Automatic and efficient simulation of operation contracts

Matthias P. Krieger, Alexander Knapp, Burkhart Wolff

Angaben zur Veröffentlichung / Publication details:

Krieger, Matthias P., Alexander Knapp, and Burkhart Wolff. 2011. "Automatic and efficient simulation of operation contracts." In *Proceedings of the GPCE '10 Proceedings of the ninth international conference on Generative programming and component engineering, Eindhoven, The Netherlands — October 10 - 13, 2010*, edited by Eelco Visser and Jaakko Järvi, 53–62. New York, NY: Association for Computing Machinery (ACM). https://doi.org/10.1145/1942788.1868303.

Automatic and Efficient Simulation of Operation Contracts

Matthias P. Krieger *
Université Paris-Sud
krieger@lri.fr

Alexander Knapp
Universität Augsburg
knapp@informatik.uni-augsburg.de

Burkhart Wolff*
Université Paris-Sud
wolff@lri.fr

Abstract

Operation contracts consisting of pre- and postconditions are a well-known means of specifying operations. In this paper we deal with the problem of operation contract simulation, i.e., determining operation results satisfying the postconditions based on input data supplied by the user; simulating operation contracts is an important technique for requirements validation and prototyping. Current approaches to operation contract simulation exhibit poor performance for large sets of input data or require additional guidance from the user. We show how these problems can be alleviated and describe an efficient as well as fully automatic approach. It is implemented in our tool OCLexec that generates from UML/OCL operation contracts corresponding Java implementations which call a constraint solver at runtime. The generated code can serve as a prototype. A case study demonstrates that our approach can handle problem instances of considerable size.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications

General Terms Algorithms, Performance

1. Introduction

Formal specifications are a well-known intermediate step between the requirements document of a system and its implementation. They state precisely and unambiguously the requirements that the system has to meet. At the same time, a declarative specification can be substantially more concise than an algorithmic description.

Methodologically, it is essential that errors and misunderstandings during requirements analysis are discovered as soon as possible. It is much easier to correct such errors before they have lead to a faulty implementation. Being among the first machine-readable artefacts produced, formal specifications are an interesting target for automated analysis aiming at requirements validation. A valuable approach to checking a specification is to simulate it, i.e., to perform computations that comply with the specification based on user-provided input data. This technique is also called animation [14].

If support for simulation is available, users can validate requirements by simulating the specification on sample sets of input data (*scenarios*). This is already possible before implementation of the system. For incomplete, faulty or inadequate specifications, simulation will typically lead to strange and alarming results. In contrast to automatically generated test cases, simulation based on user-supplied input avoids scenarios that are too artificial. Simulation can help users gain confidence in the specification by allowing the execution of scenarios that are common for the application domain.

Simulation is particularly powerful if it is accomplished by generating a prototype implementation. The generated code can be linked with components of the system that are already finished. This allows the system to be tested as a whole, although some of its parts are not yet available. It may even be possible to entirely omit the manual implementation of certain operations if they can be animated efficiently enough.

The contributions of this paper are an improved simulation method and its implementation in the tool OCLexec[1] which animates specifications written in the Object Constraint Language (OCL [30]). OCL is a textual language complementing UML. Our simulation method processes operation contracts, which are a major constituent of many formal specifications. Operation contracts consist of pre- and postconditions that express under which circumstances an operation may be called and what a correct implementation of it needs to accomplish. Besides OCL, a prominent language for formal operation contracts is the Java Modeling Language (JML [8]), an annotation language for Java programs.

For simulating an operation contract, the user supplies a system state in which the operation is called and provides any arguments to the operation. Simulation yields a new state and if necessary a return value satisfying the postconditions and any other restrictions stated in the specification such as class invariants. The challenge of animation is to compute a concrete post-state although postconditions only express abstractly which states are *admissible* but not necessarily how they can be constructed. We think that simulation support for UML/OCL operation contracts is a useful complement to existing methods for executing other UML model constituents like statecharts or action language code.

Analysis methods for operation contracts are often based on a translation of the specification language to a representation that can be processed with a constraint solver. We observe that existing constraint solving approaches for operation contracts have some significant drawbacks, in particular concerning their suitability for simulation. One fundamental problem with such approaches is that they rely on user-supplied bounds on integer values, numbers of class instances and collection sizes. These bounds are necessary in order to sufficiently instantiate quantified constraints. This is clearly a burden and, in particular, a considerable obstacle to the integration of animated operations with other code. As a solution, we propose a simplified intermediate representation for constraints that allows for a unified treatment of the different kinds of bounds. We observe that some values do not need to be bounded. For the

* This work was partially supported by the Digiteo Foundation.

GPCE'10, October 10–13, 2010, Eindhoven, The Netherlands.

[1] http://www.pst.ifi.lmu.de/Research/current-projects/oclexec

remaining values we try increasingly larger ranges until animation results can be obtained.

Efficiency is another aspect of constraint solving that is essential for simulation. Note that approaches for other types of analysis have somewhat different performance objectives. Test case generation, for example, usually only requires system states that are just large enough to satisfy a certain property. When simulating, however, users may well want to call an operation in a complex state with numerous objects. Common constraint solving approaches exhibit poor efficiency for these kinds of problems. We present optimization techniques that enable an animator to cope with system states of considerable size.

The paper is organized as follows. In Section 2 we present simulation of UML/OCL operation contracts by means of a case study. Section 3 describes a preliminary analysis of operation contracts that narrows down the set of classes for which new instances may need to be created and the set of constraints that need to be considered for animation. Section 4 introduces arithmetic formulas with bounded quantifiers as an intermediate representation of constraints. The translation of OCL constraints to arithmetic formulas is outlined. In Section 5 we describe our constraint solving approach for arithmetic formulas with bounded quantifiers and show how it is made suitable for simulation. Section 6 gives experimental results for OCLexec. After reviewing related work in Section 7, we conclude and present some ideas for future work in Section 8.

2. A Case Study

In this section we present an example of a specification that could benefit from simulation.

2.1 The Task

Figure 1 shows an excerpt from a possible UML model of a company. Employees are temporarily assigned to customers to carry out the customers' orders. Customers specify the skills that they would like the employee to have for handling their order (association end `requestedSkills`). Also, employees may give a list of customers that they prefer to work for (attribute `preferredCustomers`).

A task of the system which we are specifying is to perform an adequate assignment of employees to customers. What is sought is an assignment that respects the preferences of both the customers and the employees. This kind of assignment problem can be regarded as an instance of the prominent stable marriage problem [17]. The term *stable marriage* is inspired by the idea of matching men to women in a consistent manner. It is well-known that if the numbers of men and women are equal, it is always possible to find a *stable* assignment, i.e., an assignment in which no man and woman leave their assigned partners in order to form a new couple because they both prefer their new partner to the one that was assigned to them.

2.2 OCL in a Nutshell

Figure 2 shows an OCL operation contract of an operation `assignNewCustomers` for performing a stable assignment of employees to customers. OCL is a textual language complementing UML that can be used for specification tasks that are difficult or impossible to accomplish with UML diagrams alone. OCL has a variety of applications at different modeling levels. Here we use OCL for querying and constraining UML system states. The evaluation of OCL expressions is free of side effects. OCL provides convenient means for navigating across associations and retrieving objects. In addition to user-defined classes and the primitive types `Boolean`, `Integer`, `Real` and `String`, OCL offers the collection types `Set`, `Bag` (i.e., multiset), `Sequence` and `OrderedSet`. The OCL standard library provides numerous collection operations, including comprehensions (called `collect` in OCL) and quantification over collections. Due to this focus on collection manipulation, OCL somewhat resembles SQL.

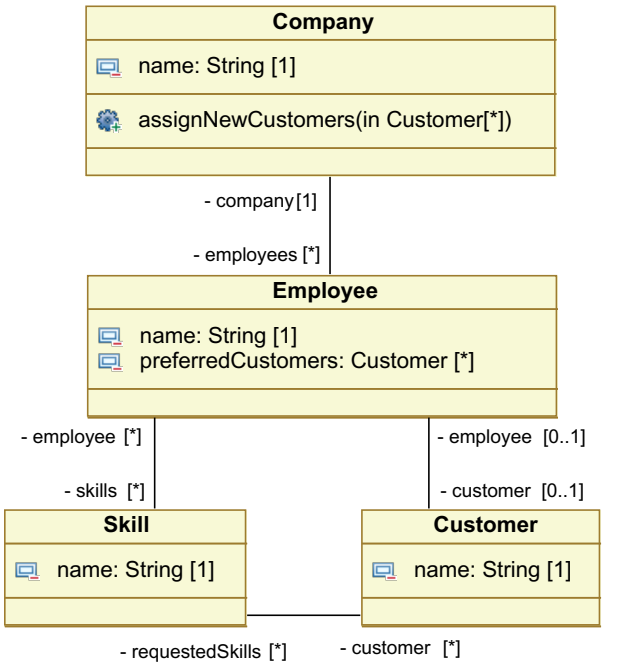


Figure 1. Excerpt from a possible UML model of a company

Pre- and postconditions are OCL expressions of type `Boolean`. A precondition has to be fulfilled when the operation is called and a postcondition has to hold when the operation returns. An OCL operation contract consists of an arbitrary number of pre- and postconditions. Another specification instrument are class invariants that have to always hold for all objects of a certain class.

2.3 Anatomy of the Operation Contract

Since the the operation contract in Figure 2 is nontrivial, we explain why it expresses the requirements. The precondition of the operation contract states that there are at least as many available employees, i.e., employees that are currently not assigned to a customer, as customers that are supposed to be matched. This condition is obviously necessary for the existence of any assignment of available employees to all new customers. As mentioned above, this condition is also sufficient for the existence of a stable assignment. In the precondition, we use the built-in operation `oclIsUndefined` for testing whether the value of the `customer` attribute of an `Employee` object is `null` or a reference to a `Customer` object. Using this test, we can form the set of available employees and apply the built-in operation `size` to it.

The first postcondition of the operation contract asserts that after completion of the operation all customers have in fact been assigned to available employees. In this postcondition, first the set of employees that were available in the pre-state but are no longer available in the post-state is defined. Then we use the `collect` comprehension of OCL to obtain the collection of customers that are assigned to this set of employees. This collection is a bag, since OCL semantics is based on the general case that several employees may be assigned to the same customer, although this is excluded by the multiplicities in the class diagram. We use the built-in operation

```
context Company::assignNewCustomers(newCustomers: Set(Customer)):

pre enoughEmployees: employees->select(customer.oclIsUndefined())->size() >= newCustomers->size()

post allCustomersAssigned:
   employees@pre->select(customer@pre.oclIsUndefined() and not customer.oclIsUndefined())
                ->collect(customer)->asSet() = newCustomers

post assignmentStable:
   employees@pre->select(customer@pre.oclIsUndefined())
      ->forAll(e | newCustomers->forAll(c |
         let
            matchedSkills : Set(Skill) = c.requestedSkills@pre->intersection(c.employee.skills@pre),
            potentialSkills : Set(Skill) = c.requestedSkills@pre->intersection(e.skills@pre)
         in
            (potentialSkills->includesAll(matchedSkills) implies potentialSkills = matchedSkills)
         or
            (e.preferredCustomers@pre->includes(c)
                implies e.preferredCustomers@pre->includes(e.customer))))

modifies only: employees->select(customer.oclIsUndefined())::customer, newCustomers::employee
```

Figure 2. Operation contract for assigning new customers to available employees

`asSet` to convert the bag to a set, so it can be compared to the set of new customers.

The second postcondition asserts that the assignment performed by the operation is stable. We quantify over all pairs `e`, `c` of available employees and new customers and consider the employee assigned to the customer (`c.employee`) as well as the customer assigned to the available employee (`e.customer`). This postcondition rules out that the pair `e`-`c` is a better match than both `c`-`c.employee` and `e`-`e.customer`. It does so by stating that the skills potentially provided by employee `e` to customer `c` are not a proper superset of the skills provided by `c.employee` to customer `c`, or that employee `e` also prefers `e.customer` if employee `e` lists customer `c` as preferred.

To complete the operation contract, we still need to specify which attribute values may be changed by the operation. We do this by adding a so-called modifies only clause which states that the operation may only modify the attribute `customer` for the available employees and the attribute `employee` for the new customers. All other attribute values must be left unchanged by the operation. Modifies only clauses have not yet been incorporated into the OCL standard. The kind of modifies only clauses we use here has been proposed in [24]. In the model, we represent modifies only clauses within a UML profile in order to achieve a standardized syntax that is compatible with other OCL tools.

We have now obtained an operation contract that precisely reflects the requirements. Note that the contract is underspecified, i.e., it does not prescribe a unique result, but allows the operation to perform any stable assignment. Moreover, the contract does not indicate how such an assignment can be found.

2.4 Simulating the Operation Contract

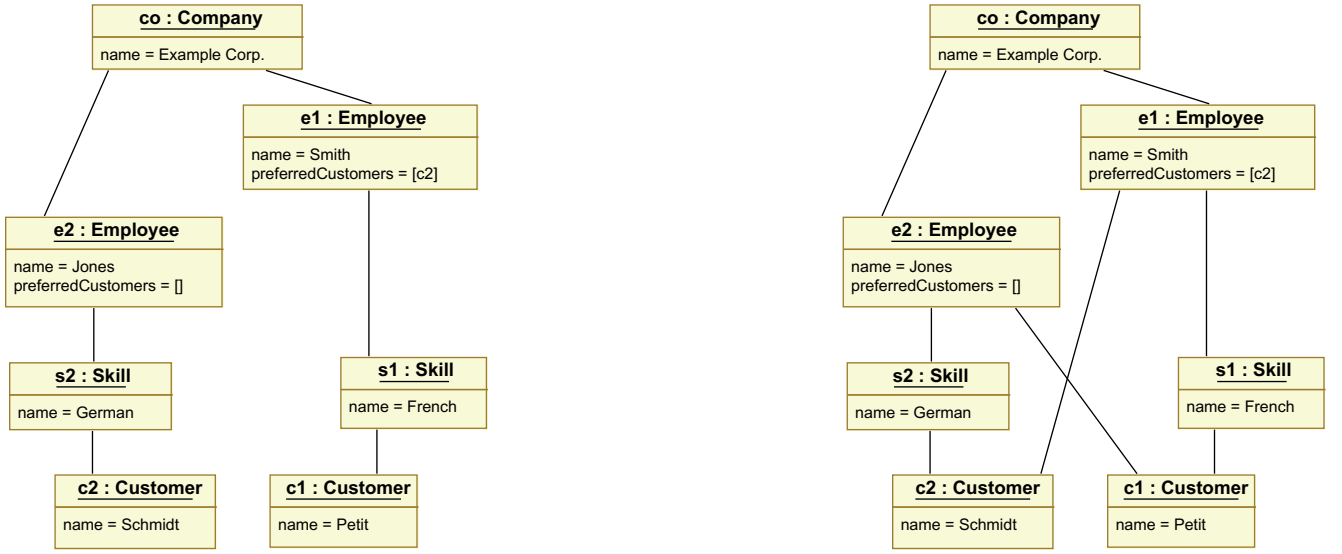
The tool OCLexec we implemented our approach in generates Java method bodies. It inserts code that enforces the postconditions of the operation and all class invariants.[2] OCLexec serializes an intermediate representation (see Section 4) of the operation contract to a file that the generated method body can access as a resource. The method body only reads the serialized file and calls a library routine responsible for simulating the operation. Note that inserting code in method bodies should not interfere with other code that may have been generated for the model. Thus, the developer can use her favorite tool for the overall code generation and then use our tool only for selected method bodies.

Figure 3(a) depicts a very simple system state in which the operation `assignNewCustomers` can be called. The company employs two staff members whose names are Smith and Jones. There are two skills: French and German language skills. Smith speaks French while Jones speaks German. There are two customers, called Petit and Schmidt, who ask for French and German language skills, respectively, from the employee that is assigned to them. Moreover, employee Smith prefers to work for customer Schmidt. Figure 3(b) shows a possible outcome of calling the generated method in this system state. Employee Smith is assigned to customer Schmidt and employee Jones is assigned to customer Petit. Unfortunately, neither customers' request for language skills is met. However, the assignment is stable, since employee Smith is now assigned to his preferred customer Schmidt and therefore not interested in changing the assignment.

Depending on the needs of the company, this result of the operation call may not be sufficient. It may well be that the customers' demands for skills are deemed more important than the preferences of the employees. If this is the case, simulation would have revealed an important flaw of the specification. Note that this kind of unforeseen behavior cannot be discovered if the constraints are only tested on system states that the specifier has designed to be correct or incorrect.

If the operation is not performance-critical and sufficiently efficient code can be generated for it, animation may allow to skip or postpone its implementation. Such an opportunity saves implementation effort and helps avoid coding errors. Moreover, a larger part of the development can be carried out on a higher and platform independent level of a abstraction. In this sense, animation of operation contracts can be regarded as a contribution to Model-Driven Development.

[2] If no valid new system state exists, an exception is thrown. Since we restrict all integer values and numbers of class instances to 32-bit numbers, the number of system states is finite and thus the existence of a valid new state is decidable. However, due to the large number of possible states, the code will appear to be non-terminating for difficult constraints.



(a) State before simulation

(b) State after simulation

Figure 3. Effect of simulating a call to the operation `assignNewCustomers` on a system state

3. Preliminary Analysis: Reasoning about New Class Instances

As a first step of simulation, we determine for which classes new instances may need to be generated. It is beneficial to restrict this set of classes as much as possible. Knowing that no instances need to be created for a certain class allows to reduce the search space that has to be explored. Such an observation also concerns the handling of class invariants. Class invariants have to always hold for all objects of a certain class. The following OCL invariant definition stating that the `name` attribute of an object of class `Skill` may never be empty could be added to the specification presented in Section 2:

```
context Skill inv: name <> ""
```

If no instances are created for a class that an invariant belongs to and the invariant references no attributes that can be modified by the operation, then we can conclude that the invariant can never be violated in the post-state if it was satisfied in the pre-state. Hence, such invariants do not need to be considered when searching for animation results, which simplifies the computation.

The creation of new instances can be restricted by the operation contract. OCL provides the `oclIsNew` test for querying whether an object has been created by the operation call. By using this language feature, postconditions can express that certain objects must have already existed before the operation call. However, it is usually not possible to observe that this test has been applied to a set of object references that is sufficient to completely rule out the creation of new instances for a certain class. In this case we must take into account that it may be necessary to create new instances of the class to satisfy the operation contract. Our goal is to compute an approximation of the set of classes requiring new instances that is as good as possible.

For defining a suitable approximation, we observe that, in order to be relevant for simulation results, a reference to a new object needs to be (i) the value of an output parameter, (ii) a value of a modified attribute of an object existing before the operation call, or (iii) a value of an attribute of a freshly created object.

We can approximate the set of classes for which new instances need to be created by analyzing for which types of objects these cases can occur. In order to limit the impact of case (iii), we demand that attributes of a freshly created object are assigned the undefined value representing an absence of value when this is possible without compromising satisfiability of the operation contract. This is the case when the multiplicity lower bound of the attribute is zero and the attribute is not referenced by any constraint considered during simulation.

Let T denote the set of classes for which new instances may be created and C be the set of constraints considered for animation. The observations above yield that the following conditions on T and C are sufficient for ensuring correct simulation.

1. Every postcondition belongs to C.
2. If a modifies only clause lists an attribute a, then every invariant referencing a belongs to C.
3. Every invariant of a class in T belongs to C.
4. If a class t is the type of an out-parameter of the operation or a subtype of the parameter's type, then $t \in T$. This corresponds to case (i) above.
5. If a modifies only clause lists an attribute a, then every class t which is the type of a or a subtype of the type of a belongs to T. This corresponds to case (ii) above.
6. For every attribute a of a class in T, if a is referenced by a constraint in C in the post-state or a does not have a multiplicity lower bound of zero, then every class t which is the type of a or a subtype of the type of a belongs to T. This condition corresponds to case (iii) above.

The smallest sets T and C that satisfy these conditions can easily be found by a closure computation. We initialize T and C to empty sets and augment the sets according to the conditions until a fixed point is reached.

When performing preliminary constraint analysis on the operation contract in Figure 2, the set C of constraints to consider consists of the two postconditions, since the specification does not de-

fine any invariants. The set T is set to $\{\texttt{Employee}, \texttt{Customer}\}$ according to Condition 5. The class `Company` is added to T by Condition 6 because the association end `company` of class `Employee` does not have a multiplicity lower bound of zero, and the procedure terminates with

$$T = \{\texttt{Employee}, \texttt{Customer}, \texttt{Company}\} .$$

Thus, no new instances of class `Skill` need to be created in the post-state. Hence, even if the example invariant given at the beginning of this section is included in the specification, it would not need to be considered during simulation. However, if the invariant was defined for the class `Company` instead of `Skill`, it would be added to the set C by Condition 3.

4. A Simplified Intermediate Representation for Constraints

UML/OCL constraints have a complex structure. They deal with a rich set of types including classes and several kinds of collections. In order to facilitate analysis, we translate constraints to a simpler intermediate representation. The translation must preserve the semantics of the constraints as far as needed for simulation.

It turns out that UML/OCL constraints can in large part be represented by integer expressions; solutions of these encoded constraints are assignments to their free variables and uninterpreted function symbols. The underlying state of an OCL expression is represented by uninterpreted function symbols for attributes. Thus, solving OCL constraints boils down to finding solutions for integer constraints. The latter reduces to two problems: first, we have to find bounds for the uninterpreted functions, and second, we have to provide an algorithm for effectively constructing solutions once these bounds are fixed. The advantage of this intermediate language is a substantial simplification of the implementation. In particular, we no longer need to separately consider bounds on numbers of class instances or on collection sizes. Moreover, the reuse of standard backend integer solvers is facilitated.

4.1 Arithmetic Expressions with Bounded Quantifiers and Uninterpreted Functions

The following language, which we refer to as the language of arithmetic expressions with bounded quantifiers and uninterpreted functions, provides the building blocks for our intermediate constraint representation. This language can be defined as follows:

1. Function symbols represent uninterpreted functions mapping $\mathbb{Z}^n$ to $\mathbb{Z}$.
2. Additional function symbols represent arithmetic operations available in OCL such as addition, subtraction and multiplication.
3. Variables are terms and assume values in $\mathbb{Z}$.
4. A function symbol applied to terms is a term.
5. Binary predicates $=$, $<$, $\leq$, $>$ and $\geq$ applied to terms are formulas.
6. Formulas can be connected using the usual boolean operations.
7. For a formula p and terms t_1 and t_2, if p then t_1 else t_2 is a term.
8. If p is a formula and t_1, t_2 are terms, then $\forall t_1 \leq x \leq t_2 \,.\, p$ is a formula. Here t_1 is the lower bound and t_2 is the upper bound of the quantifier. Similarly, $\exists t_1 \leq x \leq t_2 \,.\, p$ is a formula.

A UML system state can be represented by attribute functions that are assigned to function symbols. Using bounded quantifiers ensures that the evaluation of a closed formula in a given state is executable. Thus, this class of arithmetic formulas preserves the desirable property of OCL that constraints can be evaluated at runtime. We will make use of quantifier bounds for analyzing formulas and computing simulation results.

4.2 Translating OCL Expressions to Arithmetic Expressions

We outline how OCL expressions can be translated to nested tuples of the arithmetic expressions defined above. Representing OCL expressions of type `Integer` by arithmetic terms is straightforward; we cope with undefinedness in OCL separately (see below). Similarly, OCL expressions of type Boolean can be compiled directly to formulas.

Expressions whose type is a class are mapped to a pair (t_1, t_2) of arithmetic terms, where t_1 describes the dynamic type of the object which is the expression value. For this purpose, we assign an integer to every non-abstract class in the model. Note that due to subtyping t_1 can become quite complex. The second term t_2 gives an identifier of the object.

An expression of a collection type is mapped to the comprehension

$$\langle t_1 \leq x \leq t_2 \,|\, p(x) \bullet E(x) \rangle \,, \tag{1}$$

where x is a variable, t_1 and t_2 are terms, $p(x)$ is a formula and $E(x)$ is itself an encoding of an OCL expression whose type is the element type of the collection. The terms t_1 and t_2 are the lower and upper bound of the variable x. As indicated by the notation, x may occur in $p(x)$ and $E(x)$. For every integer i between t_1 and t_2, the value described by $E(i)$ belongs to the collection iff $p(i)$ is true. Here we denote by $p(i)$ and $E(i)$ the formula $p(x)$ and the encoding $E(x)$, respectively, with i substituted for x.

Collection operations are translated by manipulating the tuple that represents the collection expression. For example, a `select` operation on a collection represented by a comprehension of the form (1) is translated by conjoining the body of the `select` construct with the predicate $p(x)$. A `collect` operation can be translated by substituting $E(x)$ by the body of the `collect` construct.

Every OCL type includes an undefined value.[3] Therefore, we add to every translation of an OCL expression e a formula d_e that evaluates to true iff the expression does not evaluate to the undefined value.

There are a limited number of OCL language features like recursive operations that we do not support due to the effort required to encode them using this kind of framework.

We show how the translation is derived for the the first postcondition `allCustomersAssigned` of the operation contract in Figure 2. We adhere to the OCL convention that the variable holding the object the operation is called on is named `self`. We assign the integer 0 to the class `Company` and introduce the function symbol $f_{\texttt{self}}$ in order to translate the `self` variable of type `Company` to the term pair $(0, f_{\texttt{self}})$. Note that `self` is the implicit source of the attribute call `employees`**`@pre`** in this postcondition. We assign the integer 1 to the class `Employee` and introduce the 0-1-valued function symbol $f_{\texttt{employees}}$ with arity four for representing a characteristic function that indicates whether an `Employee` object is associated with a `Company` object in the pre-state. The function symbol $f_{\texttt{Employee}}$ represents the number of `Employee` instances. Thus, the expression `employees`**`@pre`** of type `Set` can be translated to the comprehension

$$\big\langle 0 \leq x \leq f_{\texttt{Employee}} - 1 \,|\, p(x) \bullet (1, x) \big\rangle \quad \text{with}$$
$$p(x) := f_{\texttt{employees}}\,(0, f_{\texttt{self}}, 1, x) = 1.$$

[3] More recent versions of the OCL standard call for two undefined values, `null` and `invalid`. For the sake of simplicity, we only use one undefined value here.

We use the 0-1-valued function symbols $f_{def(\texttt{customer})}$ and $f'_{def(\texttt{customer})}$ for indicating whether the values of the attribute `customer` are undefined in the pre- and post-state, respectively. Thus, the translation of the expression

```
employees@pre
    ->select(customer@pre.oclIsUndefined()
        and not customer.oclIsUndefined())
```

becomes $\langle 0 \leq x \leq f_{\texttt{Employee}} - 1 \,|\, q(x) \bullet (1, x) \rangle$

with $q(x) := f_{\texttt{employees}}\,(0, f_{\texttt{self}}, 1, x) = 1$
$\wedge f_{def(\texttt{customer})}\,(1, x) = 0$
$\wedge \neg f'_{def(\texttt{customer})}\,(1, x) = 0.$

We assign the integer 2 to the class `Customer` and introduce the function symbol $f'_{\texttt{customer}}$ with arity two for representing a function which maps `Employee` instances to their values of the attribute `customer` in the post-state. As a result, applying the `collect` construct with body expression `customer` to this source yields the translation

$$\langle 0 \leq x \leq f_{\texttt{Employee}} - 1 \,|\, q(x) \bullet f'_{\texttt{customer}}\,(1, x) \rangle$$

with the same formula $q(x)$ as in the previous comprehension.

Using the function symbol $f_{\texttt{newCustomers}}$ as characteristic function for the set-valued parameter `newCustomers` and applying the same translation scheme as above for this expression gives us the comprehension

$$\langle 0 \leq x \leq f_{\texttt{Customer}} - 1 \,|\, r(x) \bullet (2, x) \rangle \quad \text{with}$$
$$r(x) := f_{\texttt{newCustomers}}\,(2, x) = 1.$$

Based on these translations, we obtain the following formula that expresses that all elements of `newCustomers` belong to the collection on the left side of the equality:

$$\begin{aligned} &\forall\, 0 \leq x \leq f_{\texttt{Customer}} - 1\,. \\ &\quad f_{\texttt{newCustomers}}\,(2, x) = 1 \\ &\quad\Longrightarrow\ \exists\, 0 \leq y \leq f_{\texttt{Employee}} - 1\,. \\ &\qquad\qquad q(y) \wedge x = f'_{\texttt{customer}}\,(1, y)\,. \end{aligned}$$

In order to translate the entire equality, we only need to add an analogous formula for the containment in the other direction.

5. Solving Constraints in the Intermediate Representation

For simulating an operation, we translate the postconditions of the operation and all relevant invariants[4] to arithmetic constraints as outlined in the previous section. The conjunction of the resulting formulas expresses the condition that must be satisfied when the operation returns. In the next step we attempt to find a *model* for this formula, i.e., an assignment of specific functions to the function symbols for which the formula evaluates to true. Since our translation preserves the semantics of the operation contract, a model found yields a new system state conforming to the contract. The new state can be directly constructed from such a model. We only search for models that comply with the respective pre-state.

To find a model of the formula, we encode arithmetic formulas into Boolean constraints which are solved using an off-the-shelf satisfiability (SAT) solver. A model of the original formula can be obtained by straightforward decoding of a solution to the Boolean problem. This approach is called *bit-blasting* since a large number of Boolean variables may be necessary to encode an integer value or an arithmetic operation. Bit-blasting is a widely used approach for analyzing systems that employ finite-precision bit-vector arithmetic; see [7] for a recent overview. The advantage of this approach is that highly optimized SAT solvers are available for solving the resulting Boolean constraints. Although the Boolean satisfiability problem is NP-hard, powerful heuristics enable these solvers to scale well for a wide range of constraints arising in different application domains.

To find a model of an arithmetic formula with bounded quantifiers, we proceed as follows. First, the formula is simplified in order to remove redundant subexpressions.[5] Second, we construct a Boolean circuit that computes the validity of the formula. In order to construct the circuit, we may need to restrict the ranges of certain values. Third, the Boolean circuit is converted to conjunctive normal form (CNF). Fourth, the resulting CNF is solved by an off-the-shelf satisfiability (SAT) solver. A solution to the Boolean satisfiability problem yields a model for the original arithmetic formula. If no solution to the Boolean problem exists and we had to restrict any ranges in order to construct the circuit, we repeat the analysis with larger ranges.

In the sequel we describe this procedure in more detail and show how this approach is made suitable for simulation.

5.1 Encoding as a Boolean Circuit

Our encoding of arithmetic formulas as Boolean circuits does not differ significantly from the encodings employed by other SAT-based analysis tools [12, 35].

Encoding of Function Symbols Recall that function symbols represent uninterpreted functions mapping $\mathbb{Z}^n$ to $\mathbb{Z}$. We encode function symbols as vectors of Boolean variables. For every function value these vectors contain as subvector a bit-vector that is long enough to represent all values in the range of the function (we discuss the problem of fixing this range below in Section 5.2). Through an analysis of the formula we determine the set of possible arguments the function may be evaluated for during an evaluation of the formula. The length of the vector for a function symbol is the product of the number of possible arguments and the number of bits necessary for representing a function value.

Encoding of Terms We encode integer terms as vectors of Boolean circuits which represent the bits of the integer value. Arithmetic operations like addition and multiplication are dealt with by constructing a Boolean circuit for the operation, as would be done for computing the operation in hardware. In conformance with the UML/OCL standard, we do not allow arithmetic overflow. E.g., we encode the sum of two integer attributes mapped to Java `int`s as a vector of 33 bits, so all values that can result from the addition of two 32-bit integers can be represented. We translate function application to a multiplexer circuit that selects the bit-vector which corresponds to the value of the function argument. If a term contains free variables, we perform the encoding for every possible variable assignment. This results in a map that assigns a vector of Boolean circuits to every variable assignment.

Encoding of Formulas Boolean operations in the intermediate constraint representation can be mapped directly to corresponding gates in the generated Boolean circuit. For quantifiers, we encode the body of the quantified formula together with a guard checking the quantifier bounds for all possible assignments to the quantified variable. The resulting Boolean circuits are fed into the respective gate ($\wedge$ or $\vee$).[6]

[4] These are the constraints in the set C defined in Section 3.

[5] These simplifications apply mainly to subexpressions that identify undefined values and are not further described here.

[6] Of course this is not necessary for quantifiers that can be eliminated by skolemization.

Translating arithmetic formulas this way yields a Boolean circuit whose inputs are Boolean variables encoding a post-state and any output parameters of the operation.

5.2 Choosing Suitable Ranges for Function Symbols

Recall that in the Boolean encoding of the intermediate constraint representation outlined above, a subexpression with free variables is encoded separately for all values the variables can assume during an evaluation of the formula. It is clearly not feasible to always perform the encoding for all values in the largest possible quantifier range, e.g., all 32-bit integers.

Existing analysis tools for UML/OCL operation contracts like UML2Alloy [1] and UMLtoCSP [9] depend on bounds provided by the user for restricting quantifier ranges. The results of the analysis only make a statement about system states that comply with the provided bounds. However, for the purpose of simulation it is highly desirable to use a form of analysis that is complete in the sense that valid simulation results are obtained if they exist. We aim to relieve the user from the burden of providing adequate bounds. In particular, the necessity of specifying bounds is a considerable obstacle to the integration of animated operations with other code, since it requires a modification of the operation interface.

Some SMT (Satisfiability Modulo Theories) solvers, such as Z3 [13], apply heuristics to instantiate the quantifiers in their input sufficiently for proving that the constraint is unsatisfiable. However, these quantifier instantiation strategies are only complete for limited theory fragments. In general, that quantifier instantiation cannot derive unsatisfiability does not imply that a correct model of the formula can be obtained. So-called model finders like Paradox [12] that search for models of first-order formulas with a SAT solver proceed by generating SAT problems that do not cover all potential models. If no model is found, they generate a larger SAT problem to cover more models, and so on.

We propose a similar iterative approach that is based on restricting the ranges of certain function symbols occurring in the arithmetic formula. We restrict the ranges of those function symbols that occur in quantifier bounds provided by the intermediate representation. Through interval arithmetic, we can then derive a restricted range for each lower and upper quantifier bound. Thus, we can obtain a sufficient translation of a quantified formula by instantiating the quantified variable only for the restricted set of values that can be between the quantifier bounds. If the function symbol ranges are chosen to be small enough, this set of values the quantified variable can assume is manageable. If no model is found for the first choice of restricted function symbol ranges, a more expensive attempt with larger ranges is made, and so on.

We restrict the ranges of function symbols occurring in function arguments as we do for function symbols occurring in quantifier bounds. As a result, we can derive through interval arithmetic sufficiently bounded ranges for all terms that are function arguments. This allows us to encode function symbols as vectors of Boolean variables that are of manageable size.

Restricting the range of a function symbol results in an under-approximation of the original satisfiability problem, i.e., certain models are excluded, whereas every solution to the under-approximation is a valid model for the formula. Note that simply restricting the quantifier ranges considered during the translation while leaving function symbol ranges unchanged does not necessarily yield an under-approximation, and thus may give rise to solutions that are not valid models of the formula.

Since in a bit-blasting approach a fixed number of bits has to be allocated for every function value in order to obtain a Boolean encoding of the formula, the range of every function symbol has to be bounded. We assume that UML integers are mapped to bounded Java types by code generation. This allows us to restrict function symbols that do not occur in quantifier bounds or function arguments to ranges which are certainly sufficient. Consider as an example a function symbol corresponding to an attribute that is mapped to a Java field of type `int`. Its values may be restricted to 32-bit numbers, which is sufficient to represent all values of the Java `int` type.

As a result of this approach to bounding function values, we search for models using different bounds for the integer values in the system state. These integer values can be values of integer attributes, numbers of instances of a class or collection sizes. The bounds for these values are chosen depending on the contexts in which the values are used in the constraints. In the formulas resulting from the example translation in Section 4.2, the only function symbols that appear in quantifier bounds are $f_{\texttt{Customer}}$ and $f_{\texttt{Employee}}$. These function symbols represent numbers of class instances in the pre-state. Thus, these values are fixed, and an optimal instantiation of the concerned quantifiers is performed during the translation in this case. Assume that we were dealing with the translation of a different OCL constraint, such that the function symbols would instead represent numbers of instances in the post-state and therefore not be fixed. Then we would introduce restrictive bounds for these function symbols which, if necessary, are increased in future iterations in order to find a model. Another case are variable function symbols that do not appear in quantifier bounds. Suppose the operation contract in the example included the additional postcondition

```
totalCustomers = totalCustomers@pre
                              + newCustomers->size() .
```

Then, if the function symbol for the integer attribute `totalCustomers` used in this example does not occur in quantifier bounds, we consider all possible 32-bit values of this attribute already in the first attempt to find a valid post-state. Thus, even states with very large values for this attribute are not necessarily problematic for simulation.

5.3 Efficient Translation of Formulas to Boolean Circuits

In our approach, the actual constraint solving is performed by the SAT solver that receives the CNF. The preceding computation that generates the Boolean circuit from the arithmetic formula and converts the circuit to a CNF is deterministic and has a complexity that is polynomial in the size of the circuit. These facts suggest that the SAT solving is the bottleneck regarding runtime, whereas the preprocessing steps are uncritical for performance. Nevertheless, in our experience the cost of generating the input to the SAT solver is for many simulation problems far more expensive than the execution of the SAT solver itself.[7] We observed that many SAT instances arising during animation are solved in a fraction of a second. The main factor that determines the size of the Boolean circuit and the CNF, and thus the preprocessing time, are the quantifiers that are present in the input formula and the ranges for which they are instantiated. Nested quantifiers are particularly expensive.

In order to reduce the time used for preprocessing, we have implemented an improved algorithm for translating formulas of our intermediate language to Boolean circuits. Figure 4(a) shows a typical approach to perform an encoding like the one described in Section 5.1. The assignment `env` to the free variables of the formula is a parameter to the translation. This assignment can then be passed on to recursive calls of the procedure for translating subexpressions. The resulting subexpression translations can be used for obtaining a translation of the entire formula, e.g., by feeding them into an addition circuit constructed by the function `make_PLUS` as

[7] See Section 6 in this paper and [10, 35] for measurements showing that preprocessing consumed more time than SAT solving.

```
procedure translate(expr, env):
  case expr of PLUS(in1, in2):
    return make_PLUS(translate(in1, env),
                     translate(in2, env))

  case expr of FORALL(x, body):
     inputs := ∅;
     for i in [lowerBound(x)..upperBound(x)] do
       inputs :=
         inputs ∪ translate(body, env[x:=i])
     end for;
     return make_AND(inputs)

  case expr of ...
end procedure
```

(a) Top-down

```
procedure translate(expr)
  case expr of PLUS(in1, in2):
    translations1 := translate(in1);
    translations2 := translate(in2);
    for env in Assignments(FreeVars(expr)) do
      translations[env]
         := make_PLUS(translations1[env],
                      translations2[env])
    end for;
    return translations

  case expr of FORALL(x, body):
    body_translations := translate(body);
    for env in Assignments(FreeVars(expr)) do
      inputs := ∅;
      for i in [lowerBound(x)..upperBound(x)] do
        inputs :=
          inputs ∪ body_translations[env[x:=i]]
      end for;
      translations[env] := make_AND(inputs)
    end for

  case expr of ...
end procedure
```

(b) Bottom-up

Figure 4. Approaches to Generating Circuits from Formulas with Quantifiers (pseudo-code)

shown in Figure 4(a). For translating a quantified formula, a loop iterates over the values for which the quantifier is instantiated. For every value, the body of the quantified formula is translated with the quantified variable set to this value.[8] The translations of the body are then aggregated according to the type of the quantifier. This approach to formula translation is straightforward to implement. It is also suggested by some semantics definitions that define the semantics of quantifiers by constructs that resemble loops. This is also the case for the OCL standard [30].

However, it turns out the straightforward approach depicted in Figure 4(a) is not optimal concerning efficiency. Note that it causes a separate translation of every subexpression in the scope of a quantifier for every value the quantified variable can assume — even for subexpressions in which the quantified variable does not occur. For example, the subexpression

```
c.requestedSkills@pre
    ->intersection(c.employee.skills@pre)
```

[8] Here we assume that this body already contains the guard checking the quantifier bounds.

in the specification of Figure 2 would be translated for every value that the variable `e` of the enclosing `forAll` construct can assume, although this subexpression does not depend on this variable. This clearly is a waste of resources. It would be much more efficient to perform translations of subexpressions depending on their free variables.

This observation leads to the algorithm sketched in Figure 4(b). We call this approach bottom-up in contrast to the top-down method in Figure 4(a). The bottom-up algorithm first translates subexpressions for all possible assignments to their free variables. The resulting translations are stored in a map data structure that supports lookups based on a variable assignment. When translating an application of an operation like addition, the translations of the subexpressions are retrieved for every assignment to the free variables of the entire formula. When performing these map lookups, we discard any values for variables that are not free variables in the respective subexpression. The retrieved subexpression translations are then used for computing corresponding translations of the entire formula. For translating a quantified formula, the translations of the body are aggregated according to the type of the quantifier.

The bottom-up approach has the advantage that the number of times a subexpression is translated only depends on the values that its free variables can assume. The translation procedure visits every subexpression only once. All translations of a subexpression are performed in an efficient loop structure. This promotes optimizations like the elimination of loop invariant computations and caching of memory.

In the implementation of OCLexec, we use an adapted version of the kodkod solver for constructing the circuit as a *Compact Boolean Circuit* [35], a compressed representation of a Boolean circuit. Kodkod includes effective algorithms for constructing and compressing Compact Boolean Circuits. We do not make use of higher-level features of kodkod such as symmetry breaking or encoding of relations.

6. Experimental Results

We evaluated the efficiency of OCLexec by simulating the operation contract of Figure 2 in system states of increasing size. Specifically, these are states with a number n of `Employee` and `Customer` objects, respectively, and $\lfloor \log_2 n \rfloor$ `Skill` objects. Employees and customers are associated independently to every skill with probability $1/2$. This achieves that every subset of skills is quite likely to be associated with at least one employee or customer, which favors conflicts between the different actors. Similarly, an employee lists every customer as preferred with probability $1/2$. The results

Customers	Employees	Skills	Total Runtime	Runtime SAT
5	5	2	0.4 sec	3 msec
10	10	3	0.5 sec	7 msec
20	20	4	0.8 sec	160 msec
30	30	4	2.2 sec	1200 msec
40	40	5	23 sec	21 sec
50	50	5	78 sec	75 sec

Table 1. Experimental results

of the evaluation are shown in Table 1. We give the total execution times of animating the operation as well as the times consumed by the SAT solver alone. The measurements were performed on a machine with 2 GB RAM and a dual-core 2.4 GHz P8600 mobile CPU. The SAT solver used was MiniSat [16], a well-known state-of-the-art SAT solver.

For states with up to 20 employees and customers, the time of pre- and postprocessing the constraints dominates the runtime.

States of this size already include many interesting application scenarios. We note that for these states, simulation is efficient enough for certain applications like prototyping. However, for larger states the time consumed by SAT solving increases quickly, and animation becomes infeasible. These runtimes may seem disappointing, considering that a polynomial-time algorithm exists for the stable marriage problem. Note that the number of generated Boolean constraints grows faster than linearly in the number of employees and customers, due to e.g., the nested quantifers in the specification. Also recall that the we are processing a high-level specification in a relatively general-purpose language.

7. Related Work

There has been constant interest in animation as a research problem. Work on animation has focused particularly on the specification languages Z [15, 21, 36] and B [3, 27, 32]. The benefit of using intermediate languages for implementing animation tools has been recognized, and led to the intermediate languages μZ [21] for animating Z and CLPS-B [3] for B. These intermediate languages do not directly address the problem of quantifier bounding.

Pioneering work on animating UML/OCL can be found in [20, 31]. In comparison to our previous work [26], we introduced support for modifies clauses and improved the translation from the intermediate constraint representation to Boolean circuits. Animators for operation contracts have also been implemented, for example, for the specification language JML [4, 11, 25]. These animators do not use SAT solving like OCLexec does, but rely on other constraint solving techniques. They are automatic in the sense that they do not explicitly require the user to provide additional information such as bounds. However, they cannot handle certain constraints. Specifically, the JML-TT animator [4] lacks support for quantifiers, which severely restricts the class of specifications it can process. The jmle animator [11, 25] works by generating a prototype implementation in Java, as does OCLexec, and throws an exception at runtime for certain constraints it cannot handle. In contrast, an operation implementation generated by OCLexec always terminates successfully if valid operation results exist.

The power available through SAT solvers as constraint solving engines has long been recognized. Alloy [22], NP-SPEC [10], answer set programming systems (e.g., smodels [29]) and SAT-based CSP solvers (see e.g., [34]) are tools that process constraint languages using SAT solvers or similar search techniques. These languages avoid constructs that are difficult to encode in Boolean constraints, like multisets and nested collections which are available in OCL, and usually require narrow bounds on integer values that allow for explicit enumeration of the considered integers. As a result, these languages offer an attractive trade-off between expressiveness and efficiency. This comes at a price: since these languages are not tightly integrated into a large-scale language like UML or Java, developers need to write tedious glue code to interface with the constraint solver. We view OCLexec as complementary to these tools; after simulating an operation with OCLexec, a developer could seek a more efficient execution of the operation using such a SAT-based tool. The Alloy annotation language [23] could be regarded as an approach to close the gap between Alloy as a constraint language and Java as a large-scale language. Another closely related family of tools that use SAT solvers are SAT-based model finders [2, 12, 28]. They originally aim at analyzing mathematical theorems rather than solving constraints that arise in software development.

An alternative to the conventional type of SAT solver used in our simulation approach are Satisfiability Modulo Theories (SMT) solvers based on *theory combination* like Z3 [13]. Theory combination is a technique for integrating separate solvers for subtheories such as the theory of linear arithmetic or the theory of uninterpreted functions. This avoids encoding these subtheories using large Boolean formulas. We do not claim here that one of these constraint solving approaches is in general superior to the other, but observe that in either case efficiency depends on a carefully tuned implementation of the solver. An advantage of theory combination is that built-in solvers for linear arithmetic can handle real numbers with arbitrary precision and solve quantifier-free linear constraints without requiring any bounds.

UML2Alloy [1] and UMLtoCSP [9] are tools for analyzing UML/OCL specifications. UML2Alloy performs a translation to the Alloy language, while UMLtoCSP uses the Eclipse constraint programming system. These tools aim to verify certain properties of specifications, which basically amounts to solving the OCL constraints. Both require the user to specify bounds to restrict the scope of analysis. Moreover, they do not support modifies clauses and do not provide an interface for supplying the system state an operation is called in. A workaround could be to encode the input data as additional OCL constraints, but this is not straightforward and likely to have a negative impact on efficiency. USE [18] is an integrated environment for OCL that allows a form of animation. However, USE requires the user to explicitly construct the system states that are to be considered, and thus is not fully automatic. HOL-OCL [5, 6] is an embedding of OCL into an expressive language processed by an interactive theorem prover and also relies on substantial guidance from the user for most kinds of analysis. Another tool along these lines is the SQL query explorer Qex [37] that automatically constructs test cases for databases and is based on a powerful SMT solver.

Our bottom-up approach to translating quantifiers described in Section 5.3 is similar to a technique used by a past version of the Alloy tool [33] which augments the basic top-down approach with a cache of the generated Boolean subcircuits in order to prevent unnecessary quantifier instantiations. This technique can potentially save more quantifier instantiations than ours since it also takes identities obtained by constant folding into account. In contrast, we avoid the overhead of cache misses by performing all translations of a subexpression at once. This also allows us to process subexpressions in a predictable order, which facilitates optimizations. Our observation that a bottom-up translation can be more efficient than a straightforward top-down approach has an analogue in the area of XPath query evaluation [19].

8. Conclusions and Future Work

We presented an improved approach to simulating operation contracts. We implemented the approach in the simulation tool OCLexec that generates Java implementations from UML/OCL operation contracts.

Operation contracts are translated to arithmetic expressions with bounded quantifiers as intermediate constraint representation. Although OCL has a complex semantics, most of the UML/OCL language features can be mapped conveniently to this simple intermediate language. Final constraint solving is performed by encoding the constraints as a Boolean formula and calling an off-the-shelf SAT solver.

The intermediate constraint representation facilitates the automatic selection of bounds which are necessary for obtaining a Boolean encoding. As a result, the simulation is fully automatic, and no effort is required to link simulated operations with other code. Another useful characteristic of the simulation is that it is complete in the sense that valid simulation results are obtained if they exist.

We note that for many simulation problems the Boolean encoding phase is the performance bottleneck. As a consequence, we adopt an improved translation scheme for generating the Boolean constraint representation. Another beneficial optimization is a pre-

liminary constraint analysis that can allow to disable object creation and discard invariants for certain classes. Experimental results demonstrate that our approach can efficiently handle simulation instances of considerable size.

As future work we plan to add support for optimization according to an objective function when simulating. This would allow more operations to be specified and simulated. Also, we are considering assisting the user in providing modifies only clauses, which we find tedious to write. Finally, we would like to integrate our simulation method into a comprehensive code generation system.

References

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Proc. 10th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *Lect. Notes Comp. Sci.*, pages 436–450. Springer, 2007.

[2] J. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lect. Notes Comp. Sci.*, pages 131–146. Springer, 2010.

[3] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B — a constraint solver to animate a B specification. *Int. J. Softw. Tools Tech. Trans.*, 6(2): 143–157, 2004.

[4] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. In *Proc. 13th Int. Conf. Formal Methods 2005 (FM'05)*, volume 3582 of *Lect. Notes Comp. Sci.*, pages 75–90. Springer, 2005.

[5] A. D. Brucker and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Inf.*, 46(4):255–284, 2009.

[6] A. D. Brucker, J. Doser, and B. Wolff. An MDA framework supporting OCL. *ECEASST*, 5, 2006.

[7] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. An abstraction-based decision procedure for bit-vector arithmetic. *STTT*, 11(2):95–104, 2009.

[8] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[9] J. Cabot, R. Clarisó, and D. Riera. Verifying UML/OCL operation contracts. In M. Leuschel and H. Wehrheim, editors, *IFM*, volume 5423 of *Lect. Notes Comp. Sci.*, pages 40–55. Springer, 2009.

[10] M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artif. Intell.*, 162(1-2):89–120, 2005.

[11] N. Cataño and T. Wahls. Executing JML specifications of Java card applications: a case study. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 404–408. ACM, 2009.

[12] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proc. Wsh. Model Computation — Principles, Algorithms, Applications*, Miami, Florida, 2003.

[13] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lect. Notes Comp. Sci.*, pages 337–340. Springer, 2008.

[14] A. Dick, P. Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In J. Nicholls, editor, *Proc. 4th Z Users Workshop*, Workshops in Computing, pages 71–85, Oxford, 1989. Springer.

[15] V. Doma and R. A. Nicholl. EZ: A system for automatic prototyping of Z specifications. In S. Prehn and W. J. Toetenel, editors, *Proc. 4th Int. Symp. VDM Europe (VDM'91)*, volume 551 of *Lect. Notes Comp. Sci.*, pages 189–203, 1991.

[16] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Sel. Rev. Papers 6th Int. Conf. Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lect. Notes Comp. Sci.*, pages 502–518. Springer, 2004.

[17] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[18] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comp. Prog.*, 69(1–3):27–34, 2007.

[19] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106. Morgan Kaufmann, 2002.

[20] J. Gray and S. Schach. Constraint animation using an object-oriented declarative language. In A. J. Turner, editor, *Proc. 38th ACM Southeast Reg. Conf.*, pages 1–10. ACM, 2000.

[21] W. Grieskamp. A computation model for Z based on concurrent constraint resolution. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *Proc. 1st Int. Conf. B and Z Users (ZB'00)*, volume 1878 of *Lect. Notes Comp. Sci.*, pages 414–432. Springer, 2000.

[22] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[23] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *OOPSLA*, pages 231–245, 2002.

[24] P. Kosiuczenko. Specification of invariability in OCL. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *Lect. Notes Comp. Sci.*, pages 676–691. Springer, 2006.

[25] B. Krause and T. Wahls. jmle: A tool for executing JML specifications via constraint programming. In L. Brim, B. R. Haverkort, M. Leucker, and J. van de Pol, editors, *Rev. Sel. Papers 5th Int. Wsh. Parallel and Distributed Methods for Verification (PDMC'06)*, volume 4346 of *Lect. Notes Comp. Sci.*, pages 293–296. Springer, 2006.

[26] M. P. Krieger and A. Knapp. Executing underspecified OCL operation contracts with a SAT solver. *ECEASST*, 15, 2008. Proceedings of the 8th International Workshop on OCL Concepts and Tools.

[27] M. Leuschel and M. Butler. ProB: An automated analysis toolset for the B method. *Int. J. Softw. Tools Tech. Trans.*, 10(2):185–203, 2008.

[28] W. McCune. MACE 2.0 reference manual and guide. *Comp. Res. Rep.*, 6, 2001. `http://arxiv.org/abs/cs.LO/0106042`.

[29] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.

[30] Object Management Group. Object constraint language specification, version 2.2. Specification, OMG, 2010. `http://www.omg.org/spec/OCL/2.2`.

[31] I. Oliver and S. Kent. Validation of object oriented models using animation. In *Proc. 25th Conf. EUROMICRO*, pages 2237–2242. IEEE Computer Society, 1999.

[32] T. Servat. BRAMA: A new graphic animation tool for B models. In J. Julliand and O. Kouchnarenko, editors, *Proc. 7th Int. Conf. B Users (B'07)*, volume 4355 of *Lect. Notes Comp. Sci.*, pages 274–276. Springer, 2007.

[33] I. Shlyakhter, M. Sridharan, R. Seater, and D. Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. In E. Giunchiglia and A. Tacchella, editors, *Sel. Rev. Papers 6th Int. Conf. Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lect. Notes Comp. Sci.* Springer, May 2004.

[34] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

[35] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *Proc. 13th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lect. Notes Comp. Sci.*, pages 632–647. Springer, 2007.

[36] M. Utting. Data structures for Z testing tools. In G. Schellhorn and W. Reif, editors, *Proc. 4th Wsh. Tools for System Design and Verification (FM-TOOLS'00)*. Technical Report 2000-07, Universität Ulm, 2000.

[37] M. Veanes, P. Grigorenko, P. de Halleux, and N. Tillmann. Symbolic query exploration. In K. Breitman and A. Cavalcanti, editors, *ICFEM*, volume 5885 of *Lect. Notes Comp. Sci.*, pages 49–68. Springer, 2009.