Verifying UML/OCL Operation Contracts

Jordi Cabot*, Robert Clarisó, and Daniel Riera
Universitat Oberta de Catalunya (Spain)
{jcabot,rclariso,drierat}@uoc.edu

Abstract. In current model-driven development approaches, software models are the primary artifacts of the development process. Therefore, assessment of their correctness is a key issue to ensure the quality of the final application. Research on model consistency has focused mostly on the models' static aspects. Instead, this paper addresses the verification of their dynamic aspects, expressed as a set of operations defined by means of pre/postcondition contracts.
This paper presents an automatic method based on Constraint Programming to verify UML models extended with OCL constraints and operation contracts. In our approach, both static and dynamic aspects are translated into a Constraint Satisfaction Problem. Then, compliance of the operations with respect to several correctness properties such as operation executability or determinism are formally verified.

1 Introduction

In recent years, Model Driven Development (MDD) is gaining attention due to its promise to increase productivity in developing, documenting, and maintaining software systems. MDD emphasizes the use of models during the whole development process and thus the *correctness of a model* becomes a major issue: model defects will directly become implementation defects in the final software system due to the application of code-generation techniques. Unfortunately, popular modeling notations (UML [5] being the most widely used) are not formal enough to directly prove the correctness of the software models. Therefore, a set of model-level verification techniques are needed to ensure the quality of software model specifications. Each technique can address a variety of correctness properties and goals depending on which type of models it is analyzing.

In particular, this paper presents a new method for the verification of the behavioural aspects of software models defined using the design by contract approach [20], where each operation is defined by means of a contract consisting of a *precondition* (set of conditions on the operation input) and a *postcondition* (conditions to be satisfied at the end of the operation). In conceptual modeling, this is also known as the declarative specification of an operation, in contrast to imperative specifications where the set of updates produced by the operation on the system state is explicitly defined. Our goal will be detecting defects in the definition of the operation (e.g. potential inconsistent interactions with integrity constraints) rather than checking whether an implementation fulfills the pre/postconditions. This is an extension of our previous work [9, 19] which focused only on reasoning on integrity constraints without considering operations.

* Work partly supported by the Ministerio de Educación y Ciencia, FEDER under project TIN208-00444/TIN, Grupo Consolidado and UOC-IN3 research grant.

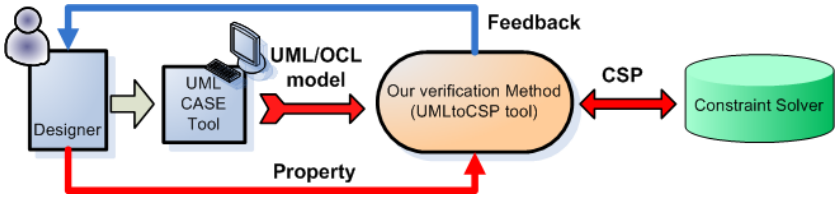


Fig. 1. Overall picture of the process.

The goal of this paper is twofold. First, we present a set of "reasonable" correctness criteria that any operation should fulfill. For example, we will try to check if a precondition is so strong that it cannot be satisfied by any state that fulfills the integrity constraints (e.g. a precondition "a $\geq$ 5" when the model includes the constraint "a $\leq$ 3" is clearly unsatisfiable). Designers can select their preferred set of criteria among the predefined set of properties we propose.

Second, we provide a method for verifying these properties on UML/OCL models. Without loss of generality, we will assume that our input model is a UML class diagram, annotated with integrity constraints, and pre/postconditions written in the Object Constraint Language (OCL) [15]. Our choice is based on the wide adoption of the UML and it high-level modeling constructs, although many concepts of this work are applicable to other modeling languages as well.

The verification will be driven by the discovery of examples/counterexamples. First, the designer selects the criteria to be checked. The model, the integrity constraints, the correctness criteria and the pre/postconditions will be transformed into a Constraint Satisfaction Problem (CSP) [2, 14] that can be solved by current Constraint Programming solvers. The solution of the CSP, if there is one, will be an example or counterexample that proves the criteria being analyzed. The example is given to the designer as a valuable feedback in the form of an object diagram (so that he/she can understand it). Our UMLtoCSP tool [19] will be used to automate the process (see Figure 1).

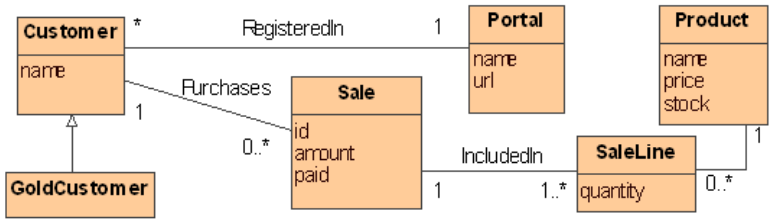
The rest of the paper is structured as follows. Section 2 introduces OCL concepts. Section 3 presents our correctness properties and Section 4 their verification with constraint programming. Tool support is commented in section 5. Section 6 discusses related work and Section 7 draws some conclusions.

2 Declarative Operations in OCL: Basic Concepts

OCL is a formal high-level language used to describe properties on UML models. It admits several powerful constructs like quantified iterators (*forAll*, *exists*) and operations over collections of objects (*union*, *select*, *includes*, ...). The pattern for specifying a declarative operation *op* in OCL is the following:

context TypeName::op(p1: Type1, ..., pN: TypeN): ResultType
pre: Boolean expression (the precondition)
post: Boolean expression (the postcondition)

Operations are always defined in the context of a specific type of the model. The *pre* and *post* clauses are used to express the preconditions and postcon-



context Product **inv** minStock: self.stock ≥ 5

context GoldCustomer **inv** salesAmount:
self.sale −>select(s | s.paid).cost −>sum() ≥ 100000

context Customer::newCustomer(name:String, p: Portal): Customer
post: result.oclIsNew() and result.name=name and result.portal=p

context Sale::addSaleLine(p: Product, quantity: Integer): SaleLine
pre: p.stock > 0
post: result.oclIsNew() and result.sale=self and result.product=p and
result.quantity=quantity and p.stock=p.stock@pre-quantity and
self.amount=self.amount@pre + quantity*p.price

context Portal::removeGoldCategory(c: Customer)
pre: c.oclIsTypeOf(GoldCustomer) and c.sale−>isEmpty()
post: not c.oclIsTypeOf(GoldCustomer)

Fig. 2. Running example: class diagram, OCL constraints and operations.

ditions of the operation contract. In the boolean expressions, the implicit parameter *self* refers to the instance of the TypeName on which the operation is applied. Another predefined parameter, *result*, denotes the return value of the operation if there is one. The dot notation is used to access the attributes of an object or to navigate from that object to the associated objects in a related type. The value of an accessed attribute or role in a postcondition is the value upon completion of the operation. To refer to the value of that property at the start of the operation, one has to postfix the property name with the keyword *@pre*.

As an example consider the diagram of Fig. 2 aimed at representing a set of web portals for selling the products of a company to a group of registered customers, some of them classified as gold customers. The model includes two textual integrity constraints and three operations. The invariant *minStock* ensures that all products have a stock of at least five units, while *salesAmount* imposes that gold customers must have paid a minimum amount of 100000 euros in sales. Regarding the operations, *newCustomer* and *addSaleLine* create a new customer and a new sale line in a sale, respectively. In OCL, the creation of an object is indicated with the operation *oclIsNew*. Operation *addSaleLine* also updates the stock of the product and the total amount of the sale. The

operator *@pre* in *p.stock=p.stock@pre - quantity* indicates that the stock of the product has been decreased by *quantity* units with respect to the previous value. *RemoveGoldCustomer* converts a gold customer with no sales into a plain one.

3 List of Correctness Properties

Pre and postconditions of declarative operations must be defined accurately, taking into account the possible interactions with the integrity constraints. For instance, preconditions which are too strong may prohibit the execution of an operation altogether (since none of the valid states of the system can satisfy the precondition). This section presents a list of properties to determine whether pre and postconditions are correctly defined.

In the definition of the correctness properties, we will use the following notation. Given a model M, let S denote a *snapshot* of M, i.e. a possible instantiation of the types defined in M. A snapshot S will be called *legal*, denoted as $\mathsf{Inv}[S]$, if it satisfies all integrity constraints of M, including all textual OCL constraints.

Given a declarative operation *op*, $\mathsf{Pre}_{op}[o, P, S]$ denotes that the precondition of *op* holds when it is invoked over an object o of an snapshot S using the values in P as argument values for the list of parameters of *op*. For the sake of clarity, we will assume that o is passed as an additional parameter in P, e.g. the first one, expressing then the preconditions simply as: $\mathsf{Pre}_{op}[P, S]$. S and P will be referred collectively as the *input* of the operation.

To evaluate the postcondition, we also need to consider the return value and the snapshot after executing the operation (considering new/deleted objects and links, updated attribute values, etc.). The final snapshot and the return value will be referred as the *output* of the operation. Then, $\mathsf{Post}_{op}[P, S \triangleright S', R]$ will denote that the postcondition of operation *op* holds when S is the snapshot before executing the operation, S' is the snapshot after executing it, P is the list of parameters and R is the return value.

According to this notation, the list of properties is defined as follows:

- **Applicability**: An operation *op* is *applicable* if the precondition is satisfiable, i.e. if there is an input where the precondition evaluates to true.

$$\exists S : \exists P : \mathsf{Inv}[S] \ \wedge \ \mathsf{Pre}_{op}[P, S]$$

- **Redundant precondition**: The precondition of an operation *op* is *redundant* if it is true for any legal input.

$$(\exists S : \mathsf{Inv}[S]) \ \wedge \ (\forall S : \forall P : \mathsf{Inv}[S] \ \rightarrow \ \mathsf{Pre}_{op}[P, S])$$

- **Weak executability**: An operation *op* is *weakly executable* if the postcondition is satisfiable, that is, if there is a legal input satisfying the precondition for which we can find a legal output satisfying the postcondition.

$$\exists S, S' : \exists P : \exists R : \mathsf{Inv}[S] \ \wedge \ \mathsf{Inv}[S'] \ \wedge \ \mathsf{Pre}_{op}[P, S] \ \wedge \ \mathsf{Post}_{op}[P, S \triangleright S', R]$$

– **Strong executability**: An operation *op* is *strongly executable* if, for every legal input satisfying the precondition, there is a legal output that satisfies the postcondition.

$$\forall S : \forall P : \exists S' : \exists R : (\mathsf{Inv}[S] \ \wedge \ \mathsf{Pre}_{op}[P, S]) \ \rightarrow \ (\mathsf{Inv}[S'] \ \wedge \ \mathsf{Post}_{op}[P, S \triangleright S', R])$$

– **Correctness preserving**: An operation *op* is *correctness preserving* if, given a legal input, each possible output satisfying the postcondition is also legal.

$$\forall S, S' : \forall P : \forall R : (\mathsf{Inv}[S] \ \wedge \ \mathsf{Pre}_{op}[P, S]) \ \rightarrow \ (\mathsf{Post}_{op}[P, S \triangleright S', R] \ \rightarrow \ \mathsf{Inv}[S'])$$

– **Immutability**: An operation *op* is *immutable* if, for some input, it is possible to execute the operation without modifying the initial snapshot.

$$\exists S : \exists P : \exists R : \mathsf{Inv}[S] \ \wedge \ \mathsf{Pre}_{op}[P, S] \ \wedge \ \mathsf{Post}_{op}[P, S \triangleright S, R]$$

– **Determinism**: An operation *op* is *non-deterministic* if there is a legal input that can produce two different legal outputs, e.g. different result values or different final snapshots.

$$\begin{aligned} \exists S, S'_1, S'_2 : \exists P : \exists R_1, R_2 : \mathsf{Inv}[S] \ \wedge \ \mathsf{Inv}[S'_1] \ \wedge \ \mathsf{Inv}[S'_2] \ \wedge \ \mathsf{Pre}_{op}[P, S] \ \wedge \\ \mathsf{Post}_{op}[P, S \triangleright S'_1, R_1] \ \wedge \ \mathsf{Post}_{op}[P, S \triangleright S'_2, R_2] \ \wedge \\ (\ (S'_1 \neq S'_2) \ \vee \ (R_1 \neq R_2)\ ) \end{aligned}$$

Studying these properties in the running example, we have, for instance, that the precondition of *addSaleLine* is redundant since it is subsumed by the integrity constraint *minStock*. Also, *addSaleLine* is weakly executable but not strongly executable: for those states where *p.stock-quantity*<5 the final state will violate the invariant *minStock*. The precondition of *removeGoldCategory* is not applicable since constraint *salesAmount* forces all gold customers to be related to at least a sale. Finally, *newCustomer* is strongly executable but not correctness preserving as it might create a gold customer (instead of a plain one) with no sales, violating *salesAmount*.

It is important to remark that, in general, designers define underspecified postconditions [20]. This means that, given an operation contract, there are usually several final states that satisfy its postcondition. Therefore, most operation contracts will be flagged by our analysis as non-deterministic. To improve the accuracy of the results, designers may want to provide postconditions which are precise enough to characterize the exact set of desired final states. For basic postcondition expressions, an educated guess of the designer's intention can be inferred by analyzing the initial ambiguous postcondition [6, 8], and thus, it would be possible to automatically generate a set of additional conditions to define more precisely the desired final state. This is left as further work.

4 Verifying Operations with Constraint Programming

This section presents a systematic and automatic procedure to verify correctness properties of operation contracts using the constraint programming paradigm.

Constraint programming [2, 14] is a declarative approach for describing and solving problems. A problem in constraint programming, called *constraint satisfaction problem* (CSP), is defined as a finite set of *variables*, *domains* (one per variable) and *constraints* over the variables. A *solution* to a CSP is an assignment of values to variables that satisfies all constraints, with each value within the domain of the variable. Constraint programming solvers use efficient backtracking-based techniques to automatically explore the search space and find solutions to the CSP. To ensure termination, the search space must be finite, thus, all variable domains must be finite.

The key idea of our approach is to translate the model, together with its integrity constraints, the desired correctness property and the operation to verify, into a CSP such that by checking whether the generated CSP has a solution we can determine if the operation satisfies the property. Both the translation procedure and the search of a solution for the CSP (performed using existing CSP solvers) are completely automatic and, therefore, all the verification process is transparent to the designer.

In short, with our translation procedure, the set of variables in the generated CSP characterize a possible snapshot of the model, i.e. the variable values represent the objects of the snapshot, their attributes values, their relations, etc. Its constraints ensure that the variable values (i.e. the snapshot) are consistent with the implicit structural UML constraints (e.g. all objects in a subtype must be also instance of its supertype), graphical constraints (e.g. multiplicities) and textual OCL constraints. Pre and postconditions of operations and correctness properties are translated as additional constraints.

Given this set of variables, domains and constraints, the final CSP is organized as a sequence of subproblems to be solved by the constraint solver in order to find a solution for the CSP, and thus, prove the desired correctness property. The exact combination of these subproblems in the CSP depends on the chosen property. For properties regarding the operation precondition, the resolution of the CSP first searches for a legal snapshot which satisfies the operation precondition (this, for instance, proves the *applicability* of the operation). If no solution is found, the solver concludes that the property is not satisfied. For properties involving postconditions, once we have a legal instance that satisfies the precondition, the solver must search for a second legal snapshot that satisfies the postcondition (see Figure 3). As we will see, for some properties we will search for solutions that falsify the pre/postcondition expressions instead.

The following subsections explain the encoding of the UML class diagram, the OCL constraints and the operations' pre and postconditions in the CSP and how they are combined, depending on the selected correctness property, to generate the final CSP that will be used to prove the property. The first two steps are a short summary of our previous work [9].

Without loss of generality, in our presentation we use the Prolog-based CSP formalism used by the constraint solver $\mathsf{ECL^iPS^e}$ [2]. Due to space limitations, only some translation excerpts can be shown. The full translation for our running example can be found in [19].

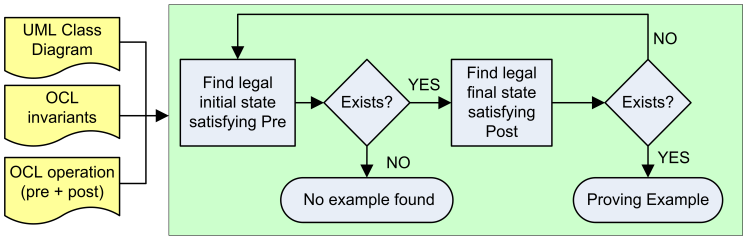


Fig. 3. Analysis of the weak executabiliy property.

4.1 Translation of the UML class diagram

A class diagram consists of a set of classes, associations and generalisation sets. Each element must be translated into a corresponding set of variables, domains and constraints in the CSP. Appropriate domains for each variable can be provided by the designer as part of the translation process or default values can be used.

For each class C, our translation creates a $SizeC$ variable to record the number of instances of C in the snapshot, a list variable $InstancesC$ to hold the C instances, where each element in the list is of type $struct(C) = (oid, f_1, \ldots, f_n)$, where: oid represents the explicit object identifier for each object and each field f_i corresponds to an attribute of C.

Similarly, for each association As the translation creates a $SizeAS$ variable to record the number of links (i.e. association instances) in As and a list variable $InstancesAs$ to store the links, where each element is of type $struct(As) = (p_1, \ldots, p_n)$, where $p_1 \ldots p_n$ are the role names of the participant classes. Each concrete participant is identified by its oid.

Generalizations do not imply the definition of new variables but of new constraints among the classes involved in the generalisation set. Additional constraints to control the cardinality constraints or the uniqueness of oid values, among others, are also defined in the CSP.

4.2 Translation of OCL invariants

Each OCL integrity constraint (*invariants* in the UML terminology) results in a new constraint in the CSP that restricts the possible assignment of values to the CSP variables, i.e. it limits the possible set of legal snapshots of the model.

OCL invariants are boolean OCL expressions defined in the context of a specific type of the model and that must be satisfied by all instances of that type, in other words, the invariant is universally quantified over the objects of the type. Therefore, the translation must ensure that the boolean condition of the invariant (its body) is satisfied by each individual object, i.e. by each possible value of the *self* variable. For instance, the invariant *minStock* (**context** Product **inv** minStock: self.stock $\geq$ 5) would be translated[1] into the rule depicted in

[1] To favour the readability of the rules some technical details are omitted.

```
invariantMinStock(Snapshot) :-
    % Get the list of Objects in Snapshot of type Product
    getObjects(Snapshot, ''Product'', Objects),
    ( foreach(Object, Objects) do   % Iterate over all objects
      % Evaluate the invariant expression using this object as ''self''
      evalRootMinStock(Snapshot, [Object], Result),
      % The invariant must evaluate to true
      Result #=1).
```

```
evalRootMinStock( Snapshot, Vars, Result ) :-
    attribStock( Snapshot, Vars, X ), % X = attrib value
    const5( Snapshot, Vars, Y ),      % Y = constant
    #>=(X, Y, Result).                % Result = X >= Y
const5( _, _, Result ):- Result #= 5.
```

Fig. 4. Translation of the invariant *minStock* (top) and some subexpressions (bottom).

Figure 4. This rule fails when the given snapshot contains a product with a too low stock. An auxiliary rule, *evalRootMinStock*, is responsible for checking this condition body on each object. The failure of the rule determines that the snapshot is not legal, and thus, forces the solver to backtrack and try a different combination of variable assignments.

The translation of the body conditions proceeds as follows. The OCL body expression is analyzed using a metamodel-based representation of the expression where each element (operator, variable, constant, method call, ...) is automatically defined as instance of the appropriate class in the OCL metamodel. Intuitively, an instance of the OCL metamodel for an OCL expression is the equivalent of an annotated syntax tree for the expression. Internal nodes correspond to operators, while the leaves of the tree are constants and variables. The information annotated on each node depends on its type as, for instance, the specific OCL operator, the value of the constant or the identifier of a variable.

The transformation of an OCL expression tree into an $\mathsf{ECL^iPS^e}$ CSP is performed by traversing the tree in postorder and translating each visited node into one Prolog rule with an unique name. For instance, in the invariant *minStock*, *evalRootMinStock* refers to the rule created for the topmost node of the *minStock* invariant body expression. Therefore, the transformation can be fully characterized by describing the Prolog rule that corresponds to each type of node in the OCL metamodel.

Prolog rules for OCL expressions follow the pattern:

```
rule-name( Snapshot, Variables, Result ) :- rule-body.
```

where `rule-name` is the unique name of the rule, `Snapshot` and `Variables` are the input arguments and `Result` stores the output of the expression. Intuitively, `Snapshot` is the snapshot of the model where the expression is evaluated.

`Variables` is the list of variables visible in the scope of the expression, e.g. the *self* variable and variables introduced by previous quantifiers due to iterator expressions like *forAll*. In the `rule-body` we specify the sequence of predicates that describe the relationship between the inputs and the output. A typical body evaluates the subexpressions (using their Prolog rule) and computes the output from those intermediate results.

As an example, let us consider the body of the invariant *minStock* (self.stock $\geq$ 5), which contains four subexpressions: a variable (self), an attribute access (stock), a constant (5) and a relational operator ($\geq$). The rules for the last two expressions are depicted in Figure 4 (see [9, 19] for more examples). For more complex OCL operators and iterator expressions we have already implemented a parametrized library of Prolog rules (available in [19]) that maps the semantics of each predefined OCL construct.

4.3 Translation of OCL operation contracts

Operations introduce new challenges in this translation: the list of parameters of the operation, the result value, and the complexity of studying two snapshots at once when analyzing postconditions.

Translation of preconditions The boolean OCL expression of a precondition is basically translated following the same procedure explained above for the translation of invariant bodies. However, there are two differences regarding how and when the precondition expression is evaluated: the *parameters* and the *quantification*.

In the analysis of a precondition, it is necessary to consider the possible value of the operation parameters. For parameters of a basic type (integer, float, boolean, string) designers must define their possible finite domain, for instance defining a lower and upper bound. Parameters whose type is one of the classes of the model (as the *self* parameter) can only refer to an object existing in the snapshot, so their value is already constrained by the valid instances of the snapshot where the operation is invoked. When evaluating a precondition, parameters become additional variables of the CSP, and their values are discovered by the solver as a part of the search for a solution to the CSP. For instance, when checking the applicability of an operation, the solver will automatically try several possible combinations of parameter values until it finds a combination (if any) that satisfies the Prolog rule generated for the precondition.

Contrary to invariants, properties on preconditions only require to find a single combination of a valid state and a possible assignment for the operation parameters that satisfy the precondition. Therefore, preconditions will be translated into a rule which simply evaluates the precondition body, invoking the rule for the topmost operator. To ensure that the rules for the precondition body have access to all parameter values during the rule evaluation, the list of visible variables for these rules (second argument of the Prolog rule) is initialized with the list of parameter values. In this way, accessing a parameter within the expression is equivalent to accessing any other variable: the rule only needs to be

aware of the position of each parameter in the variables list. As an example, the precondition rule for *addSaleLine* will be defined as follows:

```
preconditionAddSaleLine(Snapshot, Parameters, Result) :-
    % Result = truth value of evaluating the precondition
    evalRootExpr(Snapshot, Parameters, Result).
```

where *evalRootExpr* represents the rule for the root node of the precondition expression. The output *Result* value, reporting whether the given input (i.e. the *self* object plus the other parameters) satisfies the precondition, will be used later on to determine the satisfaction of correctness properties for the operation.

Translation of postconditions Two new factors in the translation of postconditions are the return value and the relationship between the two snapshots representing the initial and final states.

In our translation, the return value will simply become another variable in the list of visible variables, just like *self* or the other parameters in the precondition.

Relationships between the initial and the final state are expressed by means of the *oclIsNew* and, specially, the *@pre* OCL operators. *OclIsNew* highlights that an object should exist in the final state but not in the initial one; and *@pre* is used to retrieve the value of a subexpression in the initial state. Thus, the Prolog implementation of these two operators needs to receive an additional argument: the snapshot for the initial state. To avoid changing the general rule pattern due to this extra argument, this initial state is stored in the global variable `initialstate`. This variable will be conveniently accessed within the subrules for these two operators. Translation of all other OCL operators in the postcondition expression is not changed from previous translations steps. They are just evaluated on the particular snapshot given as argument to their Prolog rule, it does not matter if it represents the initial or the final state.

To sum up, the definition of the rule for the postcondition of the operation *addSaleLine* is shown in Figure 5. The *initialstate* variable will then be used in the rules evaluating *oclIsNew* and *@pre* nodes appearing in postcondition expressions. We provide the rule for *oclIsNew* as an example in Figure 6. It determines if the object with the *Oid* value given as an argument is an object that did not exist before executing the operation.

4.4 Translation of correctness properties

As a last step, each correctness property (or its negation) is translated as a new CSP constraint restricting the *result* values returned by the pre and postcondition rules such that finding a solution to the CSP with this new constraint suffices to prove the property.

Whether to use the property or its negation depends on the quantification used in the property formalization, *existential* or *universal* (see Section 3). Existentially quantified properties can be *proved* by finding an *example*, i.e. a case

```
:-  local reference(initialstate).
postconditionAddSaleLine(InitialState, FinalState,
                         Parameters, RetValue, Result) : -
    % Add the return value and parameters to the list of visible vars
    append([RetValue], Parameters, Variables),
    % Store the initial state, needed in oclIsNew and @pre nodes
    setval(initialstate, InitialState),
    % Result = truth value of evaluating the postcondition
    evalRootExpr(FinalState, Variables, Result).
```

Fig. 5. Translation of the OCL postcondition of operation *addSaleLine*.

```
ocl_isNew(FinalState, Oid, TypeName, Result) :-
    % Recover the initial state from the global variable
    getval(initialstate, InitState),
    % Get the list of objects before and after the operation
    getObjects(InitState, TypeName, ObjectsBefore),
    getObjects(FinalState, TypeName, ObjectsAfter),
    % Check if Oid exists before/after the operation
    existsObjectWithOid(ObjectsBefore, Oid, ExistsBefore),
    existsObjectWithOid(ObjectsAfter, Oid, ExistsAfter),
    % Result = ExistsAfter and not ExistsBefore
    and( ExistsAfter, neg ExistsBefore, Result).
```

Fig. 6. Translation of the OCL operator *oclIsNew*.

where the property is satisfied. For example, applicability can be proved by finding a legal input that satisfies the precondition. Universally quantified properties can be *disproved* by finding a *counterexample*. For instance, redundancy can be disproved by finding a legal snapshot that does not satisfy the precondition Similarly, the lack of (counter)examples can be used to (dis)prove the property.

The selected property also influences how the final CSP is organized as a combination of the rule excerpts generated during the previuos translation steps. For properties on preconditions, postcondition rules are not included. For properties on postconditions, the CSP is split up into two subproblems (see Figure 3). The first one (*findInitialState*) tries to find a legal snapshot that satisfies the precondition rule. This initial snapshot is then given as an argument to the second subproblem (*findFinalState*), in charge of finding a second legal snapshot satisfying (or not) the postcondition to prove the property. As an example, Figure 7 sketches the final CSP to determine whether *addSaleLine* is weakly executable. Other properties imply adding new constraints/subproblems to the CSP. For instance, immutability requires a new constraint imposing the equality between the initial and final states.

```
weakExecutabilityAddSaleLine(Example) :-
    Example = [InitState, FinalState, Parameters, RetValue],
    findInitialState(InitState, Parameters),
    findFinalState(InitState, FinalState, Parameters, RetValue).
findInitialState(InitState, Parameters) :-
    % Definition of variables, domains, graphical integrity constraints
    % Textual integrity constraints
    invariantMinStock(InitState), invariantSalesAmount(InitState),
    % Precondition
    preconditionAddSaleLine(InitState, Parameters, ResultOfPre),
    ResultOfPre #= 1, % Weak executability
    % Now find a solution satisfying all these constraints
    labeling([InitState, Parameters]).
findFinalState(InitState, FinalState, Parameters, RetValue) :-
    % Definition of variables, domains, graphical integrity constraints
    % Textual integrity constraints
    invariantMinStock(FinalState), invariantSalesAmount(FinalState),
    % Postcondition
    postconditionAddSaleLine(InitState, FinalState, Parameters,
                             RetValue, ResultOfPost),
    ResultOfPost #= 1, % Weak executability
    % Now find a solution satisfying all these constraints
    labeling([FinalState, RetValue]).
```

Fig. 7. CSP generated for checking weak satisfiability of *addSaleLine*. The *labeling* operator is a possible backtracking implementation offered by the constraint solver that attempts to assign values to the given list of input variables. If the assignment does not satisfy all the stated CSP constraints preceding the labeling, a new assignment is tried until the solver finds a solution or determines that no solution exists.

5 Tool Support

The verification method presented in this paper is being implemented as an extension of our UMLtoCSP tool [19]. Given an UML/OCL model and a correctness property P, the tool determines whether the model satisfies P and shows graphically example snapshots that prove/disprove it. For instance, Fig. 8 shows the counterexample provided by UMLtoCSP when analyzing whether *addSaleLine* is correctness preserving: there is a legal input (the snapshot on the left satisfies the invariants and precondition) with an illegal output (the snapshot on the right satisfies the postcondition but not the invariant *minStock* as there are only 4 items of $Product1$). The translation from the model to the CSP, the search of the counterexample snapshots and the graphical depiction are performed automatically by UMLtoCSP. See [19] for more details and examples.

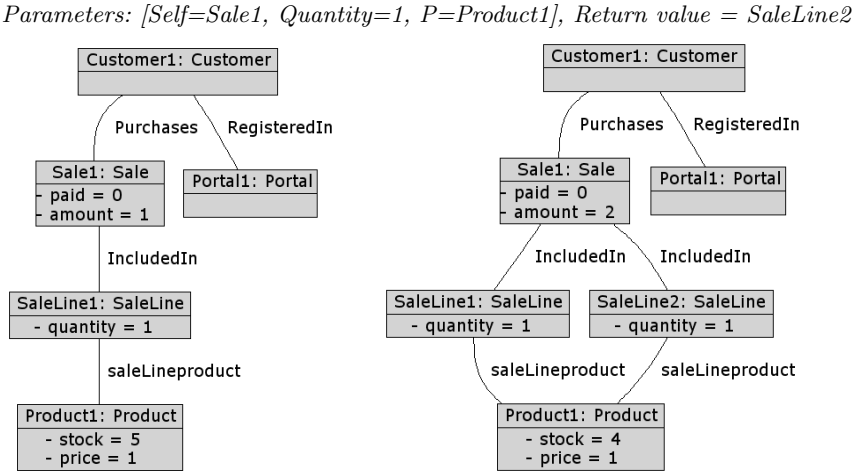


Fig. 8. Counterexample proving that *addSaleLine* is not correctness preserving: initial state (left), final state (right), parameter values and return value (top). The final state violate the invariant *minStock*.

6 Related Work

Typically, approaches devoted to the verification of UML/OCL models (as [1, 4, 7, 10, 13, 16, 17] or our own approach among others) transform the diagram into a formalism where efficient solvers or theorem provers are available. However, there are complexity and decidability issues to be considered. Reasoning on UML class diagrams is EXPTIME-complete [3], and, when general OCL constraints and/or operations are allowed, it becomes undecidable.

By choosing a particular formalism, each method commits to a different trade-off (expressivity vs termination vs automation vs completeness) for the verification process. In what follows we compare the features of methods supporting the verification of declarative operations, a small subset of the ones listed above, with this paper.

HOL-OCL [7] embeds OCL into the higher-order logic (HOL) instance of the interactive theorem prover Isabelle. It supports the full OCL expressivity but it requires user-interaction to complete proofs, and thus, it is not automatic.

The UMLtoAlloy tool [1] proposes an automatic translation of UML/OCL to Alloy [12]. Alloy is a mature tool for the automated analysis of software specifications that works by transforming the entire problem, including operation specifications, into an instance of SAT (satisfiability of a boolean formula in conjunctive normal form). However, the translation in [1] is only partial and Alloy itself presents some limitations, such as the need explicitly identify which model elements are modified by an operation or limited support for arithmetic operations. Thus, the usefulness of Alloy for verifying high-level UML/OCL specifications is somewhat limited.

Recent results [11] have extended the description logics formalism (in short, a decidable subset of first-order logic) to define and reason on operation contracts.

However, these approaches need to restrict the constructs that may appear in the model to keep the reasoning decidable. Thus, most OCL operations cannot be translated into this formalism.

Previous approaches based on constraint programming like [10, 13] did not admit any kind of OCL expressions. Our previous work in [9, 19] was limited to OCL invariants and did not support the analysis of declarative operations.

In contrast, the new approach presented in this paper is fully automatic, expressive and decidable. We believe that these three characteristics are key features in order to extend the adoption of formal methods among the modeling community. As a trade-off, our method is not complete: results are only conclusive if a solution to the CSP (the example/counterexample) is found. However, the absence of solutions within a finite search space cannot be used as a proof: a solution may still exist outside that search space.

Although this may limit the applicability of our method, we believe an efficient decidable procedure provides more useful information than a semidecidable procedure, even if the answer is not conclusive. Moreover, when checking the correctness of a model, most errors can be found even if we bound the search space of the verification process. This "small scope" hypothesis, i.e. that it is possible to prove interesting properties about models by focusing only on small instantiations, is shared by other bounded methods [12]. Moreover, if desired, it is still possible to use our method on infinite domains [2] resulting in a complete but semidecidable method (for properties that can be satisfied by *finite* instances).

Our approach can be complemented with other verification approaches addressing other kinds of behavioural UML diagrams (as state machines [18]) in order to provide more global results.

7 Conclusions and Further Work

We have presented a new automatic method for the formal verification of declarative operations in UML/OCL models. We believe our approach can be used to leverage current UML/OCL verification approaches, more focused on the verification of the static parts of the model.

Regarding efficiency, the search space for examples/counterexamples depends on the size of the model, so scalability quickly becomes an issue even when using sofisticated constraint solvers. As a further work, we plan to improve the search process in several ways. First, we would like to refine our translation process by considering implicit semantics in the initial contract specification (as the *nothing else changes assumption*). Also, we plan to work on the automatic inference of variable domains, discovered by a static analysis of the OCL constraints to tune the solving process. Furthermore, we are considering the abstraction of information from the model which is not relevant to the operation being verified and the relevant subset of integrity constraints.

We also plan to explore the verification of dynamic aspects of the model when specified in combination with other constructs or UML diagrams like sequence diagrams or state machines and the benefits of porting these techniques to other design by contract languages as JML, Eiffel or Spec#.

References

1. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *MODELS'07*, volume 4735 of *LNCS*, pages 436–450, 2007.
2. K. R. Apt and M. G. Wallace. *Constraint Logic Programming using* ECL$^{\text{i}}$PS$^{\text{e}}$. Cambridge University Press, Cambridge, UK, 2007.
3. A. Artale, D. Calvanese, R. Kontchakov, V. Ryzhikov, and M. Zakharyaschev. Reasoning over extended ER models. In *ER'07*, volume 4801 of *LNCS*, pages 277–292, 2007.
4. D. Berardi, D. Calvanese, and G. D. Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168:70–118, 2005.
5. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
6. A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Trans. Software Eng.*, 21(10):785–798, 1995.
7. A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.
8. J. Cabot. From declarative to imperative UML/OCL operation specifications. In *ER'07*, volume 4801 of *LNCS*, pages 198–213, 2007.
9. J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW'08*, pages 73–80, 2008.
10. M. Cadoli, D. Calvanese, G. D. Giacomo, and T. Mancini. Finite satisfiability of UML class diagrams by Constraint Programming. In *DL'2004*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.
11. C. Drescher and M. Thielscher. Integrating action calculi and description logics. In J. Hertzberg, M. Beetz, and R. Englert, editors, *KI*, volume 4667 of *LNCS*, pages 68–83. Springer, 2007.
12. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.
13. H. Malgouyres and G. Motet. A UML model consistency verification approach based on meta-modeling formalization. In *SAC'2006*, pages 1804–1809. ACM Press, 2006.
14. K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Massachussetts, 1998.
15. Object Management Group. *UML 2.0 OCL Specification*, 2003.
16. A. Queralt and E. Teniente. Reasoning on UML class diagrams with OCL constraints. In *ER'06*, volume 4215 of *LNCS*, pages 497–512, 2006.
17. R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *UML'03*, volume 2863 of *LNCS*, pages 326–340. Springer, 2003.
18. E. Turner, H. Treharne, S. Schneider, and N. Evans. Automatic generation of CSP B skeletons from xuml models. In *ICTAC*, volume 5160 of *LNCS*, pages 364–379, 2008.
19. UMLtoCSP. A tool for the formal verification of UML/OCL models based on Constraint Programming. http://gres.uoc.edu/UMLtoCSP.
20. R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv.*, 30(4):459–527, 1998.