Tracing Properties of UML and OCL Models with Maude

Francisco Durán
ETSI Informática. Universidad de Málaga, Spain

Martin Gogolla
University of Bremen, Germany

Manuel Roldán
ETSI Informática. Universidad de Málaga, Spain

The starting point of this paper is a system described in form of a UML class diagram where system states are characterized by OCL invariants and system transitions are defined by OCL pre- and postconditions. The aim of our approach is to assist the developer in learning about the consequences of the described system states and transitions and about the formal implications of the properties that are explicitly given. We propose to draw conclusions about the stated constraints by translating the UML and OCL model into the algebraic specification language and system Maude, which is based on rewrite logic. We will concentrate in this paper on employing Maude’s capabilities for state search. Maude’s state search offers the possibility to describe a start configuration of the system and then explore all configurations reachable by rewriting. The search can be adjusted by formulating requirements for the allowed states and the allowed transitions.

1 Introduction

In the last years, model-driven development has become the topic of various research activities. The employment of models in all phases of software development and for different purposes is said to offer a higher level of abstraction than traditional code-centric development. Models should concentrate on crucial properties of a system to be developed or to be documented. Therefore, particular attention must be paid to checking such properties in order to guarantee software quality. For example, one is interested in verifying whether the formulated model properties are consistent, i.e., there are no contradictions between the properties, or one wants to check whether some further property is implied by other fixed properties. This paper will basically use UML and OCL to formulate models and properties in form of OCL constraints. OCL offers to express properties regarding the system states by invariants and properties concerning system transitions by pre- and postconditions. Thus static and dynamic aspects can be handled.

Given a UML and OCL model of a system employing invariants and pre- and postconditions as the central description means, the developer frequently wants to know about the consequences on the described system states and transitions and about the formal implications of the properties that are explicitly given. For example, given some invariant and some state, one wants to check whether another state, where a further invariant does hold, can be reached by valid transitions or not. Concerning operation pre- and postconditions, the developer could wish to see all valid operation calls in a given state which satisfy the preconditions. Another question is whether from a given start state another explicitly given end state can be reached by valid operations such that all intermediate states are respecting the invariants. Questions like these and further ones will be handled in our approach. For example, in our approach it is also
possible to ask for all sequences of messages and for all objects which have to be created in order to reach a state satisfying specific OCL constraints.

On the technical level, we want to draw conclusions about the stated constraints by translating the UML and OCL model into the algebraic specification language and system Maude. Maude is based on rewrite logic, is a well-established language, and offers sophisticated tools for the analysis of specifications [4]. For example, Maude allows the developer to check the satisifiability of LTL formulas using its model checker, or to prove system properties with the inductive theorem prover ITP. However, we will concentrate in this paper on employing Maude’s capabilities for reachability analysis: we assume invariants are represented by state predicates, operations by Maude rules and pre- and postconditions by predicates as well. Basically, Maude’s state search functionality offers the possibility to describe a start configuration of the system and then explore all configurations reachable from it by rewriting (or a finite subset of all reachable configurations by imposing a maximal depth on the graph of all possible configurations). The search can be adjusted by formulating requirements for the allowed states and the allowed transitions. To represent UML model in Maude and to evaluate OCL expressions on such models we use mOdCL [11].

We assume the reader is familiar with the Maude representation of classes, objects and configurations as, for example, described in [4]. We nevertheless provide a short description of the required Maude features in Section 3. The rest of this paper is structured as follows. In Section 2 we introduce our running example. Section 3 discusses the mOdCL representation of OCL types, user-defined UML classes, and OCL constraints. Section 4 concentrates on system dynamics by introducing the general approach to handle operation calls. Section 5 shows how to exploit properties of the model by employing the Maude state search. Section 6 finishes with concluding remarks and a brief discussion of future work.

2 Running Example

Our running example describes a simple marriage world in which persons can get married and can get divorced. In UML terms, we have a class Person and two enumerations Gender (with literals female, male) and CivilStatus (with literals single, married, divorced). In the Person class we have attributes civstat, gender, wife and husband.

```plaintext
class Person
attributes
civstat:CivilStatus  gender:Gender
wife:Set(Person)    husband:Set(Person)
end
```

For the example, we have decided to present the spouses with set-valued attributes, because we want to show how to use OCL expressions to avoid situations like polygamy and homosexual marriage, and because we wanted to avoid attributes being undefined. If we would have single-valued attributes, e.g., wife:Person, then an unmarried male would be represented with wife being equal to undefined, whereas in our model this is represented as wife being equal to the empty set. In the Person class there are two operations for marrying and divorcing explained in more detail below. We have developed and checked our example with the UML and OCL tool USE [6]. See Figure 1.

As shown in the following, the states of the model are restricted by a number of OCL invariants and the transitions in the model, the operation calls, are required to obey certain operation contracts in form of OCL pre- and postconditions.
A person should not be married to a person of the same gender \(\text{femaleHasNoWife}\) and \(\text{maleHasNoHusband}\), a person can have at most one spouse \(\text{atMostOneHusband}\) and \(\text{atMostOneWife}\), and marriage is a symmetric association \(\text{wifeHusbandInverse}\). We can express these invariants for the Person class in OCL:

\[
\begin{align*}
\text{inv femaleHasNoWife: gender=female} & \implies \text{wife} \rightarrow \text{isEmpty}() \\
\text{inv maleHasNoHusband: gender=male} & \implies \text{husband} \rightarrow \text{isEmpty}() \\
\text{inv atMostOneHusband: husband} & \rightarrow \text{size()<=1} \\
\text{inv atMostOneWife: wife} & \rightarrow \text{size()<=1} \\
\text{inv wifeHusbandInverse:} & \\
& \left(\text{wife} \rightarrow \text{notEmpty()} \implies \text{wife}.\text{husband} \rightarrow \text{includes}(\text{self})\right) \text{ and } \\
& \left(\text{husband} \rightarrow \text{notEmpty()} \implies \text{husband}.\text{wife} \rightarrow \text{includes}(\text{self})\right)
\end{align*}
\]

In order to guarantee consistency of the attribute values, one could additionally state the invariant \(\text{civstat=married} = (\text{wife} \rightarrow \text{notEmpty()} \text{ or } \text{husband} \rightarrow \text{notEmpty()}\).

Class Person has a marry\((a\text{Spouse}:\text{Person})\) operation. If (a) the argument \(a\text{Spouse}\) of the operation is defined\(^1\), (b) both the person receiving the message and the argument \(a\text{Spouse}\) in the message are not married, and (c) both persons have different gender, then the operation results in a state where both persons are married. The operation can be characterized as follows in OCL:\(^2\)

\[
\begin{align*}
\text{pre aSpouseDefined: aSpouse.isDefined} \\
\text{pre aSpouseDefined: aSpouse.isDefined} \\
\end{align*}
\]

\(^1\)In OCL, there is an exception element representing undefined; it is possible to pass this undefined element as an argument to a marry call.

\(^2\)Given the first part of the consequent \(\text{husband=Set\{aSpouse\}}\) in the \text{femaleHasMarriedHusband} constraint, one could simplify its second part to \(a\text{Spouse}.\text{civstat}=\text{married}\) assuming that the formal operation param-
Two married persons can get divorced using the `divorce()` operation. This operation can be specified in OCL as follows:

```ocl
pre isMarried: civstat=married
post isDivorced: civstat=divorced
post husbandDivorced: gender=female implies husband->isEmpty() and husband@pre.civstat->forall(cs|cs=divorced)
post wifeDivorced: gender=male implies wife->isEmpty() and wife@pre.civstat->forall(cs|cs=divorced)
```

### 3 The mOdCL Representation of Class Diagrams

The representation of UML models used in mOdCL is described in detail in [11]. This representation is inspired by the representation of object-oriented modules in Maude, as those used in Maude-based metamodeling frameworks such as MOMENT [1], Maudeling [12], and e-Motions [10]. In these approaches, class diagrams are represented as Maude object-oriented modules, and system states as configurations of objects. The representation used in mOdCL has its own particularities, mainly in the representation of attributes, links and methods, and of course in the support of OCL. In comparison to other Maude-like approach to UML and OCL [2, 5], mOdCL is the only Maude implementation providing support for pre- and postconditions, and the only one supporting the dynamic validation of OCL constraints.

We briefly describe in the following sections the representation of class diagrams and OCL constraints used in mOdCL illustrating it with our running example.

#### 3.1 Object-Oriented Programming in Maude

Maude [3, 4] is a high-level language and a high-performance system that supports membership equational logic and rewriting logic specification and programming of systems.

Membership equational logic [8], a Horn logic whose atomic sentences are equalities \( t = t' \) and membership assertions of the form \( t : S \), stating that a term \( t \) has sort \( S \). Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and the definition of partial functions with equationally defined domains.

Rewriting logic [7] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In rewriting logic, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification \((\Sigma, E)\), where \( \Sigma \) is a signature of sorts (types) and operations, and \( E \) is a set of equational axioms. The dynamics of a system in rewriting logic is then specified by rewrite rules of the form \( t \rightarrow t' \), where \( t \) and \( t' \) are \( \Sigma \)-terms. This rewriting happens

eter aSpouse cannot be changed by the operation. And similarly for the maleHasMarriedWife constraint. However, we prefer in our example the explicit formulation.
modulo the equations $E$, describing in fact local transitions $[t]_E \rightarrow [t']_E$. These rules describe the local, concurrent transitions possible in the system, i.e. when a part of the system state fits the pattern $t$ (modulo the equations $E$) then it can change to a new local state fitting pattern $t'$.

In Maude, concurrent object-oriented systems can be naturally modeled in Maude as multisets of objects and messages, loosely coupled by some suitable communication mechanism. The basic sorts needed to describe an object system are: \texttt{Object}, \texttt{Msg} (messages), and \texttt{Configuration}.

Given a class $C$ with attributes $a_1:S_1$, ..., $a_n:S_n$, where $a_i$ are attribute identifiers and $S_i$ are the sorts of the corresponding attributes, objects of such a class $C$ are then record-like structures of sort \texttt{Object} of the form $<O:C|a_1:v_1,...,a_n:v_n>$, where $O$ is the (unique) identifier of the object, and $v_i$ are the current values of its attributes.

Objects can interact in a number of different ways, including message passing. Messages are declared in Maude as operations of sort \texttt{Msg}.

To represent objects, additional sorts are available: \texttt{Oid} (object identifiers), \texttt{Cid} (class identifiers), \texttt{Attribute} (named elements of objects’ states), and \texttt{AttributeSet} (comma-separated multisets of attributes). Thus, for example, for a class \texttt{Account} with a single attribute \texttt{balance} of sort \texttt{Int}, and messages \texttt{withdraw} and \texttt{transfer} we have the following declarations:

\begin{verbatim}
sort Account.
sort Account < Cid.
op Account : -> Account [ctor].
op balance : _ : Int -> Attribute [ctor].
op withdraw : Oid Int -> Msg [ctor].
op transfer : Oid Oid Int -> Msg [ctor].
\end{verbatim}

Thus, given the predefined operator

\begin{verbatim}
op <:_:|> : Oid Cid AttributeSet -> Object [ctor].
\end{verbatim}

we can represent an account object with identifier $a$ with balance 5 as a term

\begin{verbatim}
< a : Account | balance : 5 >
\end{verbatim}

and a message to $a$ to withdraw 3 as

\begin{verbatim}
withdraw(a, 3)
\end{verbatim}

In a concurrent object-oriented system, the concurrent state, which is called a \textit{configuration}, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting using rules that describe the effects of the communication events of objects and messages. The predefined sort \texttt{Configuration} represents configurations of Maude objects and messages, with \texttt{none} as empty configuration and the empty syntax operator \texttt{__} as union of configurations.

\begin{verbatim}
sort Configuration.
sort Object Message < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
[ctor assoc comm id: none].
\end{verbatim}

Thus, such rewrite rules define transitions between configurations, and their general form is:

\begin{verbatim}
crl [r] : 
< O_1 : C_1 | atts_1 > ... < O_n : C_n | atts_n >
\end{verbatim}
Tracing OCL with Maude

Roles end as constants of the mOdCL sort:

\[ C \]

The name of a given class

\[ Oid \]

Role ends with multiplicity 1 are represented as attributes of sort

\[ \text{Oid} \]

as attributes of sort

\[ \text{Set} \]

Types are then declared as constants of the respective types. E.g., the declarations for

\[ \text{CivilStatus} \]

are declared as enumerations by declaring them as subsorts of

\[ \text{Enum} \]

In OCL, the

\[ \text{Sequence} \]

Set

\[ \text{Object identifiers}. \]

OCL predefined collection types are mapped into sorts

\[ \text{Enum}, \text{Set}, \text{Bag}, \text{OrderedSet}, \text{and Sequence}. \]

All these sorts are subsorts of \[ \text{OclType} \].

Classes are represented following the standard representation of classes in Maude (see Section 3 and 4).

The Attribute sort provides the syntax for attributes and role ends in the objects, given by its name,

\[ \text{AttributeName} \]

and its type, of sort \[ \text{OclType} \].

The name of a given class \[ C \] is represented as a constant \[ C \] of the Maude sort \[ \text{Cid} \], and its attributes and role ends as constants of the mOdCL sort \[ \text{AttributeName} \]. Thus, for the \text{Person} class we add declarations:

\[ \text{sort} \ Person . \]
\[ \text{subsort} \ Person < \text{Cid} . \]
\[ \text{op} \ \text{ctor} : \text{Person} \rightarrow \text{Person} \ [\text{ctor}] . \]
\[ \text{ops} \ \text{civstat} \text{ gender husband wife} : \rightarrow \text{AttributeName} \ [\text{ctor}] . \]

Role ends with multiplicity 1 are represented as attributes of sort \[ \text{Oid} \], and role ends with multiplicity \[ * \] as attributes of sort \[ \text{Set} \] (for \[ \text{Oid} \] sets).
An operation \( op(\text{arg}_1: \text{type}_1, \ldots, \text{arg}_n: \text{type}_n) : \text{type} \) is represented as a constant \( op \), of sort \( \text{OpName} \) (of operation names), and constants \( \text{arg}_1, \ldots, \text{arg}_n \), of sort \( \text{Arg} \) (of arguments). Thus, operations \( \text{marry(aSpouse:Person)} \) and \( \text{divorce()} \) require the following declarations:

\[
\begin{align*}
\text{ops} & \quad \text{marry} \divorce : \rightarrow \text{OpName} [\text{ctor}] . \\
\text{op} & \quad \text{aSpouse} : \rightarrow \text{Arg} [\text{ctor}] .
\end{align*}
\]

Once we have defined the structural part, we can create object configurations. Since we will be using these configurations in several commands in the following sections, we define constants \( \text{abc-single} \) and \( \text{ab-married-c-single} \) representing, respectively, the object configurations in which persons \( \text{ada} \), \( \text{bob} \), and \( \text{cyd} \) are single, and \( \text{ada} \) and \( \text{bob} \) married and \( \text{cyd} \) single, respectively.

\[
\begin{align*}
\text{ops} & \quad \text{ada bob cyd} : \rightarrow \text{oid} . \\
\text{ops} & \quad \text{abc-single ab-married-c-single} : \rightarrow \text{Configuration} . \\
\text{eq} & \quad \text{abc-single} \\
& \quad = < \text{ada} : \text{Person} \mid \text{gender} : \text{female}, \text{civstat} : \text{single}, \text{wife} : \text{Set} \{\}. \text{husband} : \text{Set} \{\} > \\
& \quad < \text{bob} : \text{Person} \mid \text{gender} : \text{male}, \text{civstat} : \text{single}, \text{wife} : \text{Set} \{\}. \text{husband} : \text{Set} \{\} > \\
& \quad < \text{cyd} : \text{Person} \mid \text{gender} : \text{male}, \text{civstat} : \text{single}, \text{wife} : \text{Set} \{\}. \text{husband} : \text{Set} \{\} > . \\
\text{eq} & \quad \text{ab-married-c-single} \\
& \quad = < \text{ada} : \text{Person} \mid \text{gender} : \text{female}, \text{civstat} : \text{married}, \text{wife} : \text{Set} \{\}. \text{husband} : \text{Set} \{\text{bob}\} > \\
& \quad < \text{bob} : \text{Person} \mid \text{gender} : \text{male}, \text{civstat} : \text{married}, \text{wife} : \text{Set} \{\text{ada}\}. \text{husband} : \text{Set} \{\} > \\
& \quad < \text{cyd} : \text{Person} \mid \text{gender} : \text{male}, \text{civstat} : \text{single}, \text{wife} : \text{Set} \{\}. \text{husband} : \text{Set} \{\} > .
\end{align*}
\]

### 3.3 OCL Constraints Representation

In mOdCL, OCL expressions are represented as terms of sort \( \text{OclExp} \). We can represent the OCL expressions from Section 2 as corresponding mOdCL terms of sort \( \text{OclExp} \). The mOdCL terms mirror quite closely the OCL expressions given in Section 2. We give here a few of them for illustration purposes:

\[
\begin{align*}
\text{op} & \quad \text{femaleHasNoWife} : \rightarrow \text{OclExp} . \\
\text{eq} & \quad \text{femaleHasNoWife} = (\text{context Person inv} \text{gender} = \text{female implies} \text{wife} \rightarrow \text{isEmpty}()) . \\
\text{op} & \quad \text{atMostOneHusband} : \rightarrow \text{OclExp} . \\
\text{eq} & \quad \text{atMostOneHusband} = (\text{context Person inv} \text{husband} \rightarrow \text{size}() \leq 1) . \\
\text{op} & \quad \text{wifeHusbandInverse} : \rightarrow \text{OclExp} . \\
\text{eq} & \quad \text{wifeHusbandInverse} \\
& \quad = \text{context Person inv} \\
& \quad \quad (\text{wife} \rightarrow \text{notEmpty}() \text{implies} \text{wife} . \text{husband} \rightarrow \text{includes}(\text{self})) \text{and} \\
& \quad \quad (\text{husband} \rightarrow \text{notEmpty}() \text{implies} \text{husband} . \text{wife} \rightarrow \text{includes}(\text{self})) . \\
\text{op} & \quad \text{aSpouseDefined} : \rightarrow \text{OclExp} . \\
\text{eq} & \quad \text{aSpouseDefined} = (\text{aSpouse} . \text{isDefined}()) . \\
\text{op} & \quad \text{femaleHasMarriedHusband} : \rightarrow \text{OclExp} . \\
\text{eq} & \quad \text{femaleHasMarriedHusband} \\
& \quad = (\text{gender} = \text{female implies} \\
& \quad \quad \text{husband} = \text{Set} \{\text{aSpouse}\} \text{ and} \text{civstat} \rightarrow \text{forAll}(\text{cs} | \text{cs} = \text{married})) . \\
\text{op} & \quad \text{husbandDivorced} : \rightarrow \text{OclExp} . \\
\text{eq} & \quad \text{husbandDivorced} \\
& \quad = \text{gender} = \text{female implies} \\
& \quad \quad (\text{husband} \rightarrow \text{isEmpty}() \text{and} \text{husband} @\text{pre} . \text{civstat} \rightarrow \text{forAll}(\text{cs} | \text{cs} = \text{divorced})) .
\end{align*}
\]

Notice the use of constants (like \( \text{femaleHasNoWife} \)) to identify these expressions. Invariants and pre- and postconditions are then specified by operators \( \text{inv}, \text{pre} \) and \( \text{post} \) in order to build the \( \text{OclExp} \) to be checked when the validation process requires to inspect invariants, pre- or postconditions of a given method, respectively.
The global invariant to be satisfied is given as an operator \( \text{inv} \) defined as the conjunction of all invariant expressions. Given the above definitions, the \( \text{inv} \) constant can be defined as:

\[
\text{eq } \text{inv} = \text{femaleHasNoWife} \land \text{maleHasNoHusband} \land \text{atMostOneHusband} \land \\
\text{atMostOneWife} \land \text{wifeHusbandInverse}.
\]

The \( \text{pre} \) and \( \text{post} \) operators must be defined for each method. These operators need the associated operation as an argument. We only show the corresponding declarations for the \( \text{marry} \) operation; it works analogously for \( \text{divorce} \).

\[
\text{eq } \text{pre(marry)} = (\text{aSpouseDefined} \land \text{isUnmarried} \land \text{aSpouseUnmarried} \land \text{differentGenders}) . \\
\text{eq } \text{post(marry)} = (\text{isMarried} \land \text{femaleHasMarriedHusband} \land \text{maleHasMarriedWife}) .
\]

With this structural description of the system, we can evaluate any OCL expression on object configurations by using the mOdCL’s \( \text{eval} \) operator. We can for instance evaluate the \( \text{inv} \) OCL expression defined above on the \( \text{abc-single} \) object configuration introduced in Section 3.

\[
\text{Maude}\text{> red eval(}\text{inv, abc-single}\text{).} \\
\text{result} \text{ Bool: true}
\]

### 4 System Dynamics

#### 4.1 Representation of Operation Calls

Once the static part of the system is completely determined, we can specify its behavior. We could do that in different ways, but our goal is to be able to generate execution traces leading to states satisfying or violating given OCL constraints. However, making sure that all invariants and pre- and postconditions are satisfied in the right places, is not an easy task. To discharge the user of the burden, mOdCL provides facilities to perform the appropriate checks in the appropriate places just by following very simple rules.

Basically, the preconditions of an operation must be checked before its execution starts, and when it is completed, its postconditions and the invariants must be satisfied. To be able to automatically perform the checks before and after the execution of each operation, mOdCL expects that methods are invoked with a call message of the form

\[
\text{call(}<\text{method-name}>, <\text{addressee}>, <\text{argument-list}>)
\]

and upon their completion they send a return message of the form

\[
\text{return(}<\text{return-value}>)
\]

The infrastructure of mOdCL will intercept these messages and will take the following measures.

The processing of a \( \text{call} \) operator results in the execution of a method, for which a context object, representing the execution context with the appropriate information for the running method, is generated. This object, of class \( \text{Context} \), is of the form

\[
<\text{Ctx} : \text{Context} | \text{op} : \text{M}, \text{self} : \text{Id}, \text{args} : \text{Vars} >
\]

where \( \text{Ctx} \) is the identifier of the object, \( \text{M} \) is the name of the active method, \( \text{Id} \) is the identifier of the current object, and \( \text{Vars} \) is a set of (variable name,value) pairs corresponding to the arguments of the
method invocation and local variables. The modeler can make use of this Context object to get or manipulate the value of an argument variable or a local variable.

A return message return(<return-value>) will be replaced by a resume message of the form

\[
\text{resume(<return-value>)}
\]

To manage the chaining of method invocations — an invoked method can invoke another one (or recursively itself) — the validator uses an execution stack in which the necessary information is stored.\(^3\) mOdCL configurations are terms of sort Configuration+, which is declared as a sort with a single constructor

\[
\text{op } \{\_\} : \text{Configuration} \rightarrow \text{Configuration}^+.
\]

where one of the objects in the wrapped configuration represents the execution stack, which is represented by a stack operator.

### 4.2 Operations marry and divorce

With these restrictions, we can specify the behavior of our operations as follows. Since the husband and wife attributes are sets of object identifiers, the marry operation just adds a new husband or a new wife to the corresponding set, depending on the gender of the receiver of the message.

\[
\begin{align*}
\text{vars} & \quad \text{Ctx Self Dude} \quad : \quad \text{Oid}. & \quad \text{vars} & \quad \text{St1 St2} \quad : \quad \text{CivilStatus}. \\
\text{vars} & \quad \text{Gd1 Gd2} \quad : \quad \text{Gender}. & \quad \text{vars} & \quad \text{Hb1 Hb2 Wf1 Wf2} \quad : \quad \text{Set}.
\end{align*}
\]

\[
\text{r1} [\text{MARRY}] : \\
\begin{align*}
\& \quad \text{Ctx} : \text{Context} \mid \text{op} : \text{marry}, \text{self} : \text{Self}, \text{args} : \text{arg(aSpouse, Dude)} > \\
\& \quad \text{marry-token} \\
\& \quad \text{Self} : \text{Person} \mid \text{civstat} : \text{St1}, \text{gender} : \text{Gd1}, \text{husband} : \text{Hb1}, \text{wife} : \text{Wf1} > \\
\& \quad \text{Dude} : \text{Person} \mid \text{civstat} : \text{St2}, \text{gender} : \text{Gd2}, \text{husband} : \text{Hb2}, \text{wife} : \text{Wf2} > \\
\& \quad \text{if} \ Gd1 = \text{female} \\
\& \quad \text{then} \ Gd1 = \text{false} \\
\& \quad \text{then} \ < \text{Self} : \text{Person} \mid \text{civstat} : \text{married}, \text{gender} : \text{Gd1}, \text{husband} : \text{Set}\{\text{Dude}\}, \text{wife} : \text{Wf1} > \\
\& \quad \text{Dude} : \text{Person} \mid \text{civstat} : \text{married}, \text{gender} : \text{Gd2}, \text{husband} : \text{Hb2}, \text{wife} : \text{Set}\{\text{Self}\} > \\
\& \quad \text{else} \ < \text{Self} : \text{Person} \mid \text{civstat} : \text{married}, \text{gender} : \text{Gd1}, \text{husband} : \text{Hb1}, \text{wife} : \text{Set}\{\text{Dude}\} > \\
\& \quad \text{Dude} : \text{Person} \mid \text{civstat} : \text{married}, \text{gender} : \text{Gd2}, \text{husband} : \text{Set}\{\text{Self}\}, \text{wife} : \text{Wf2} > \\
\& \quad \text{fi} \\
\& \quad \text{Ctx} : \text{Context} \mid \text{op} : \text{marry}, \text{self} : \text{Self}, \text{args} : \text{arg(aSpouse, Dude)} > \\
\& \quad \text{return}(0).
\end{align*}
\]

In the case of the divorce operation, the set of the corresponding husband or wife attribute is just left empty. We use two rules to distinguish cases, depending on the gender, but show only the rule for females.

\[
\begin{align*}
\text{vars} & \quad \text{Ctx Self Dude} \quad : \quad \text{Oid}. & \quad \text{vars} & \quad \text{St1 St2} \quad : \quad \text{CivilStatus}. & \quad \text{var} & \quad \text{Wf} \quad : \quad \text{Set}. \\
\text{vars} & \quad \text{Gd1 Gd2} \quad : \quad \text{Gender}. & \quad \text{vars} & \quad \text{Atts1 Atts2} \quad : \quad \text{AttributeSet}. \\
\text{r1} [\text{DIVORCE}] : \\
\begin{align*}
\& \quad \text{Ctx} : \text{Context} \mid \text{op} : \text{divorce}, \text{self} : \text{Self}, \text{args} : \text{empty} > \\
\& \quad \text{divorce-token} \\
\& \quad \text{Self} : \text{Person} \mid \text{civstat} : \text{St1}, \text{gender} : \text{female}, \text{husband} : \text{Set}\{\text{Dude}\}, \text{Atts1} > \\
\& \quad \text{Dude} : \text{Person} \mid \text{civstat} : \text{St2}, \text{wife} : \text{Wf}, \text{Atts2} > \\
\& \quad \text{if} \ \text{Gd1} = \text{false} \\
\& \quad \text{then} \ < \text{Self} : \text{Person} \mid \text{civstat} : \text{divorced}, \text{gender} : \text{female}, \text{husband} : \text{Set}\{\}, \text{Atts1} > \\
\& \quad \text{Dude} : \text{Person} \mid \text{civstat} : \text{divorced}, \text{wife} : \text{Set}\{\}, \text{Atts2} > \\
\& \quad \text{then} \ < \text{Ctx} : \text{Context} \mid \text{op} : \text{divorce}, \text{self} : \text{Self}, \text{args} : \text{empty} > \\
\& \quad \text{return}(0).
\end{align*}
\]

\(^3\)mOdCL currently assumes a single thread of execution. A multithreaded execution would also be possible just by adding support for multiple stacks.
These methods are very simple, and they are both specified so that their execution will consists in the application of one rule. The marry-token and divorce-token operators are used to prevent the execution of the rules corresponding to the operation out of the scope of a call to such an operation. More intricate methods may require several rules to be executed sequentially, in which case more than a simple token will be required.

Methods are invoked with call operators. However, for convenience, and to use a notation more friendly, we will use the following operators.

op _ . marry(_): Oid Oid -> Msg [msg] .
op _ . divorce(): Oid -> Msg [msg] .

These methods will be used to produce the sending of a corresponding call message.

rl [MARRY-INVOCATION] :
Self . marry(Arg)
stack(nil)
=> call(marry, Self, arg(aSpouse, Arg))
stack(nil)
marry-token .

rl [DIVORCE-INVOCATION] :
Self . divorce()
stack(nil)
=> call(divorce, Self, empty)
stack(nil)
divorce-token .

Notice the presence of the stack operator, with nil contents in both rules, to guarantee that no new message is sent until the execution stack is empty, that is, the processing of all previous messages has been completed.

Given this behavior we can simulate the system by using the rewrite Maude command[^1] or we can explore all possible executions searching for states satisfying a specific property.

Maude> rew { abc-single
        cyd . marry(ada)
        stack(nil) } .
result Configuration+:
{ stack(nil)
  resume(marry, 0)
  < ada : Person | civstat : married, gender : female, husband : Set{cyd}, wife : Set{} >
  < bob : Person | civstat : single, gender : male, husband : Set{}, wife : Set{} >
  < cyd : Person | civstat : married, gender : male, husband : Set{}, wife : Set{ada} > }

What happens if we try to do something that does not respect the defined constraints? For instance, if we send a marry message to an already married person, the precondition of the marry operation should fail. mOdCL will detect situations like this and will send corresponding error messages.

Maude> rew { ab-married-c-single
        cyd . marry(ada)
        stack(nil) } .
result [Configuration+]:
{ marry-token
  < ada : Person | civstat : married, gender : female, husband : Set{bob}, wife : Set{} >
  < bob : Person | civstat : married, gender : male, husband : Set{}, wife : Set{ada} >
  < cyd : Person | civstat : single, gender : male, husband : Set{}, wife : Set{} >
  error("Precondition violation") .

[^1]: The rewrite Maude command (abbreviated rew) causes the specified term to be rewritten using the rules, equations, and membership axioms in the given module.
The error message provides a string explaining the failure, and the OCL expression that has failed. In the case of pre- and postconditions, the name of the operation of the condition is also indicated.

5 Tracing Properties

Notice that the configuration rewritten is a set, and so, if several messages were in it, no order in which they are to be consumed is assumed. If we were interested in a sequence of messages with a specific order, we could very easily specify it. But this associative-commutative configuration allows us to explore the different traces in which certain messages may be consumed. For instance, the following search command looks up all possible traces leading to states in which there are two persons married using messages cyd . marry(ada), ada . marry(bob), and bob . divorce().

Maude> search { abc-single
    (cyd . marry(ada))
    (ada . marry(bob))
    (bob . divorce())
    stack(nil)}
=>* { stack(nil)
    Conf:Configuration } such that
        eval(Person . allInstances -> exists(P |
            Person . allInstances -> exists(Q |
                P . wife -> includes(Q) and Q . husband -> includes(P)))).
Conf:Configuration .
Solution 1 (state 4)
Conf:Configuration
    --> bob . divorce()
    resume(marry, 0)
    (cyd . marry(ada))
    < ada : Person | civstat : married, gender : female, husband : Set{bob}, wife : Set{} >
    < bob : Person | civstat : married, gender : male, husband : Set{}, wife : Set{ada} >
    < cyd : Person | civstat : single, gender : male, husband : Set{}, wife : Set{} >
Solution 2 (state 5)
Conf:Configuration
    --> bob . divorce()
    resume(marry, 0)
    (ada . marry(bob))
    < ada : Person | civstat : married, gender : female, husband : Set{cyd}, wife : Set{} >
    < bob : Person | civstat : single, gender : male, husband : Set{}, wife : Set{} >
    < cyd : Person | civstat : married, gender : male, husband : Set{}, wife : Set{ada} >
Solution 3 (state 12)
Conf:Configuration
    --> resume(marry, 0)
    resume(marry, 0)
    resume(divorce, 0)
    < ada : Person | civstat : married, gender : female, husband : Set{cyd}, wife : Set{} >
    < bob : Person | civstat : divorced, gender : male, husband : Set{}, wife : Set{} >
    < cyd : Person | civstat : married, gender : male, husband : Set{}, wife : Set{ada} >
No more solutions.

The Maude search command gives us all possible reachable states satisfying the specified OCL expression. More interestingly, we have access to the paths followed for each of these solutions. By using the show path and show path labels commands, we can get all the information on the path followed.
up to any of the states in the search space, and specifically to the ones given as solutions of the search command — notice the state number given on the right of the solution number.

Maude> show path labels 12.
MARRY−INVOCATION
MARRY DIVORCE−INVOCATION
DIVORCE MARRY−INVOCATION
MARRY

5.1 Spontaneous Generation of Messages

This is very powerful already, but we can do more. In the above search we had to give what messages we wanted to use to generate all possible paths. But that does not guarantee that we get all possible ways to get a specific state. We now present a possible alternative in which new messages are spontaneously generated. As a result of the execution we are interested in getting the sequence of messages leading to the searched states. To obtain it, we add a messages operator that keeps the list of messages generated.

Consider the following rules, by which, if there exist two persons, a new marry message for the marriage of these two persons is generated:

\[
\text{rl [NEW−MARRY−MESSAGE]} : \\
\text{< Dude1 : Person | Atts1 >} \\
\text{< Dude2 : Person | Atts2 >} \\
\text{messages(ML)} \\
\Rightarrow \text{< Dude1 : Person | Atts1 >} \\
\text{< Dude2 : Person | Atts2 >} \\
\text{hold(messages(ML ; Dude1 . marry(Dude2)))} \\
\text{Dude1 . marry(Dude2)}.
\]

In addition to sending a new marry message, the rule adds the generated message to the list of messages. And, since we do not want to generate a new message until the previous one was consumed, we block the messages operator with a hold operator.

\[
\text{op hold : Msg \rightarrow Msg [ctor]}.
\]

Once the operation has finished, we can unblock the message list.

\[
\text{var ML : MsgList .} \\
\text{var Arg : OclType .}
\]

\[
\text{rl [RETURN−MARRY]} : \\
\text{resume(marry, Arg)} \\
\text{hold(messages(ML))} \\
\Rightarrow \text{messages(ML)}.
\]

Similar rules are added to spontaneously generate new divorce messages.
Notice the use of the messages operator to collect the sequence of messages generated. When a new message is generated it is added to the list of messages in the messages operator. With it we will know the messages used to reach a given state. We could instead use the show path command to get the sequence of rules, and from it obtain the sequence of messages, but notice that the search command looks for states, and only keeps the shortest path to each reached state.

We can get, e.g., the states reached when searching for married couples. Of course, the number of reachable states satisfying this condition is infinite, and we must either limit the number of solutions to look for or the maximum depth of the search. We can search for the first 10,000 states as follows:

Maude> search [10000]
{ stack(nil)
  abc=single
  messages(nilMsgList) }
=>* { Conf:Configuration messages(ML:MsgList) }
such that
eval(Person . allInstances -> exists(P | Person . allInstances -> exists(Q | P . wife -> includes(Q) and Q . husband -> includes(P))),
Conf:Configuration).

Solution 1 (state 17)
Conf:Configuration
  --> stack(nil)
  < ada : Person | civstat : married, gender : female, husband : Set{bob}, wife : Set{cyd} >
  < bob : Person | civstat : married, gender : male, husband : Set{cyd}, wife : Set{ada} >
  < cyd : Person | civstat : single, gender : male, husband : Set{cyd}, wife : Set{ada} >
ML:MsgList --> ada . marry(bob)

...

Solution 10000 (state 131696)
Conf:Configuration
  --> stack(nil)
  < ada : Person | civstat : married, gender : female, husband : Set{cyd}, wife : Set{cyd} >
  < bob : Person | civstat : divorced, gender : male, husband : Set{cyd}, wife : Set{cyd} >
  < cyd : Person | civstat : married, gender : male, husband : Set{cyd}, wife : Set{cyd} >
ML:MsgList --> ada . marry(cyd):
  cyd . divorce();
  bob . marry(ada);
  bob . divorce();
  cyd . marry(ada);
  cyd . divorce();
  ada . marry(cyd);
  ada . divorce();
  cyd . marry(ada)

We may also look for more interesting states. For example, we may search for states in which the precondition of an operation fails. We must again limit the search.

Maude> search [20,20]
{ stack(nil)
  abc=single
  messages(nilMsgList) }
=>* { error(Str:String, ON:OpName, Exp:OCL-Exp)
Conf:[Configuration]
  hold(messages(ML:MsgList)) }.

Solution 1 (state 10)
Conf:[Configuration]
  --> marry-token
  < ada : Person | civstat : single, gender : female, husband : Set{mt}, wife : Set{mt} >
  < bob : Person | civstat : single, gender : male, husband : Set{mt}, wife : Set{mt} >
  < cyd : Person | civstat : single, gender : male, husband : Set{mt}, wife : Set{mt} >
ML:MsgList --> bob . marry(cyd)
However, we can show how we cannot reach states in which invariants are not satisfied.

Maude> search [20,20]
{ stack(nil)
  abc-single
  messages(nilMsgList) }
=>* { error(Str:String, Exp:OCL-Exp)
    Conf:[Configuration]
    hold(messages(ML:MsgList)) } .
No solution.

Of course, since we have limited the search, the absence of solutions does not imply that such states do not exist. But notice that the reachable state space is infinite because of the messages operator. Without it, the state space is in fact small for our example. We can verify properties on our example by using an equational abstraction consisting in just removing the messages in the messages list.

```
var ML : MsgList .
ceq messages(ML) = messages(nilMsgList) if ML != nilMsgList .
```

We can now, for example, prove that within our Maude description no reachable state fails the invariant.

Maude> search { stack(nil)
  abc-single
  messages(nilMsgList) }
=>* { error(Str:String, Exp:OCL-Exp)
    Conf:[Configuration]
    hold(messages(ML:MsgList)) } .
No solution.

Or more interestingly, that after you marry you will never be single again.

Maude> search { stack(nil)
  ab-married-c-single
  messages(nilMsgList) }
=>* { Conf:[Configuration]
    messages(ML) } such that
eval(ada . civstat = single or bob . civstat = single, Conf) .
No solution.
5.2 Spontaneous Generation of Objects

But why limiting ourselves to playing with ada, bob, and cyd? In the same way we can spontaneously generate new messages to consider all possible sequences of messages, we can also think on the possibility of spontaneously generating new objects. In fact, what we need is just a new or create message that gives place to the object. We declare a new message with the identifier of the object, its class, and its gender to generate Person objects.

```
| op | new : Oid Cid Gender -> Msg [ctor] . |
```

As in the generation of messages, to generate a new object we need to initialize its attributes. In our current example, they are generated single, and the only values we need to worry about are their identifier and their gender. Genders can be assigned nondeterministically. We give two rules to generate objects of each of the possible genders. Regarding identifiers, we use a counter operator to both indexing object identifiers id(1), id(2), ... and limiting the number of generated objects.

```
| op | id : Nat -> Oid [ctor] . |
| op | counter : Nat -> Msg [ctor] . |
| var | ML : MsgList . |
| var | N : Nat . |
| var | G : Gender . |
| var | O : Oid . |
| ops | P Q : -> Vid . |

```

```
| rl | [NEW-MALE] : |
| counter(s N) messages(ML) => | counter(N) |
| new(id(N), Person, male) hold(messages(ML : new(id(N), Person, male))). |

| rl | [NEW-FEMALE] : |
| counter(s N) messages(ML) => | counter(N) |
| new(id(N), Person, female) hold(messages(ML : new(id(N), Person, female))). |

| rl | [PERSON-GENERATION] : |
| new(O, Person, G) hold(messages(ML)) => | < O : Person | civstat : single, gender : G, husband : Set{}, wife : Set{} > |
| messages(ML) . |
```

We can now look for traces reaching states with married couples as follows:

```
Maude> search [100,30] { stack(nil) messages(nilMsgList) counter(3) } =>* { Conf:Configuration messages(ML:MsgList) } such that eval(Person . allInstances -> exists(P | Person . allInstances -> exists(Q | P . wife -> includes(Q) and Q . husband -> includes(P))), Conf) .
Solution 1 (state 121)
Conf:Configuration -> stack(nil) counter(1) < id(1) : Person | civstat : married, gender : female, husband : Set{id(2)}, wife : Set{} > < id(2) : Person | civstat : married, gender : male, husband : Set{}, wife : Set{id(1)}> ML:MsgList --> new(id(2), Person, male) ;
    new(id(1), Person, female) ;
```
We can look for persons married to themselves.

\texttt{Maude}\> search [20,20] \{ stack(nil) \> messages(nilMsgList) \> counter(3) \} =\>
\{ Conf:Configuration \> messages(ML:MsgList) \} such that eval(Person . allInstances -> exists(P | P . wife -> includes(P) or P . husband -> includes(P)), Conf:Configuration) .

No solution.

6 Conclusions

We have made a proposal to draw consequences about stated constraints in a UML and OCL model by translating the model into Maude. Maude allows the developer to check for system properties in various ways. We have employed the Maude state search and were able to show that particular states can (resp. cannot) be reached under given assumptions, i.e., by limiting the search to a finite subset of possible configurations. We have generated scenarios, i.e., state sequences and transitions, where the transition sequences were automatically constructed in order to achieve a path from a start state to an end state.

We believe Maude with its sophisticated tools offers more and even stronger possibilities than the one we have employed. The Maude ITP theorem prover could be used to achieve general propositions about invariants. The Maude temporal logic checker could be employed for verifying and finding complex states and transition constellations. Fine grained information about constraint failure would be desirable for mOdCL. Larger case studies must give feedback about the practicability of our approach. All results obtained on the Maude level must be translated back into UML and OCL, for example, in terms of object and sequence diagrams.
References


