=\{1,2\}}, E_{i=\{1,2\}})$ with $V_{i=\{1,2\}} = \{p_{1,i}, p_{2,i}, f_{1,i}, f_{2,i}, t_{1,i}\}$, $E_{i=\{1,2\}} = \{e_{1,i} = (t_{1,i}, f_{1,i}), e_{2,i} = (t_{1,i}, f_{2,i})\}$ describes the graphs of a simplified dining philosophers problem with the intuition that there are two forks on the table and two philosophers. The graph morphism $m = (f_V, f_E) : G_1 \to G_2$ with $f_V(p_{1,1}) = p_{1,2}$, $f_V(p_{2,1}) = p_{1,2}$, $f_V(t_{1,1}) = t_{1,2}$, $f_V(f_{1,1}) = f_{1,2}$, $f_V(f_{2,1}) = f_{1,2}$, $f_E(e_{1,1}) = e_{1,2}$, $f_E(e_{2,1}) = e_{2,2}$ is visualized in Figure 47. The partial graph morphism $f' = (f'_V, f'_E) : G_1 \to G_2$ with $f_V(p_{1,1}) = p_{1,2}$, $f_V(f_{1,1}) = f_{1,2}$, $f_V(t_{1,1}) = t_{1,2}$, $f_E(e_{1,1}) = e_{1,2}$ is visualized in the same figure when considering only the fully opaque mappings.

In order to be able to formalize the intuition that the p vertices are philosophers, the f vertices are vertices and the t vertices are tables, a type graph is used. This type graph corresponds to the metamodel in modeling. Mappings between graph and type graph specify the has–type relation.

Definition 37 (Type Graph)**.** A *type graph* is a graph $TG = (V_{TG}, E_{TG})$. A *typed graph* $G^T = (G, type)$ over a type graph TG is a graph with an associated graph morphism $type : G \to TG$. A *typed graph morphism* $m : G_1^T \to G_2^T$ is a graph morphism $m : G_1 \to G_2$ which is type preserving, i.e. $type_2 \circ m = type_1$. □

Example 52 (Type graph). A type graph shown in Figure 48 of the simplified dining philosophers problem is $TG = (V_{TG}, E_{TG})$ with $V_{TG} = \{phil, fork, table\}$, $E_{TG} = \{e_{tg1} = (phil, fork), e_{tg2} = (table, fork)\}$. The corresponding typed graph $G^T = (G, type)$ includes the morphism $type = (type_V, type_E)$ with $type_V(p_1) = type_V(p_2) = phil$, $type_V(f_1) = type_V(f_2) = fork$, $type_V(t) = table$, $type_E(e_1) = type_E(e_2) = (table, fork)$

The definition and algorithms for the graph transformations discussed require some further definitions which are given next.

Definition 38 (Special Morphisms)**.** A morphism $m : B \to C$ is called *monomorphism* if it is left-cancellative, so $m \circ f = m \circ g \Rightarrow f = g \; \forall f, g : A \to B \in Mor_C$. A morphism $m : B \to C$ is called *epimorphism* if it is right–cancellative, so $f \circ m = g \circ m \Rightarrow f = g$ $\forall f, g : A \to B \in Mor_C$. A morphism $m : B \to C$ is called *isomorphism* if there exists an inverse morphism $m^{-1} : B \to A \in Mor_C$ with $i \circ i^{-1} = id_B$ and $i^{-1} \circ i = id_A$. □

In all the examples above, mono–, epi– and isomorphisms correspond to injective, surjective and bijective morphisms, respectively.

Example 53 (Special morphisms). The morphism depicted in Figure 47 is neither mono–, epi– nor isomorphism. The morphism depicted in Figure 49 is an isomorphism, thus as well a mono– and epimorphism. The blue morphism $m_1 : G \to G'$ depicted in Figure 50 is a monomorphism. The purple morphism $m_2 : G' \to G$ in this figure is an epimorphism.

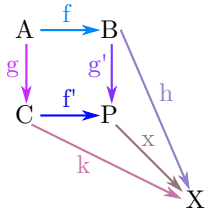


Figure 51: Pushout structure

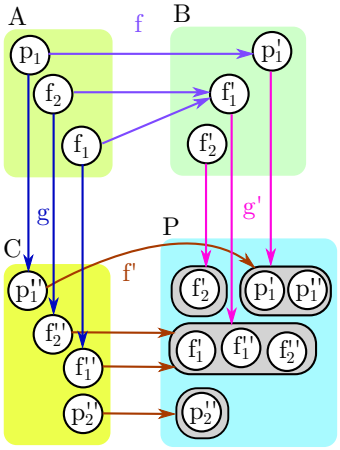


Figure 52: Set push out example

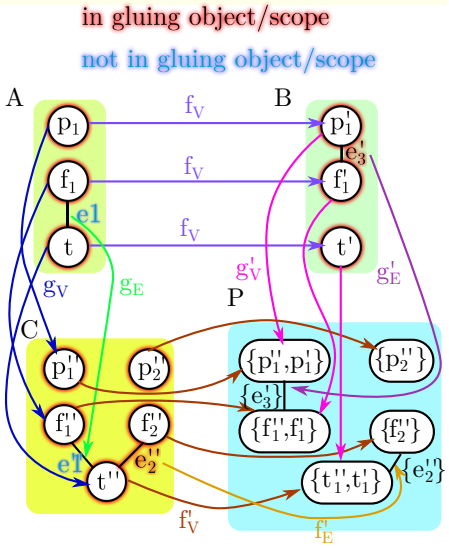


Figure 53: Graph push out example

The first approach we discuss is the *single pushout approach* whose structure is shown in Figure 51. In this approach, a single (partial) morphism f connects the LHS A to the RHS B. A second mapping g given is the mapping from the LHS to the source graph C. It is required to determine the target graph P from the source graph, the LHS, its mapping to the RHS and its mapping to the source graph. The *pushout* determines a sensible target graph.

Definition 39 (Pushout)**.** A *pushout* $C \xrightarrow{f'} P \xleftarrow{g'} B$ over a morphism $f : A \to B$ and a total morphism $g : A \to C$ consists of an object P and two morphisms $f' : C \to P$ and $g' : B \to P$ which commute, i.e. $f' \circ g = g' \circ f$. It has the universal property that for all objects X and morphisms $h : B \to X$, $k : C \to X$ with $k \circ g = h \circ f$ there is a unique morphism $x : P \to X$ with $x \circ g' = h$ and $x \circ f' = k$. This structure is depicted in Figure 51. □

The main special property of the target graph P is that there is exactly one homomorphism x to any other target graph candidate X. This means that the target graph is constructed in a way that it is as large as possible without having elements not mapped by f' or g' and being consistent with $f' \circ g = g' \circ f$. If it is smaller than this, than no homomorphism x exists for the largest such constructed graph. If it contains elements neither mapped by f' nor g', these elements can be mapped to arbitrary elements by x, so x is not unique any more.

Initially, the set of all objects which can be mapped by both $f' \circ g$ and $g' \circ f$ called the gluing object $f \bigtriangledown g$ is determined. This set can only contain elements which both f and g map. But if g maps multiple objects $x_1, x_2, ..., x_n$ to a single object y, then we know that y is either mapped by f' to a single object in P or not mapped. If it is mapped, then $f' \circ g$ is defined for all $x_1, ..., x_n$. If it is not mapped, then $f' \circ g$ is not defined for any of these elements. If f does not map any of $x_1, ..., x_n$, then $g' \circ f$ cannot be defined for this element. Since both $f' \circ g$ and $g' \circ f$ are equal, they must be defined for the same elements. Thus, if f does not map a single element out of $x_1, ..., x_n$, then neither of these elements is in $f \bigtriangledown g$. As well, if a vertex is not mapped, any edge connected to this vertex may as well not be mapped. All elements mapped by both f and g not rejected by both properties are taken.

In the next step, the domains of the morphisms f' and g' are determined. The domains should be as large as possible without conflicting with $g' \circ f = f' \circ g$. Thus, the only objects which may not be mapped are the objects which are mapped by f or g, but cannot be mapped by both $f' \circ g$ and $g' \circ f$. The target graph should be as large as possible, so elements in domains of f' and g' are mapped to distinct elements unless they share a common preimage in f or g because $(g' \circ f)(e) = (f' \circ g)(e)$.

The pushout construction is unique modulo isomorphism. Formally, the pushout can be constructed in four steps [49]:

- Construct the gluing object $f \bigtriangledown g$ as largest subalgebra with $f \bigtriangledown g \subseteq A_f$ (A_f is the domain of the morphism f) where $\forall x \in f \bigtriangledown g, y \in A : f(x) = f(y) \vee g(x) = g(y) \implies y \in f \bigtriangledown g$.
- Construct the domains $C_{f'}$ of f' and $B_{g'}$ of g'. $C_{f'}$ is the largest subalgebra of C contained in $(C - g(A)) \cup g(f \bigtriangledown g)$. Likewise, $B_{g'}$ is the largest subalgebra of B contained in $(B - f(A)) \cup f(f \bigtriangledown g)$.
- Build the *gluing* construction of $\mathrm{P} = (B_{g'} \cup C_{f'})_{\equiv}$ with $x \equiv y \Leftrightarrow \exists z \in f \bigtriangledown g : x = f(z), y = g(z)$[1].

[1]Or any other combination of f and g, but we assume here that x is from B_{g_f} and y is from C_{f_g}

- Construct the *pushout homomorphisms* $f' : C \to P$ with domain $C_{f'}$ which is defined for all $x \in C_{f'}$ by $f'(x) = [x]_{\equiv}$ and likewise, $g' : B \to P$ on scope $B_{g'}$.

In sets, the pushout $C \xrightarrow{f'} P \xleftarrow{g'} B$ of the partial morphism $f : A \to B$ and the total morphism $g : A \to C$ is constructed as follows:

- The gluing object $f \bigtriangledown g$ is constructed iteratively. Initially, $f \bigtriangledown_0 g = A_f$. Then, $f \bigtriangledown_{i+1} g = \{x \in f \bigtriangledown_i g | f^{-1}(f(x)) \cup g^{-1}(g(x)) \subseteq f \bigtriangledown_i g\}$. The gluing object $f \bigtriangledown g = f \bigtriangledown_n g$ if a fixpoint has been reached, i.e. $f \bigtriangledown_n g = f \bigtriangledown_{n+1} g$. The gluing object is the largest common subdomain of the domains of f and g suitable for pushout construction. The gluing object equals to A_f for injective morphisms and A for total morphisms.

- The domain $C_{f'}$ of f' is $(C - g(A)) \cup g(f \bigtriangledown g)$. The domain $B_{g'}$ of g' is $(B - f(A)) \cup f(f \bigtriangledown g)$. If the morphism is total, the scopes are C and B, respectively.

- Then, the pushout graph D is constructed by building equivalence classes[2] of the relevant scopes $P = (B_{g'} \cup C_{f'})_{\equiv}$ with $a \equiv b \Leftrightarrow \exists z \in f \bigtriangledown g : x = f(z), y = g(z)$, i.e. $f^{-1}(x) \cap g^{-1}(y) \neq \varnothing$, i.e. both elements share a common preimage.

Example 54 (Set Pushout). This example illustrates a simple set pushout. As example, consider the set of two forks and one philosopher in Figure 52. Both f and g are total, so the domain of $f \circ g'$ and $g \circ f'$ is A, the domain of f' is C and the domain of g' is B. P contains equivalence classes built from $B \cup C$. All forks f_1, f_2 of set A are mapped to the same fork f'_1 in set B, thus in the pushout P, the forks f'_1, f''_1 and f''_2 build a single class. Fork f'_2, however, has no preimage and thus builds a single equivalence class, like philosopher p''_2. p_1 gets mapped to p'_1 and p''_1, so they form an equivalence class.

Example 55 (Single Graph Pushout). This example illustrates the application of the single pushout approach.

In (typed) graphs, pushouts can be constructed component-wise for their vertex and edge sets except that edges are not in the scope if one of their source or target vertices are not in the scope. Figure 53 shows an example of a graph pushout where an edge is removed from the table to a fork and an edge is added from the fork to a philosopher in $f : A \to B$ while a philosopher and a fork are added in $g : A \to C$.

Initially, the set $\tilde{V}_A = V_{Af} \cap V_{Ag}$ contains all vertices since they are all mapped by both f_G and g_V. The set $E_{Af} \cap E_{Ag}$, however contains no edge since this edge is not mapped by f_E. The scope of B, B_{f_g} includes everything. The preimages of p'_1, f'_1 and t' are all in $\tilde{V}_A$ and e'_3 has no preimage. Likewise, in C_{g_f}, p''_1, f''_1 and t'' are included because their preimages are in $\tilde{V}_A$ and p''_2, f''_2, e''_2 because they have no preimage. e''_1, however, is not included because it has a preimage, e_1, but this preimage is not in $\tilde{V}_A$ because it is not mapped by f_E. p''_1 and p'_1 share a common preimage and thus are in the same

[2] This construction is also called *coequalizer* application.

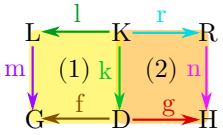


Figure 54: Double pushout

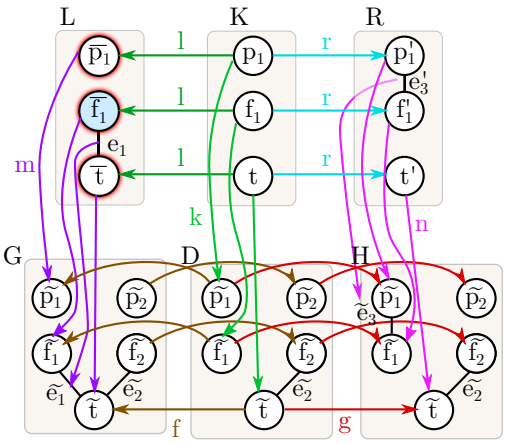


Figure 55: Double push out example

equivalence class. The same applies for f_1'', f_2'' and t', t''. p_2'', f_2'', e_2'', e_3' have no preimage and thus get an own equivalence class each.

The pushout definition is easier if both morphism f and g are total. Then, there is no need to figure out different scopes. The *double pushout* operates on total morphisms only. There are two total, injective, morphisms l and r connecting the LHS L to the RHS R as in $(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$. The intermediate graph K contains all objects which are not changed. L contains some additional objects which are deleted from the source graph, R contains additional objects added to the source graph.

Definition 40 (Double Pushout transformation)**.** A *double pushout transformation rule* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ for transforming graphs G to graphs H consists of (typed) graphs L, K and R and injective (typed) graph morphisms l and r. A direct (typed) graph transformation $G \overset{p,m}{\Rightarrow} H$ is given by the pushouts (1) and (2), a production p and a match $m : L \rightarrow G$.

The *gluing points* GP are those vertices and edges in L not deleted by p: $GP = l(K)$. The *identification points* IP are all vertices and edges where m does not map injectively, i.e. $IP = \{v \in V_L | \exists w \in V_L, w \neq v : m_V(v) = m_V(w)\} \cup \{e \in E_L | \exists f \in E_L, f \neq e : m_E(e) = m_E(f)\}$. The *dangling points* are all vertices of L mapped to a vertex in G having an edge not in the image of m, i.e. $DP = \{v \in V_L | \exists e \in E_G \backslash m_E(E_L) : m_V(v) \in \{s_G(e), t_G(e)\}\}$. □

A production is applicable if the *gluing condition* is fulfilled, i.e. $IP \cup DP \subseteq GP$. If this gluing condition is fulfilled, $G \overset{p,m}{\Rightarrow} H$ is constructed in two steps:

- All corresponding vertices and edges of L not reached from K are deleted from G, so $D = (G \backslash m(L)) \cup m(l(K))$ so that $G = L +_K D$, i.e. G is the corresponding pushout.

- All nodes and edges in R not in K are added, i.e. $H = D \cup (R \backslash r(K))$ such that $H = R +_K D$

Example 56 (Double Pushout). Let us again consider the rule handing a fork from the table to a philosopher depicted in Figure 55. We define the match to be $m(\overline{p}_1) = \tilde{p}_1, m(\overline{f}_1) = \tilde{f}_1, f(\overline{t}) = \tilde{t}$. At first we want to check whether the production is applicable. The gluing points are all vertices and edges having a pre–image in K which are thus $\overline{p}_1, \overline{f}_1$ and $\overline{t}$. There are no identification points because the mapping m is injective. The edge $\tilde{e}_2$ is the only edge not mapped by m, thus $\tilde{t}$ is the only dangling point. Thus, $DP \cup IP \subseteq GP$ and the production is applicable. For constructing D, we have to remove $\tilde{e}_1$ since it has a preimage in L but none of that preimages has a preimage in K. Then, we see e'_3 is in R but not in $r(K)$, thus we add it to D.

APPENDIX B

Questionnaire

The questionnaire used in the user study contains four sections. At first, there is a self assessment considering various topics considered as important for the user study. Then, an introduction to the problem is given. Since the user study was done in interviews, this introduction is not extensive. Depending on the participant's experience in various topics, these topics were explained in more or less detail during the user study. Both pages 2 and 3 also served as reference during the comprehensibility evaluation. The fourth page contains tasks the participants were encouraged to solve using the tool. The last page contained the subjective evaluation with lots of space for comments and recommendations. Note that the language syntax was changed after the user study. The operator `sometimes` equals the operator `Exists`, all other operators are simply written in full lowercase instead of being capitalized.

COCL Questionnaire

1 Self assessment of previous knowledge

Please tick the boxes best matching your experience.

Scope	Type	None	Little	Signi-ficant	Expert
Modelling	Reading structural models	☐	☐	☐	☐
	Writing structural models	☐	☐	☐	☐
	Reading behavioural models	☐	☐	☐	☐
	Writing behavioural models	☐	☐	☐	☐
	Reading OCL queries	☐	☐	☐	☐
	Writing OCL queries	☐	☐	☐	☐
Graph transformations	Reading graph transformations	☐	☐	☐	☐
	Writing graph transformations	☐	☐	☐	☐
	Executing graph transformations (using tools)	☐	☐	☐	☐
Model Checking	Propositional logic (e.g. $p \wedge q$)	☐	☐	☐	☐
	Predicate logic (e.g. $\forall x : P(x) \rightarrow Q(s)$)	☐	☐	☐	☐
	Temporal logics (e.g. $AG(p \rightarrow X\, q)$)	☐	☐	☐	☐
	CTL	☐	☐	☐	☐

I have used any model checkers: Yes ☐ No ☐.

The model checkers I used are:

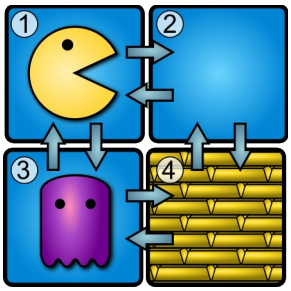
Main field of interest: ☐ Software Engineering
☐ Modeling
☐ Logic
☐ Other: _____

I am a practitioner ☐ researcher ☐

Reference:

None	Never heard of that topic, never used it
Little	Heard of it once or used it, but forgotten most
Significant	Limited knowledge/use of basic concepts
Expert	In-depth knowledge/use

2 Problem introduction



2.1 Setting

In the following we will consider a variant of the game Pacman modelled using graph transformation. The game is over, i.e. no further turns are possible, if Pacman has found the treasure or collides with a ghost.

The expression `self.fields->exists(f | f.treasure)` for example indicates that there must be a field with a treasure

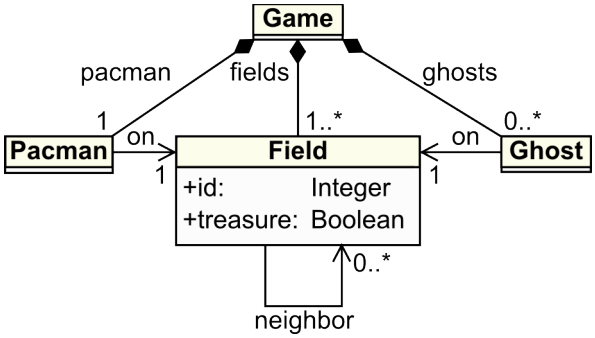


Figure 3: Pacman class diagram

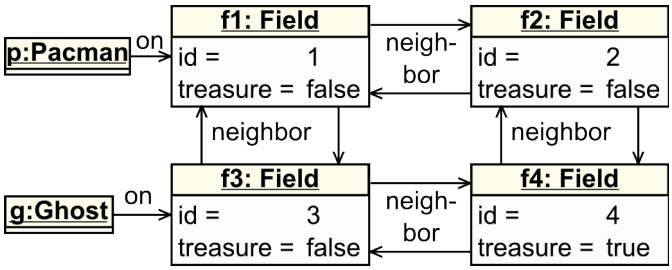


Figure 2: Partial object diagram of the initial field. All objects are contained in the game object.

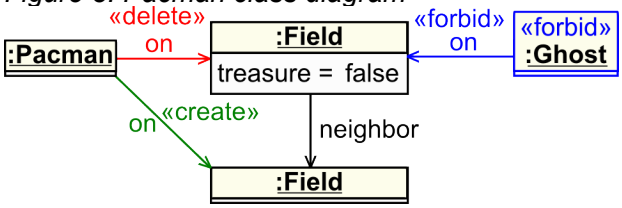


Figure 4: Pacman move turn rule

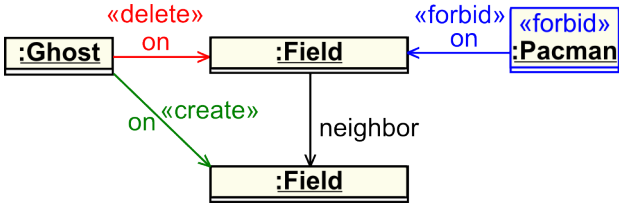


Figure 5: Ghost move turn rule

2.2 Language concepts

The language is an extension of OCL. The language extends OCL by structures of the form `((always|sometimes) (globally|eventually|next) <OCL Condition>)` and `((always|sometimes) <OCL condition #1> (until|unless) <OCL condition#2>)` which can be used everywhere an OCL boolean could be used.

Expression	Meaning
`always globally <#1>`	Surely, #1 is true all the time
`sometimes globally <#1>`	There is a possibility that #1 is true all the time (there is at least a certain rule sequence where it is true all the time, but it might be false for other rule sequences)
`always eventually <#1>`	Surely, #1 is true eventually in the future (but it is unknown when)
`sometimes eventually <#1>`	There is a possibility that #1 is true eventually in the future
`always next <#1>`	Surely, #1 is true after a single turn or there is no next turn
`sometimes next <#1>`	There is a possibility that #1 is true after a single turn
`always <#1> until <#2>`	Surely, #2 is true somewhere in the future and before it is true, #1 is true.
`sometimes <#1> until <#2>`	There is a possibility that #2 is true somewhere in the future and before #2 is true, #1 is true
`always <#1> unless <#2>`	Surely, either #2 is true somewhere in the future and before it is true, #1 is true or #1 is true all the time
`sometimes <#1> unless <#2>`	There is a possibility that either #2 is true somewhere in the future and before it is true, #1 is true or #1 is true all the time

All these statements may of course be mixed, for example **`sometimes next sometimes next <OCL condition>`** may be used to refer to something two rule executions away.

2.3 Example

We might want to ask questions about the model, for example, whether Pacman will find the treasure in all cases. Pacman finds the treasure by moving on a field with treasure attribute true. For example, Pacman might find the treasure by moving two times: From field 1 to field 2, from field 2 to field 4 as shown in Fig. 6.

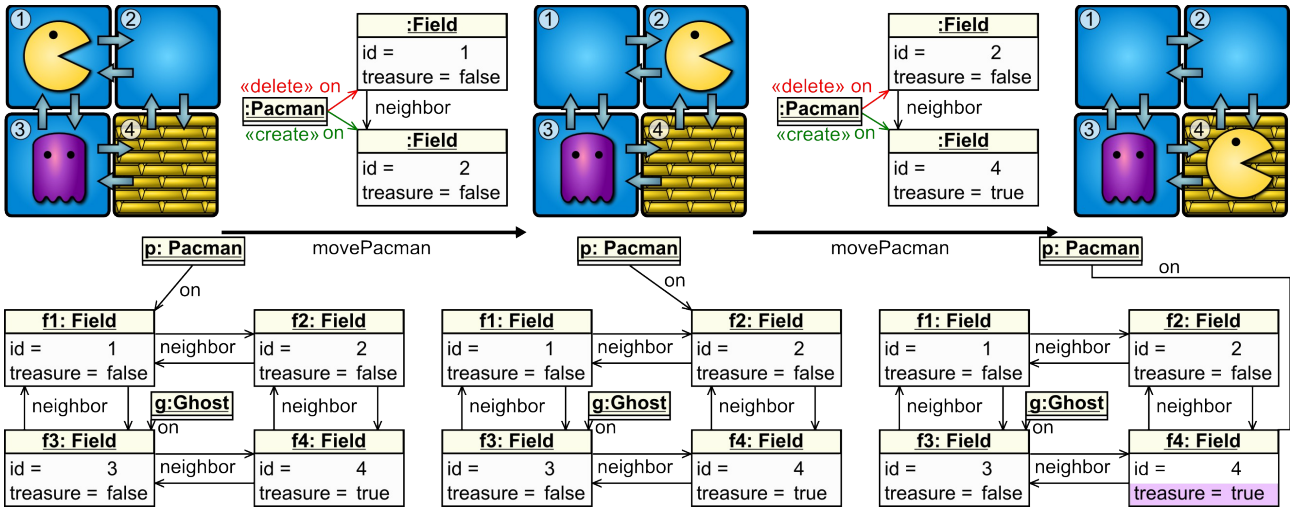


Figure 6: Pacman finds the treasure by moving two times. Afterwards, he cannot move any longer since the treasure-attribute of the field where he is on is true (cf. Fig. 4)

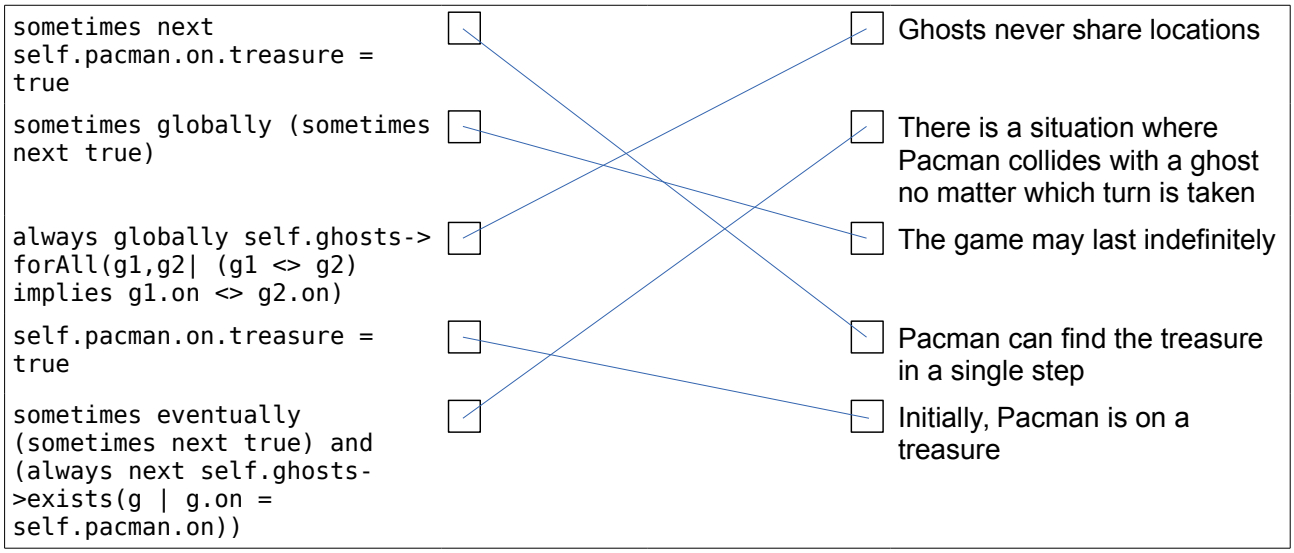
We can formulate many natural language expressions into equivalent COCL expressions

Natural language expression	**COCL expression**
Initially, there is a field containing a treasure	`self.fields->exists(field | field.treasure)`
The game is over/not over	`always next false/sometimes next true`
Pacman will find the treasure in all cases.	`always eventually (self.pacman.on.treasure = true)`
It is possible that a ghost reaches a treasure.	`sometimes eventually self.ghosts->exists (ghost | ghost.on.treasure = true)`
It is possible that the game is over.	`sometimes eventually (always next false)`
The game will surely be over sometimes.	`always eventually (always next false)`
The set of all ghosts imposing a potential danger to Pacman	`self.ghosts->select(g | sometimes eventually g.on = self.pacman.on)`
If two ghosts share the same location, their location will be different after the next turn	`always globally self.ghosts->forAll(g1,g2 | g1 <> g2 implies ((g1.on = g2.on) implies ((sometimes next true) and (always next g1.on <> g2.on))))`
If the treasure is next to Pacman, he can find it the next turn	`always globally self.pacman.on.neighbour->exists(field | field.treasure) implies (sometimes next self.pacman.on.treasure)`
As long as not all fields next to Pacman are occupied by ghosts, there is a possibility that the game is not over the next turn	`always globally self.pacman.on.neighbour->exists(field | self.ghosts->forAll(g | field <> g.on) implies (sometimes next (sometimes next true)))`
As long as the game is not over, every ghost may move to at least two different positions	`always self.ghosts->forAll(g | g.on.neighbour->select(field | sometimes next g.on = field)->size()  >= 2) unless (always next false)`

3 Comprehensibility evaluation

We now want to know whether it is possible to understand the meaning of COCL expressions.

1. **Please connect the corresponding COCL expressions to their natural language equivalent.**



2. **Please try to find out the natural language equivalent of the following COCL expressions and give an example game or other explanation for the result**

a) `sometimes next sometimes next self.pacman.on.treasure = true`

Pacman might find the treasure in two steps. This is true because Pacman might first go to field 2, then to field 4.

b) `sometimes eventually self.ghosts->exists(ghost | ghost.on = self.pacman.on)`

The game might end with Pacman losing. This is true because Pacman might at first go to field 3 to the ghost.

c) `self.ghosts->exists(g | g.on.treasure) and sometimes eventually (self.pacman.on.treasure and self.ghosts->forAll(ghost | ghost.on <> self.pacman.on))`

While initially a ghost is on the treasure, Pacman might still win the game. This is true since initially, no ghost is on a treasure.

3. **Please try to find out an equivalent COCL expression for these properties and and give an example game or other explanation for the result**

a) After three turns, Pacman maybe has found the treasure.

sometimes next sometimes next sometimes next self.pacman.on.treasure = true

This is true since Pacman might at first move to the treasure by going to field 2, then to field 4 and then the ghost might move.

b) A rule of the game: The game is over if Pacman has found the treasure.

always globally self.pacman.on.treasure implies (always next false)

This is false, see the result of 3a

4 Subjective Evaluation

Task	Easy	Medium	Difficult	Infeasible
Reading COCL expressions	☐	☐	☐	☐
Writing COCL expressions	☐	☐	☐	☐
Interpreting the tool output	☐	☐	☐	☐

In which way was the tool helpful/non-helpful for understanding the meaning of COCL expression results?

Do you have any recommendations regarding further language concepts (e.g. for easier specification)?

Do you have any recommendations regarding the tool (e.g. for enhancing usability)?

Do you have any further comments regarding COCL, the tool or this questionnaire?

Bibliography

[1] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *ACM Computing Surveys*, 14(2):159–192, 1982.

[2] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *MoDELS*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.

[3] Colin Atkinson and Thomas Kühne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press Cambridge, 2008.

[5] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 35–42. FMCAD Inc, 2010.

[6] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[7] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software & Systems Modeling*, 11(2):227–250, 2012.

[8] Stefan Rosenegger Boris Ljepoja, Thomas Pfenning. Der Pentiumbug. `http://www5.in.tum.de/lehre/seminare/semsoft/unterlagen_02/pentiumbug/website/`.

[9] Julian Bradfield, Juliana Filipe Küster, and Perdita Stevens. Enriching OCL Using Observational Mu-Calculus. In *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 203–217. Springer, 2002.

[10] Franck Budinsky, David Steinberg, and Raymond Ellersick. *Eclipse Modeling Framework : A Developer's Guide*. Addison-Wesley Professional, 2003.

[11] Albert F. Case. Computer-aided software engineering (CASE): technology for improving software development productivity. *ACM SIGMIS Database*, 17(1):35–43, 1985.

[12] Alessandro Cimatti. Industrial Applications of Model Checking. In *Modeling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 153–168. Springer, 2001.

[13] Edmund M. Clarke. 25 Years of Model Checking. chapter The Birth of Model Checking, pages 1–26. Springer, 2008.

[14] Edmund M. Clarke and E. Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[15] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM, 1977.

[16] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[17] Maria del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. Debugging UML designs with model checking. *Journal of Object Technology*, 1(2):101–117, 2002.

[18] Edsger Wybe Dijkstra. *Notes on structured programming*. Technological University Eindhoven Netherlands, 1970.

[19] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. Towards model checking OCL. In *Proceedings of the ECOOP Workshop on Defining a Precise Semantics for UML*, 2000.

[20] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[21] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE '99)*, pages 411–420. IEEE, 1999.

[22] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of algebraic graph transformation*, volume 373. Springer, 2006.

[23] Technische Universiteit Eindhoven. μ–calculus — mcrl2 201310.0 documentation. `http://www.mcrl2.org/release/user_manual/language_reference/mucalc.html`, 2011.

[24] E. Allen Emerson. 25 Years of Model Checking. chapter The Beginning of Model Checking: A Personal Perspective, pages 27–45. Springer, 2008.

[25] E. Allen Emerson and Joseph Y. Halpern. «Sometimes» and «Not Never» Revisited: On Branching Versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, 1986.

[26] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. On model checking for the μ-calculus and its fragments. *Theoretical Computer Science*, 258(1–2):491 – 522, 2001.

[27] Thorsten Fischer, Jörg Niere, and Lars Torunski. *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*, chapter Story–Driven Modeling (SDM). Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, 1998.

[28] Limor Fix. 25 Years of Model Checking. chapter Fifteen Years of Formal Property Verification in Intel, pages 139–144. Springer, 2008.

[29] International Organization for Standardization and International Electrotechnical Comission. Information Technology—Syntactic Metalanguage—Extended BNF 1.0. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`, December 1996.

[30] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Workshop on the Future of Software Engineering (FOSE '07)*, pages 37–54. IEEE Computer Society, 2007.

[31] Sebastian Gabmeyer, Petra Brosch, and Martina Seidl. A Classification of Model Checking-Based Verification Approaches for Software Models. In *Proceedings of the 2013 STAF Workshop on Verification of Model Transformations (VOLT '13)*, 2013.

[32] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1–3):27 – 34, 2007. Special issue on Experimental Software and Toolkits.

[33] Kanchi Gopinath, Jon Elerath, and Darrell Long. Reliability Modelling of Disk Subsystems with Probabilistic Model Checking. Technical report, UCSC-SSRC-09-05, University of California, Santa Cruz, 2009.

[34] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. The Formal Specification Language mCRL2. In *Proceedings of the 2007 Dagstuhl Seminar on Methods for Modelling Software Systems (MMOSS '07)*, volume 06351 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[35] Object Management Group. Model Driven Architecture (MDA) Guide V1.0.1. `http://www.omg.org/mda/`, January 2006.

[36] Object Management Group. Object Constraint Language (OCL) V2.2. http://www.omg.org/spec/OCL/2.2/, February 2010.

[37] Object Management Group. OMG Meta Object Facility (MOF) Core Specification V2.4.1. http://www.omg.org/spec/MOF/2.4.1/, August 2011.

[38] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure V2.4.1. http://www.omg.org/spec/UML/2.4.1/, August 2011.

[39] Object Management Group. Foundational UML (FUML). http://www.omg.org/spec/FUML/1.1, August 2013.

[40] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3):287–313, 1996.

[41] Reiko Heckel. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, 2006.

[42] Bilal Kanso and Safouan Taha. Temporal Constraint Support for OCL. In *Software Language Engineering*, volume 7745 of *LNCS*, pages 83–103. Springer, 2013.

[43] Harmen Kastenberg and Arend Rensink. Model Checking Dynamic States in GROOVE. In *Model Checking Software*, volume 3925 of *LNCS*, pages 299–305. Springer, 2006.

[44] Alexander Knapp. Hugo/RT. http://www.pst.informatik.uni-muenchen.de/projekte/hugo/#Publications, August 2008.

[45] Alexander Knapp and Jochen Wuttke. Model Checking of UML 2.0 Interactions. In *Proceedings of the 2006 MoDELS Workshops*, volume 4364 of *LNCS*, pages 42–51. Springer, 2006.

[46] Christian Krause and Stefan Neumann. Instance-Aware Model Checking of Graph Transformation Systems using Henshin and mCRL2. https://www.eclipse.org/henshin/documents/henshin_mcrl2.pdf, 2010.

[47] Robert P. Kurshan. 25 Years of Model Checking. chapter Verification technology transfer, pages 46–64. Springer, 2008.

[48] Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and Jin Song Dong. A formal semantics for complete UML state machines with communications. In *Integrated Formal Methods*, volume 7940 of *LNCS*, pages 331–346. Springer, 2013.

[49] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1–2):181 – 224, 1993.

[50] Luis Mandel and Maria Victoria Cengarle. On the expressive power of the Object Constraint Language OCL. http://www.fast.de/projeckte/forsoft/ocl, 1999.

[51] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[52] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Communications of the ACM*, 53(2):58–64, 2010.

[53] John Mullins and Raveca Oarga. Model Checking of Extended OCL Constraints on UML Models in SOCLe. In *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *LNCS*, pages 59–75. Springer, 2007.

[54] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

[55] Norbisrath, Ultrich and Zündorf, Albert and Jubeh, Ruben. *Story Driven Modeling*. CreateSpace Independent Publishing Platform, 2013.

[56] Frédéric Painchaud, Damien Azambre, Matthieu Bergeron, John Mullins, and Raveca M Oarga. Socle: Integrated design of software applications and security. Technical report, DTIC Document, 2005.

[57] Carl Adam Petri and Wolfgang Reisig. Petri net. `http://www.scholarpedia.org/article/Petri_net`, 2008.

[58] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, 1977*, pages 46–57, 1977.

[59] Henshin Project. Henshin. `https://www.eclipse.org/henshin`.

[60] Arend Rensink, Iovka Boneva, Harmen Kastenberg, and Tom Staijen. User manual for the groove tool set. *Department of Computer Science, University of Twente*, 2010.

[61] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In *Graph Transformations*, volume 3256 of *LNCS*, pages 226–241. Springer, 2004.

[62] Mark Richters and Martin Gogolla. OCL: Syntax, Semantics, and Tools. In *Object Modeling with the OCL*, volume 2263 of *LNCS*, pages 42–68. Springer, 2002.

[63] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31, 2006.

[64] Ákos Schmidt and Dániel Varró. CheckVML: A Tool for Model Checking Visual Modeling Languages. In *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *LNCS*, pages 92–95. Springer, 2003.

[65] Bran Selic. What will it take? A view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.

[66] Moshe Vardi. Branching vs. Linear Time: Final Showdown. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 1–22. Springer, 2001.

[67] Jos Warmer, Anneke Kleppe, Tony Clark, Anders Ivner, Jonas Högström, Martin Gogolla, Mark Richters, Heinrich Hussmann, Steffen Zschaler, and Simon Johnston. Response to the UML 2.0 OCL RfP-Revised Submission. `http://deptinfo.cnam.fr/Enseignement/CycleSpecialisation/MAI/OCL_2.pdf`, 2003.

[68] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann Publishers Inc., 2009.

[69] Paul Ziemann and Martin Gogolla. An Extension of OCL with Temporal Logic. In *Critical Systems Development with UML*, pages 53–62. Technical Report, Technical University of Munich, 2002.

[70] Paul Ziemann and Martin Gogolla. An OCL Extension for Formulating Temporal Constraints. Technical report, Universität Bremen, 2003.

[71] Paul Ziemann and Martin Gogolla. OCL Extended with Temporal Logic. In *Perspectives of System Informatics*, volume 2890 of *LNCS*, pages 351–357. Springer, 2003.