Tracing Properties of UML and OCL Models with Maude

Francisco Durán[1], Martin Gogolla[2], and Manuel Roldán[1]

[1] ETSI Informática. Universidad de Málaga, Spain
[2] University of Bremen, Germany

Abstract. 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 satisfiability 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 familar 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`.

```
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.

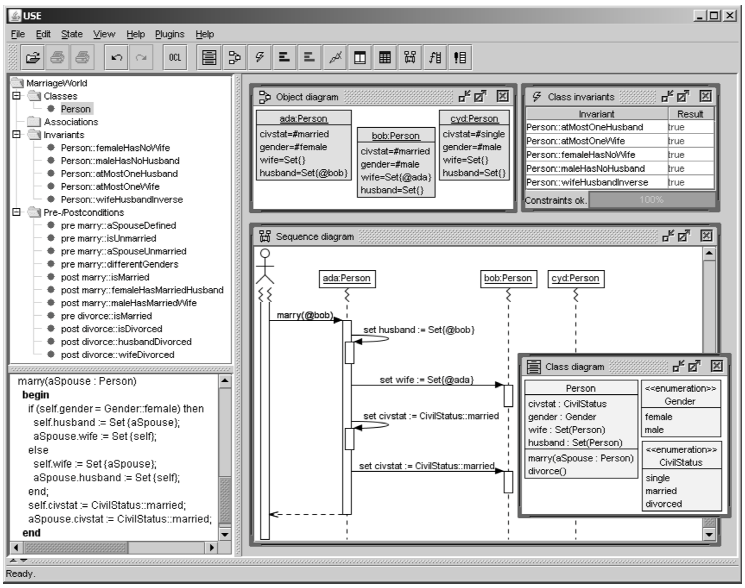

Fig. 1. Snapshot of the USE tool for one of the example scenarios.

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 (`femaleHasNoWife` and `maleHasNoHusband`), a person can have at most one spouse (`atMostOneHusband` and `atMostOneWife`), and marriage is a symmetric association (`wifeHusbandInverse`). We can express these invariants for the `Person` class in OCL:

```
inv femaleHasNoWife: gender=female implies wife->isEmpty()
inv maleHasNoHusband: gender=male implies husband->isEmpty()
inv atMostOneHusband: husband->size()<=1
inv atMostOneWife: wife->size()<=1
inv wifeHusbandInverse:
      (wife->notEmpty() implies wife.husband->includes(self)) and
      (husband->notEmpty() implies husband.wife->includes(self))
```

One could additionally state the invariant `(civstat=married)=(wife->notEmpty() or husband->notEmpty())` in order to guarantee consistency of the attribute values.

Class `Person` has a `marry(aSpouse:Person)` operation. If (a) the argument `aSpouse` of the operation is defined,[3] (b) both the person receiving the message and the argument `aSpouse` 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:[4]

```
pre  aSpouseDefined: aSpouse.isDefined
pre  isUnmarried: civstat<>married
pre  aSpouseUnmarried: aSpouse.civstat<>married
pre  differentGenders: gender<>aSpouse.gender
post isMarried: civstat=married
post femaleHasMarriedHusband: gender=female implies
     husband=Set{aSpouse} and husband.civstat->forAll(cs|cs=married)
post maleHasMarriedWife: gender=male implies
     wife=Set{aSpouse} and wife.civstat->forAll(cs|cs=married)
```

Two married persons can get divorced using the `divorce()` operation. This operation can be specified in OCL as follows:

```
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 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] In OCL, there is an exception element representing undefined; it is possible to pass this undefined element as an argument to a `marry` call.

[4] For the `femaleHasMarriedHusband` constraint, given the first part of the consequent `husband=Set{aSpouse}`, one can simplify its second part to `aSpouse.civstat=married` assuming that the formal operation parameter `aSpouse` cannot be changed by the operation. And similarly for the `maleHasMarriedWife` constraint. However, we prefer in our example the explicit formulation.

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 (Σ, E), where Σ 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 $t \to t'$, where t and t' are Σ-terms. This rewriting happens modulo the equations E, describing in fact local transitions $[t]_E \to [t']_E$. 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: `Object`, `Msg` (messages), and `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 `Object` of the form $< O : C \mid a_1 : v_1$, ..., $a_n : v_n >$, 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 `Msg`.

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

```
sort Account .
subsort Account < Cid .
op Account : -> Account [ctor] .
op balance :_ : Int -> Attribute [ctor] .

op withdraw : Oid Int -> Msg [ctor] .
op transfer : Oid Oid Int -> Msg [ctor] .
```

Thus, given the predefined operator

```
op <_:_|_> : Oid Cid AttributeSet -> Object [ctor] .
```

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

```
< a : Account | balance : 5 >
```

and a message to `a` to withdraw 3 as

```
withdraw(a, 3)
```

In a concurrent object-oriented system, the concurrent state, which is called a *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 `Configuration` represents configurations of Maude objects and messages, with `none` as empty configuration and the empty syntax operator `__` as union of configurations.

```
sort Configuration .
subsorts Object Message < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
    [ctor assoc comm id: none] .
```

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

$$
\begin{array}{l}
\texttt{crl [}r\texttt{] :} \\
\quad \texttt{<}\ O_1\ \texttt{:}\ C_1\ \texttt{|}\ atts_1\ \texttt{>}\ \ldots\ \texttt{<}\ O_n\ \texttt{:}\ C_n\ \texttt{|}\ atts_n\ \texttt{>} \\
\quad M_1\ \ldots\ M_m \\
\quad \texttt{=>}\ \texttt{<}\ O_{i_1}\ \texttt{:}\ C'_{i_1}\ \texttt{|}\ atts'_{i_1}\ \texttt{>}\ \ldots\ \texttt{<}\ O_{i_k}\ \texttt{:}\ C'_{i_k}\ \texttt{|}\ atts'_{i_k}\ \texttt{>} \\
\qquad \texttt{<}\ Q_1\ \texttt{:}\ C''_1\ \texttt{|}\ atts''_1\ \texttt{>}\ \ldots\ \texttt{<}\ Q_p\ \texttt{:}\ C''_p\ \texttt{|}\ atts''_p\ \texttt{>} \\
\qquad M'_1\ \ldots\ M'_q \\
\quad \texttt{if}\ Cond\ \texttt{.}
\end{array}
$$

where r is the rule label, $M_1 ... M_m$ and $M'_1 ... M'_q$ are messages, $O_1 ... O_n$ and $Q_1 ... Q_p$ are object identifiers, $C_1 ... C_n$, $C'_{i_1} ... C'_{i_k}$ and $C''_1 ... C''_p$ are classes, $i_1 ... i_k$ is a subset of $1 ... n$, and $Cond$ is a Boolean condition (the rule's *guard*). The result of applying such a rule is that: (a) messages $M_1 ... M_m$ disappear, i.e., they are consumed; (b) the state, and possibly the classes of objects $O_{i_1} ... O_{i_k}$ may change; (c) all the other objects O_j vanish; (d) new objects $Q_1 ... Q_p$ are created; and (e) new messages $M'_1 ... M'_q$ are created, i.e., they are sent. Rule labels and guards are optional. When several objects or messages appear in the left-hand side of a rule, they need to synchronize in order for such a rule to be fired.

Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration `C < C'`, indicating that `C` is a subclass of `C'`, is a particular case of a subsort declaration `C < C'`, by which all attributes, messages, and rules of the superclasses, as well as the newly defined attributes, messages and rules of the subclass characterize its structure and behavior. Multiple inheritance is also supported in Maude [4].

3.2 OCL Basic Types

Basic predefined OCL types are directly mapped on predefined Maude sorts. Thus, the OCL types `Boolean`, `Integer`, `Real` and `String` are mapped into the Maude predefined sorts `Bool`, `Int`, `Float`, and `String`, respectively. User-defined classes are mapped on the Maude predefined sort `Oid` (of object identifiers). OCL predefined collection types are mapped into sorts `Set`, `Bag`, `OrderedSet`, and `Sequence`. All these sorts are subsorts of `OclType`.

In OCL, the `Enum` sort is intended to be used to define enumeration types. Thus, types `Gender` and `CivilStatus` are declared as enumerations by declaring them as subsorts of `Enum`. The values of the types are then declared as constants of the respective types. E.g., the declarations for `CivilStatus` are:

```
sort CivilStatus .
subsort CivilStatus < Enum .
ops single married divorced : -> CivilStatus [ctor] .
```

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, of sort `AttributeName`, and its type, of sort `OclType`.

```
op _:_ : AttributeName OclType -> Attribute [ctor] .
```

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

```
sort Person .
subsort Person < Cid .
op Person : -> Person [ctor] .
ops civstat gender husband wife : -> AttributeName [ctor] .
```

Role ends with multiplicity 1 are represented as attributes of sort `Oid`, and role ends with multiplicity $\star$ as attributes of sort `Set` (for `Oid` sets).

An operation $op(arg_1 : type_1, \ldots, arg_n : type_n) : type$ is represented as a constant op, of sort `OpName` (of operation names), and constants $arg_1, \ldots, arg_n$, of sort `Arg` (of arguments). Thus, operations `marry(aSpouse:Person)` and `divorce()` require the following declarations:

```
ops marry divorce : -> OpName [ctor] .
op aSpouse : -> Arg [ctor] .
```

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 `abc-single` and `ab-married-c-single` representing, respectively, the object configurations in which persons `ada`, `bob`, and `cyd` are single, and `ada` and `bob` married and `cyd` single, respectively.

```
ops ada bob cyd : -> Oid .
ops abc-single ab-married-c-single : -> Configuration .
eq abc-single
  = < ada : Person |
```

```
        gender : female, civstat : single, wife : Set{}, husband : Set{} >
    < bob : Person |
        gender : male,   civstat : single, wife : Set{}, husband : Set{} >
    < cyd : Person |
        gender : male,   civstat : single, wife : Set{}, husband : Set{} > .
eq ab-married-c-single
  = < ada : Person |
        gender : female, civstat : married, wife : Set{},    husband : Set{bob} >
    < bob : Person |
        gender : male,   civstat : married, wife : Set{ada}, husband : Set{} >
    < cyd : Person |
        gender : male,   civstat : single,  wife : Set{},    husband : Set{} > .
```

3.3 OCL Constraints Representation

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

```
op femaleHasNoWife : -> OCL-Exp .
eq femaleHasNoWife
  = (context Person inv gender = female implies wife -> isEmpty()) .
```

```
op atMostOneHusband : -> OCL-Exp .
eq atMostOneHusband = (context Person inv husband -> size() <= 1) .
```

```
op wifeHusbandInverse : -> OCL-Exp .
eq wifeHusbandInverse = context Person inv
     (wife -> notEmpty() implies wife . husband -> includes(self)) and
     (husband -> notEmpty() implies  husband . wife -> includes(self)) .
```

```
op aSpouseDefined : -> OCL-Exp .
eq aSpouseDefined = (aSpouse . isDefined()) .
```

```
op femaleHasMarriedHusband : -> OCL-Exp .
eq femaleHasMarriedHusband = (gender = female implies
     husband = Set{aSpouse} and
     husband . civstat -> forAll(cs | cs = married)) .
```

```
op husbandDivorced : -> OCL-Exp .
eq husbandDivorced = gender = female implies
     (husband -> isEmpty() and
      husband @pre . civstat -> forAll(cs | cs = divorced)) .
```

Notice the use of constants (like `femaleHasNoWife`) to identify these expressions. Invariants and pre- and postconditions are then specified by operators `inv`, `pre` and `post` in order to build the `OclExp` to be checked when the validation process requires to inspect invariants, pre- or postconditions of a given method, respectively.

```
op inv : -> OclExp .
ops pre post : OpName -> OclExp .
```

The global invariant to be satisfied is given as an operator `inv` defined as the conjunction of all invariant expressions. Given the above definitions, the `inv` constant can be defined as:

```
eq inv = femaleHasNoWife and maleHasNoHusband and atMostOneHusband and
         atMostOneWife and wifeHusbandInverse .
```

The `pre` and `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 `marry` operation; it works analogously for `divorce`.

```
eq pre(marry)
  = (aSpouseDefined and isUnmarried and aSpouseUnmarried and differentGenders) .
eq post(marry)
  = (isMarried and femaleHasMarriedHusband and maleHasMarriedWife) .
```

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

```
Maude> red eval(inv, abc-single) .
result 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

```
call(<method-name>, <addressee>, <argument-list>)
```

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

```
return(<return-value>)
```

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

The processing of a `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 `Context`, is of the form

```
< Ctx : Context | op : M, self : Id, args : Vars >
```

where `Ctx` is the identifier of the object, `M` is the name of the active method, `Id` is the identifier of the current object, and `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

```
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.[5] mOdCL configurations are terms of sort `Configuration+`, which is declared as a sort with a single constructor

```
op {_} : Configuration -> 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.

```
vars Ctx Self Dude : Oid .    vars St1 St2 : CivilStatus .
vars Gd1 Gd2 : Gender .       vars Hbs1 Hbs2 Wfs1 Wfs2 : Set .
```

```
rl [MARRY] :
  < Ctx : Context | op : marry, self : Self, args : arg(aSpouse, Dude) >
  marry-token
  < Self : Person | civstat : St1,   gender : Gd1, husband : Hbs1, wife : Wfs1 >
  < Dude : Person | civstat : St2,   gender : Gd2, husband : Hbs2, wife : Wfs2 >
  => if Gd1 == female
     then < Self : Person |
             civstat : married, gender : Gd1, husband : Set{Dude}, wife : Wfs1 >
          < Dude : Person |
             civstat : married, gender : Gd2, husband : Hbs2, wife : Set{Self} >
     else < Self : Person |
             civstat : married, gender : Gd1, husband : Hbs1, wife : Set{Dude} >
          < Dude : Person |
             civstat : married, gender : Gd2, husband : Set{Self}, wife : Wfs2 >
     fi
     < Ctx : Context | op : marry, self : Self, args : arg(aSpouse, Dude) >
     return(0) .
```

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.

[5] mOdCL currently assumes a single thread of execution. A multithreaded execution would also be possible just by adding support for multiple stacks.

```
vars Ctx Self Dude : Oid .          vars St1 St2 : CivilStatus .
vars Gd1 Gd2 : Gender .             vars Husbands Wifes : Set .
vars Atts1 Atts2 : AttributeSet .
```

```
rl [DIVORCE] :
  < Ctx : Context | op : divorce, self : Self, args : empty >
  divorce-token
  < Self : Person | civstat : St1, gender : female, husband : Set{Dude}, Atts1 >
  < Dude : Person | civstat : St2, wife : Wifes, Atts2 >
  => < Self : Person |
         civstat : divorced, gender : female, husband : Set{}, Atts1 >
     < Dude : Person | civstat : divorced, wife : Set{}, Atts2 >
     < Ctx : Context | op : divorce, self : Self, args : empty >
     return(0) .
```

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,[6] 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 |
```

[6] The `rewrite` Maude command (abbreviated `rew`) causes the specified term to be rewritten using the rules, equations, and membership axioms in the given module.

```
      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",
          marry,
          aSpouse . isDefined() and civstat <> married and
          gender <> aSpouse . gender and aSpouse . civstat <> married) }
```

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.

```
sort MsgList .
subsort Msg < MsgList .
op nilMsgList : -> MsgList .
op _;_ : MsgList MsgList -> MsgList [assoc id: nilMsgList] .
```

```
op messages : MsgList -> Msg [ctor] .
```

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

```
var  ML : MsgList .                vars Dude1 Dude2 : Oid .
vars Atts1 Atts2 : AttributeSet .  var  Arg : OclType .

rl [NEW-MARRY-MESSAGE] :
  < Dude1 : Person | Atts1 >
  < Dude2 : Person | Atts2 >
  messages(ML)
  => < Dude1 : Person | Atts1 >
     < Dude2 : Person | Atts2 >
     hold(messages(ML ; Dude1 . marry(Dude2)))
     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.

```
op hold : Msg -> Msg [ctor] .
```

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

```
var  ML : MsgList .      var  Arg : OclType .

rl [RETURN-MARRY] :
  resume(marry, Arg)
  hold(messages(ML))
  => 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{} >
      < bob : Person |
          civstat : married, gender : male, husband : Set{}, wife : Set{ada} >
      < cyd : Person |
          civstat : single, gender : male, husband : Set{}, wife : Set{} >
ML:MsgList ---> ada . marry(bob)

...

Solution 10000 (state 131696)
Conf:Configuration
  ---> stack(nil)
      < 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} >
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)
Str:String ---> "Precondition_violation"
ON:OpName ---> marry
Exp:OCL-Exp
  ---> aSpouse . isDefined() and civstat <> married and
       gender <> aSpouse . gender and aSpouse . civstat <> married
...
Solution 20 (state 74)
Conf:[Configuration]
  ---> marry-token
      < ada : Person |
          civstat : married, gender : female, husband : Set{bob}, wife : Set{mt} >
      < bob : Person |
          civstat : married, gender : male, husband : Set{mt}, wife : Set{ada} >
      < cyd : Person |
          civstat : single, gender : male, husband : Set{mt}, wife : Set{mt} >
ML:MsgList ---> (bob . marry(ada)) ; cyd . marry(bob)
Str:String ---> "Precondition_violation"
ON:OpName ---> marry
Exp:OCL-Exp ---> aSpouse . isDefined() and civstat <> married and
```

```
                gender <> aSpouse . gender and aSpouse . civstat <> married
```

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 [9] 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) ;
               id(1) . marry(id(2))
Solution 2 (state 122)
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) ;
               id(2) . marry(id(1))
```

```
...
Solution 100 (state 2420)
Conf:Configuration
  --> stack(nil)
      counter(0)
      < id(1) : Person |
         civstat : married, gender : female, husband : Set{id(0)}, wife : Set{} >
      < id(2) : Person |
         civstat : single, gender : male, husband : Set{}, wife : Set{} >
      < id(0) : Person |
         civstat : married, gender : male, husband : Set{mt}, wife : Set{id(1)} >
ML:MsgList --> new(id(2), Person, male) ;
               new(id(1), Person, female) ;
               new(id(0), Person, male) ;
               id(1) . marry(id(0)) ;
               id(0) . divorce() ;
               id(0) . marry(id(1))
```

We can look for persons married to themselves.

```
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

1. A. Boronat and J. Meseguer. An Algebraic Semantics for MOF. In J. L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International*

Conference, FASE 2008. Proceedings, volume 4961 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2008.

2. A. Boronat, J. Oriente, A. Gómez, I. Ramos, and J. A. Carsí. An Algebraic Specification of Generic OCL Queries Within the Eclipse Modeling Framework. In A. Rensink and J. Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2006.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science. Springer, 2007.
5. M. Clavel and M. Egea. ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams. In M. Johnson and V. Vene, editors, *AMAST*, volume 4019 of *Lecture Notes in Computer Science*, pages 368–373. Springer, 2006.
6. M. Gogolla, F. Büttner, and M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.
7. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
8. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.
9. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational Abstractions. *Theoretical Computer Science*, 403(2-3):239–264, 2008.
10. J. E. Rivera, F. Durán, and A. Vallecillo. A Graphical Approach for Modeling Time-Dependent Behavior of DSLs. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009, Corvallis, OR, USA, 20-24 September 2009, Proceedings*, pages 51–55, 2009.
11. M. Roldán and F. Durán. Dynamic validation of OCL constraints with mOdCL. In J. Cabot, R. Clarisó, M. Gogolla, and B. Wolff, editors, *International Workshop on OCL and Textual Modelling*, 2011.
12. J. R. Romero, J. E. Rivera, F. Durán, and A. Vallecillo. Formal and Tool Support for Model Driven Engineering with Maude. *Journal of Object Technology*, 6(9):187–207, 2007.