Executing Underspecified OCL Operation Contracts with a SAT Solver

Matthias P. Krieger and Alexander Knapp

Ludwig-Maximilians-Universität München
{kriegerm,knapp}@pst.ifi.lmu.de

Abstract Executing formal operation contracts is an important technique for requirements validation and rapid prototyping. Current approaches require additional guidance from the user or exhibit poor performance for underspecified contracts that describe the operation results non-constructively. We present an efficient and fully automatic approach to executing OCL operation contracts which uses a satisfiability (SAT) solver. The operation contract is translated to an arithmetic formula with bounded quantifiers and later to a satisfiability problem. Based on the system state in which the operation is called and the arguments to the operation, an off-the-shelf SAT solver computes a new state that satisfies the postconditions of the operation. An effort is made to keep the changes to the system state as small as possible. We present a tool for generating Java method bodies for operations specified with OCL. The efficiency of our method is confirmed by a comparison with existing approaches.

1 Introduction

The Object Constraint Language (OCL [1]) has established itself as a textual notation for describing behavior that cannot be expressed adequately through UML diagrams. OCL allows the modeler to specify class invariants and to define operation contracts through pre- and postconditions. Tool support is essential for OCL to gain widespread acceptance among modelers, but is at the moment basically restricted to constraint evaluation. The Dresden OCL Toolkit [2] for example provides runtime constraint checking, while USE [3] is an integrated OCL evaluation environment. OCL would be much more useful if OCL operation contracts could be executed directly and compiled to code. Executing an operation contract means determining a valid system state in which an invocation of the operation can return. Based on the system state in which the operation is called and the arguments to the operation, a new state is to be computed that satisfies the postconditions of the operation and all invariants. Specification execution can help the developer gain confidence in the specification and find errors early in the design process.

In this paper we present tool support for executing OCL operation contracts and generating code from them. The task of executing specifications, also called *animation* [4], is a long standing research problem (see Leuschel and Butler [5] for a recent overview). Several animation tools have been implemented (see e.g. [6,7,8,9,5]). Executing operation contracts is challenging since postconditions and invariants only state which conditions must hold in the new system state. The constraints often do not indicate how a valid system state can be found. In particular underspecified contracts often describe the operation results non-constructively. A contract is *underspecified* if it allows multiple behaviors for the same input.

Animators need constraint solving algorithms to handle underspecified contracts. The algorithms used by existing animators, although often sophisticated, scale poorly for many simple contracts. In order to allow for more efficient animation, we propose to use a so-called Satisfiability (SAT) solver as the underlying constraint solver. A SAT solver decides whether a Boolean formula is satisfiable, i.e. if there is an assignment to the variables in the formula for which the formula is true. By translating OCL constraints to Boolean formulas, SAT solvers can be used to determine valid system states for the purpose of animation. A large variety of SAT solvers have been implemented. It is our intention to make use of the intense research on satisfiability solving [10] for executing specifications. We have implemented our SAT-based animation approach in OCLexec[1], a tool that generates Java method bodies from OCL operation contracts.

1.1 Related Work

A lot of effort has been put into animation tools, in particular for the specification languages Z [11,12,6] and B [13,9,5]. Animators for the Java Modeling Language (JML) have been implemented too [7,8]. There has also been work on animating UML/OCL [14,15]. None of these tools uses SAT solvers.

There are many tools that translate languages with a richer semantics than Boolean formulas to a satisfiability problem. Several so-called model finders [16,17,18] analyze first-order formulas with a SAT solver. The formulas processed by these tools are similar to the arithmetic formulas we introduce in the sequel.

The Alloy Analyzer [19] processes a language that is similar to UML/OCL models. This tool constructs system states that meet constraints specified in a first-order language. The analysis is performed with the SAT-based model finder Kodkod [20], a software package that our tool uses as well. Alloy reads no input other than the Alloy language, no kind of code is generated. The user needs to give bounds to clarify what kind of output she expects. In particular, the integer values considered by Alloy have to be restricted to a small user-defined range.

The tool UML2Alloy [21] translates UML/OCL constructs to their counterparts in the Alloy language. The output is a text file that the user can open with Alloy in order to analyze the constraints. The user needs to specify the same kinds of bounds as for any other analysis with Alloy. UML2Alloy focuses on invariant constraints, support for translating UML operations to Alloy is quite restricted. Operations cannot have parameters or return a value, and the `@pre` construct of OCL is not taken into account by the translation.

1.2 Organization of the Paper

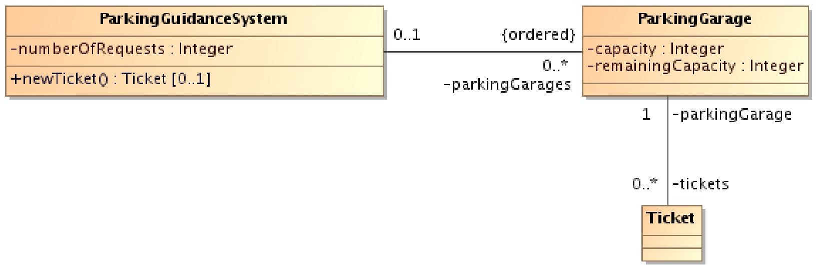
In Section 2 we present code generation from OCL contracts and discuss its benefits. Section 3 explains how the part of the system state that needs to be modified by anima-

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

tion is approximated. Section 4 introduces arithmetic formulas with bounded quantifiers as an intermediate representation of constraints that is amenable to analysis with a SAT solver. We outline how OCL constraints can be translated to arithmetic formulas. Section 5 describes our code generation tool OCLexec in more detail. In Section 6 we give experimental results for OCLexec and existing tools.

2 Code Generation from Operation Contracts

In this section we present code generation from operation contracts and discuss its benefits. Figure 1 shows a simplified OCL specification of a parking guidance system which will serve as example. The parking guidance system monitors the remaining capacity of a list of connected parking garages. There are tickets for entering and parking in a parking garage associated with the ticket. An operation `newTicket` of the parking guidance system creates and returns a `Ticket` associated to a `ParkingGarage` which is not full. If all monitored parking garages are full, `null` is returned. For statistical purposes the counter `numberOfRequests` is incremented each time the operation is called. This behavior of the operation is specified with the OCL contract in Fig. 1. Since the operation can choose any `ParkingGarage` that is not full, the contract is underspecified. The expression `result.oclIsUndefined()` is true when the operation returns `null`. The condition `result.oclIsNew()`

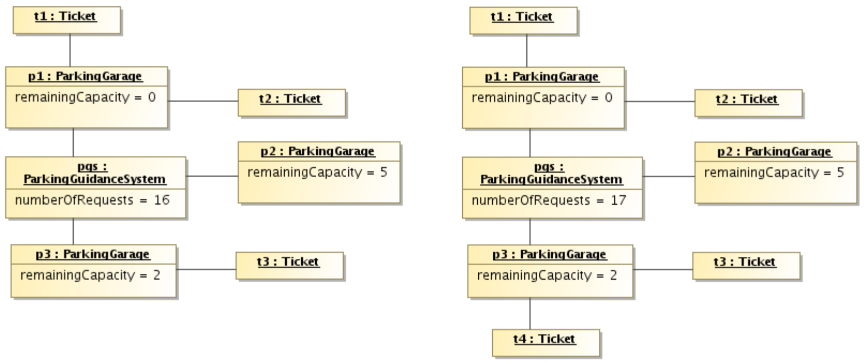


(a) Simplified UML model of a parking guidance system

```
context ParkingGuidanceSystem::newTicket(): Ticket
post: if not result.oclIsUndefined() then
        result.parkingGarage.remainingCapacity@pre > 0
          and result.oclIsNew()
          and not result.parkingGarage.oclIsNew()
      else
        parkingGarages@pre->forAll(remainingCapacity@pre = 0)
      endif
post: numberOfRequests = numberOfRequests@pre + 1
```

(b) Operation contract for returning a ticket for a parking garage that is not full

Figure 1. Example OCL specification



(a) State before animation (b) State after animation

Figure 2. Effect of animating a call to `newTicket` on a system state

ensures that a `Ticket` object returned by the operation is new and did not already exist before the operation was called. In order to prevent that a `Ticket` returned refers to a new `ParkingGarage` created by the operation, the condition `not result.parkingGarage.oclIsNew()` is added.

The tool we present in this paper generates Java method bodies. It inserts Java code that enforces the postconditions of the operation and all class invariants. If no valid new system state exists, an exception is thrown[2]. Objects can be created by the generated method body. However, objects are not explicitly destroyed. An object can be regarded as deleted if it is inaccessible from other parts of the object graph.

If the tool is run on the contract in Fig. 1, code will be generated that behaves as required. In particular, all postconditions listed in the contract always hold after the code is executed. The values of the properties `remainingCapacity` and `parkingGarages` are not changed by the generated code since these properties are only referenced by the contract at precondition time. The code generator observes that modifying values of these properties cannot help satisfy the postconditions. Only the association between the classes `ParkingGarage` and `Ticket` is modified by a call of this operation.

Figure 2(a) depicts a possible system state in which the operation `newTicket` can be called. The parking guidance system monitors three parking garages, only one of them is full. Figure 2(b) shows a possible outcome of animating a call of `newTicket` in this system state. Animation creates and returns `Ticket t4` that is linked to `ParkingGarage p3`. Since `ParkingGarage p2` is not full either, animation could have linked `Ticket t4` to `ParkingGarage p2` as well.

Code generation from operation contracts has several benefits. Execution of a specification can be a significant contribution to its validation. Execution scenarios that are

[2] 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.

generated automatically without additional guidance from the modeler can be surprising. It may become clear that additional constraints are necessary. For example, it may turn out that it is not desired that a `ParkingGarage` with very little remaining capacity is selected if there is a `ParkingGarage` with plenty of empty space available. Such unforeseen behavior can not be discovered if the constraints are only tested on system states that the modeler has designed to be correct or incorrect.

It may be possible that an improved version of our tool allows the implementation of operations for which sufficiently efficient code is generated to be skipped or postponed. This may well apply to a large part of the system since most of the execution time tends to be used for executing only a small fraction of the code. The operation `newTicket` performs an unsophisticated computation that is unlikely to have a significant impact on the overall performance of the whole system. Relatively efficient code can easily be generated from the contract of this operation. 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, code generation from operation contracts can be regarded as a contribution to Model-Driven Development.

In addition to these benefits, OCL contracts can be used for other purposes that are not directly related to our tool. For example, constraints can serve as test oracles. The contract in Fig. 1 can be used to test any implementation of the operation. It does not matter whether the implementation selects the first suitable `ParkingGarage` on the list or a different `ParkingGarage` which is not full — every correct implementation complies with the contract.

3 Preliminary Constraint Analysis

As a first step of animation, we need to determine which parts of the system state may change during animation. We seek to avoid changes to the system state as much as possible. This is important since the current OCL standard does not provide a mechanism for specifying which parts of the system state an operation may modify. Instead, the modeler has to list the parts of the system state that the operation should not change. For example, in order to prevent the operation `newTicket` specified in Fig. 1 from changing the attribute `remainingCapacity` of the class `ParkingGarage`, the further postcondition

```
parkingGarages@pre
    ->forAll(remainingCapacity = remainingCapacity@pre)
```

needs to be added. Systematically adding constraints like this one is usually infeasible except for very simple specifications. Since animation would be useless if it modifies parts of the system state that are completely unrelated to the animated operation, we only allow animation to perform changes that may be necessary for satisfying the operation contract.

Restricting changes of the system state is also important for the efficiency of animation. The more of the old system state can be reused, the less search is necessary to find a new system state satisfying the operation contract. Moreover, if certain class invariants only depend on constant parts of the system state, we do not need to consider these constraints during animation.

We only allow animation to modify parts of the object graph that are connected to the objects passed to the animated operation. The objects passed to the operation are the `self` instance and the objects contained in the arguments of the operation call and in the values of any static attributes. In fact, these are the only parts of the object graph that are accessible to a Java method body implementing the operation.[3] We denote the modifiable parts of the system state by giving a set P of properties that may be altered and a set C of classes for which new instances may be created. Recall that we do not give animation any opportunity to delete class instances except by making them inaccessible from other parts of the object graph. Hence, we do not give a set of classes for which instances may be deleted.

We can write an OCL class invariant as $\forall x \,.\, F(x)$, where x is the `self` variable and $F(x)$ is an OCL constraint depending on x. The variable x ranges over all instances of the class for which the invariant is specified. Note that a new system state can only violate the invariant if properties that are referenced in $F(x)$ were altered, or if new instances of the class were created. We define I to be the set of invariants that reference a property in P or are specified for a class in C.

For animating a query operation, which is not supposed to modify the system state, we set $P = \emptyset$ and $C = \emptyset$. For non-query operations the following conditions on P and C guarantee that the modifiable portions of the system state are sufficient for animation.[4]

1. If a property p is referenced at postcondition time by a postcondition of the operation to be animated, then $p \in P$.
2. If an invariant in I references a property p, then $p \in P$.
3. If a property $p \in P$ is an association end, then the opposite association end must be in P as well.
4. For every property $p \in P$, if a class c is the type of p or a subtype of the type of p, then $c \in C$.
5. If a class c is the type of an out-parameter of the operation or a subtype of the parameter's type, then $c \in C$.
6. For every property p of a class in C, if a class c is the type of p or a subtype of the type of p, then $c \in C$.

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

Example 1. When applied to the operation contract in Fig. 1, preliminary constraint analysis sets P to $\{$`parkingGarage`, `numberOfRequests`$\}$ according to Condition 1. The association end opposite to `parkingGarage`, `tickets`, is added to P

[3] Since Java does not provide convenient means for accessing all instances of a class, we do not support the `allInstances` operation of OCL.

[4] It is necessary that the constraints do not use the `allInstances` operation.

according to Condition 3. The set C is set to $\{\texttt{ParkingGarage}, \texttt{Ticket}\}$ by Condition 4. The class `ParkingGuidanceSystem` is added to C according to Condition 6, and the procedure terminates with

$$P = \{\texttt{parkingGarage}, \texttt{numberOfRequests}, \texttt{tickets}\} ,$$
$$C = \{\texttt{Ticket}, \texttt{ParkingGarage}, \texttt{ParkingGuidanceSystem}\} .$$

Since the attribute `remainingCapacity` does not belong to P, we do not need any additional constraint in order to prevent animation from changing it. However, since the association end `parkingGarage` belongs to P, animation of the operation `newTicket` may change the parking garages that existing tickets are linked to. Adding the postcondition

```
parkingGarages@pre
    ->forAll(tickets->excluding(result) = tickets@pre)
```

prevents this.

4 An Intermediate Representation for Constraints

OCL constraints have a complex structure. 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 animation.

4.1 Arithmetic Formulas with Bounded Quantifiers and Uninterpreted Functions

As intermediate representation we choose arithmetic formulas with bounded quantifiers and uninterpreted functions. These formulas have the following form.

1. Function symbols represent uninterpreted functions mapping $\mathbb{Z}^n$ to $\mathbb{Z}$.
2. We assume the availability of additional function symbols which represent arithmetic operations occurring in OCL expressions such as addition, subtraction and multiplication. The functions $\min(t_1, t_2)$ and $\max(t_1, t_2)$ return the minimum and the maximum of their arguments, respectively. Integer constants are available as well.
3. Variables are terms. Variables can assume values in $\mathbb{Z}$.
4. A function symbol applied to terms is a term.
5. Terms applied to the binary predicates $=$, $<$, $\leq$, $>$ and $\geq$ are formulas.
6. Formulas can be connected using the usual boolean operations.
7. If A is a formula and t_1, t_2 are terms, then $\forall t_1 \leq x \leq t_2 \,.\, A$ is a formula. Here t_1 is the lower bound and t_2 is the upper bound of the quantifier. Similarly, $\exists t_1 \leq x \leq t_2 \,.\, A$ is a formula.

Given an assignment of specific functions to the function symbols, the truth of such a formula without free variables is decidable since all quantifiers are bounded. Such an assignment for which the formula is true is called a *model*. Moreover, if we know that the ranges of the uninterpreted functions are restricted to certain finite intervals, then we can derive upper and lower bounds for every term. This means that the quantifier bounds are bounded themselves. As a result, it is decidable whether a model with restricted function ranges exists.

It turns out that OCL constraints can be translated to semantically equivalent arithmetic formulas with bounded quantifiers. This may seem surprising, since expressions in the specification languages Z, B or JML cannot be mapped to formulas with bounded quantifiers. Note that these languages allow stating number theoretic propositions as constraints such that the constraints are satisfied iff the propositions are true. These propositions include statements like Fermat's Last Theorem that clearly are difficult to verify. However, OCL was designed to facilitate runtime constraint checking and does not support such constraints.

4.2 Translation of OCL Constraints to Arithmetic Formulas

We now sketch how OCL constraints can be translated to arithmetic formulas. OCL expressions of type `Integer` can be translated almost literally to terms. Recall that we assume that any necessary arithmetic operations are available as function symbols. Similarly, OCL expressions of type `Boolean` can be translated directly to formulas.

Expressions whose type is a class are mapped to a pair (t_1, t_2) of arithmetic terms. The first term t_1 describes the type of the object which is the expression value. For this purpose, we assign an integer to every non-abstract class in the model. The second term t_2 gives the precise identity of the object. For every non-abstract class c, we introduce a constant f'_c that gives the number of instances of c at postcondition time. The term t_2 identifies the object by denoting an integer between 0 and $f'_c - 1$. We introduce another function symbol f_c to give the number of class instances at precondition time. Since we do not allow animation to delete objects, and we use the same integers for identifying objects at precondition time as we do at postcondition time, the object was created by animation iff $t_2 \geq f_c$. This relationship is used for translating the `oclIsNew` operation. If the procedure described in Section 3 determines that animation is not supposed to create new instances of the class, we set f'_c equal to f_c.

Now we turn to expressions whose type is a collection type. We describe the translation for the type `OrderedSet`, other collection types can be handled similarly. An expression of type `OrderedSet` is mapped to a triple $(s, x, E(x))$, where s is an arithmetic term, x is a variable and $E(x)$ is itself a translation of an OCL expression whose type is the element type of the collection. As indicated by the notation, x may occur in $E(x)$. The term s gives the size of the `OrderedSet`. For every integer i between 0 and $s - 1$, the i-th element of the `OrderedSet` is described by $E(i)$, i.e. the translation $E(x)$ with i substituted for x. OCL iterator expressions like `forAll` and `exists` expressions can be translated by using $E(x)$ as the translation of the iterator variable. The translation of the iterator expression can then be obtained by applying the corresponding bounded quantifier to x with lower bound 0 and upper bound $s - 1$.

Example 2. We show how the translation is derived for the the postconditions of the operation contract in Fig. 1. We assign the integer 0 to the class `Ticket` and introduce the function symbols $f_{\texttt{Ticket}}$ and $f'_{\texttt{Ticket}}$ as described above. The function symbol $f_{\texttt{result}}$ indicates the identity of the object returned by the operation. Recall that the integer identifying an object of class c must be between 0 and $f'_c - 1$. We achieve this by applying the functions $\min$ and $\max$ to the value returned by $f_{\texttt{result}}$. Thus, the identity of the `result` object corresponds to the value of the term

$$t_{\texttt{result}} := \min\left(\max\left(f_{\texttt{result}}, 0\right), f'_{\texttt{Ticket}} - 1\right),$$

and the translation of the OCL expression `result` is the term pair $(0, t_{\texttt{result}})$. The expression `result.oclIsNew()` is translated to $t_{\texttt{result}} \geq f_{\texttt{Ticket}}$. We use the function symbol f_{def} for expressing that `result` assumes an undefined value as $f_{def} = 0$. This is the translation of the expression `result.oclIsUndefined()`.

We assign the integer 1 to the class `ParkingGarage` and use the function symbols $f_{\texttt{ParkingGarage}}$ and $f'_{\texttt{ParkingGarage}}$ for holding the numbers of instances. For the attribute `remainingCapacity` we introduce the function symbol $f_{\texttt{remainingCapacity}}$ with arity two for representing a function which maps `ParkingGarage` instances to their attribute values at precondition time. Since the postconditions of the operation contract only access this attribute at precondition time, we do not need an additional function symbol for the attribute values at postcondition time.

The function symbol $f'_{\texttt{parkingGarage}}$ with arity two represents a function which maps `Ticket` instances to integers identifying the objects at the association end `parkingGarage` at postcondition time. The identity of the object `result.parkingGarage` corresponds to the value of the term

$$t_{\texttt{result.parkingGarage}} := \\ \min\left(\max\left(f'_{\texttt{parkingGarage}}\left(0, t_{\texttt{result}}\right), 0\right), f'_{\texttt{ParkingGarage}} - 1\right),$$

and the translation of the expression `result.parkingGarage` becomes $(1, t_{\texttt{result.parkingGarage}})$. The translation of the expression

`result.parkingGarage.remainingCapacity@pre > 0` is

$$f_{\texttt{remainingCapacity}}\left(1, t_{\texttt{result.parkingGarage}}\right) > 0 .$$

We assign the integer 2 to the class `ParkingGuidanceSystem` and introduce the constant $f_{\texttt{self}}$ in order to translate the `self` variable to the term pair $(2, f_{\texttt{self}})$. The identity of the `self` object is already determined prior to animation, so we can fix a value for $f_{\texttt{self}}$ which is passed separately to the formula analyzer. Hence, we do not need to use $\min$ and $\max$ operations for restricting the range of the second term like we did above. We use the function symbols $f_{\texttt{numberOfRequests}}$ and $f'_{\texttt{numberOfRequests}}$ for holding the values of the `numberOfRequests` attribute at pre- and postcondition time, respectively. The translation of the expression

`numberOfRequests = numberOfRequests@pre + 1` becomes

$$f'_{\texttt{numberOfRequests}}\left(2, f_{\texttt{self}}\right) = f_{\texttt{numberOfRequests}}\left(2, f_{\texttt{self}}\right) + 1 .$$

The function symbol $f_{\texttt{size}}$ represents the number of objects at the association end `parkingGarages`, and we introduce the function symbol $f_{\texttt{parkingGarages}}$ for giving the objects at the association end (both for precondition time). Thus, the translation of the expression `parkingGarages@pre` is

$$\left(f_{\texttt{size}}\left(2, f_{\texttt{self}}\right), x, \left(1, f_{\texttt{parkingGarages}}\left(2, f_{\texttt{self}}, x\right)\right)\right) .$$

We obtain the translation of

`parkingGarages@pre->forAll(remainingCapacity@pre = 0)` as

$$\forall\, 0 \leq x \leq f_{\texttt{size}}\left(2, f_{\texttt{self}}\right) - 1 \,.$$
$$f_{\texttt{remainingCapacity}}\left(1, f_{\texttt{parkingGarages}}\left(2, f_{\texttt{self}}, x\right)\right) = 0 \,.$$

The arithmetic formulas we derived here can be connected by standard Boolean operations in order to obtain translations of the complete postconditions in Fig. 1.

4.3 Finding Models for Arithmetic Formulas

For animating an operation, we translate the postconditions of the operation and all relevant invariants[5] as outlined above. We then attempt to find a model for the conjunction of the resulting formulas. The system state in which the operation is called is used to specify a partial solution, i.e. we only search for models that comply with the respective system state at precondition time. Since our translation preserves the semantics of the operation contract, a model found yields a new system state conforming with the contract. The new system state can be directly constructed from the model.

There are a number of software packages available for finding models. These *model finders* accept classes of formulas that are similar to our arithmetic formulas. We use the model finder Kodkod [20] in the back-end of our animator. The arithmetic formulas are translated to formulas that Kodkod accepts. Like many other model finders for first-order formulas, Kodkod encodes the problem in propositional logic and solves the resulting propositional problem using a satisfiability (SAT) solver. Integer expressions are encoded as vectors of Boolean expressions. An integer variable can be modelled by a vector of Boolean variables. An integer function is represented as an array of such variable vectors. 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 order to generate a propositional formula from a first-order formula, model finders need to eliminate the quantifiers occurring in the first-order formula. Some quantifiers can be removed by skolemization. For the remaining quantifiers, the quantified variable has to be substituted for every value within the range of the quantifier. Thus, the quantifier ranges need to be small enough in order to make this quantifier elimination feasible. Some model finders cope with this difficulty 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. Kodkod instead requires the user to give bounds that imply restricted quantifier ranges.

[5] These are the invariants in the set I defined in Section 3.

We obtain sufficiently bounded quantifiers by restricting the ranges of certain function symbols occurring in our formulas. We determine the function symbols that could be relevant for quantifier bounds and restrict their ranges. If no model is found for these restricted ranges, a more expensive attempt with larger ranges is made, and so on. Function symbols which are not relevant for quantifier bounds are only restricted to ranges which are certainly sufficient, e.g. an integer value may be restricted to a 32-bit number.

This means that 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 symbol which is relevant for quantifier bounds is $f_{\texttt{size}}$. This function symbol represents the number of objects at the association end `parkingGarages` at precondition time. Since this value is already fixed at precondition time, optimal quantifier bounds can be used for the translation in this case. If the association end was referenced by the constraint at postcondition time, increasing upper bounds for the function symbol would be tried when searching for models. On the other hand, the function symbol for the integer attribute `numberOfRequests` is not relevant for quantifier bounds, so we consider all possible values of this attribute already in the first attempt to find a new system state. Thus, even system states with very large values for this attribute are not necessarily problematic for animation.

5 Implementation

We describe our code generation tool OCLexec in more detail. The input to the tool is an XMI file containing an UML model and a separate text file with OCL constraints[6]. We use the Eclipse Model Development Tools library for handling the UML model and the constraints. At the time of writing, our tool merely exists as a first prototype, hence only restricted subsets of UML and OCL are supported. However, our approach to specification execution can cope with almost all language features of OCL. We have already implemented all four collection types of OCL, but currently only a subset of the collection operations is provided. Properties can have arbitrary multiplicities. The operation `oclIsNew` for testing whether an object has been created by an operation call is supported. We have implemented the undefined values `null` and `invalid`, which are propagated during expression evaluation as prescribed by the OCL standard.

For every class defined in the model, the tool looks for corresponding Java source code. If a declaration of a method is found for which a postcondition is defined in the OCL file, the constraints that are relevant for this operation are processed. An intermediate representation (see Section 4) of these constraints is serialized to a file which can be accessed as a resource. A method body is inserted into the source code that only consists of a call to a library routine responsible for executing the operation. The name of the resource with the serialized constraints and the arguments to the method are the ar-

[6] OCL constraints can also be stored with the model in an XMI file. We read a separate file with constraints since many modeling tools do not support editing of OCL constraints.

guments to the library routine. The Java source code modifications are performed using Java abstract syntax trees provided by the Eclipse Java Development Tools library.

The tool also adds a private no-argument constructor with empty body to every class in the model whose source code is found. The new constructor is only added if the class does not already provide a no-argument constructor. This constructor is needed by the library routine to create new class instances via Java reflection. Actually, it is likely that not all classes need new instances for executing a certain set of operations (see Section 3), so a no-argument constructor does not have to be provided by every class.

Note that inserting code in method bodies and adding constructors should not interfere with other code that may have been generated for the model, such as code for state charts. Thus, the modeler can use her favorite tool for the overall code generation and then use our tool only for selected method bodies.

The library routine responsible for executing the operation reads the serialized constraints. It traverses the parts of the object graph that are accessible through the object references it received. This is done using Java reflection. During the traversal the library routine collects instances of classes that are relevant for executing the operation. Values of properties of these classes are recorded. This information about class instances and property values at the time of the operation call is passed together with the constraints to the back-end of the tool (see Section 4.3). If a valid new system state exists, the back-end returns the description of a valid state. Property values that have changed in the new system state are written, and new class instances are created if needed. These modifications of the system state are also performed using reflection.

6 Experimental Results

We show that existing tools scale very poorly for some simple constraints, while our method remains robust due to the high maturity of SAT solvers. OCLexec is compared to the following tools:

– The `jmle` tool [8], which is included in the Common JML Tools distribution and generates Java bytecode from JML contracts.
– Brama [9], a commercial animation tool for the B language that is integrated in the Rodin IDE. Brama animates a B specification and displays the results graphically.

	OCLexec	`jmle`	Brama	ProB	JML-Test.	Alloy
`newTicket` (Fig. 2(a))	1.1	—[a]	—[b]	0	—[b]	0.3
`solve` (`c` = 500)	0.8	—[a]	>500	15	0	16
`fill12` (`size` = 10)	0.8	16	—[b]	0	—[b]	0.2
`fill12` (`size` = 15)	1.0	57	—[b]	0	—[b]	0.3
`makeDescending` (`size` = 15)	1.1	—[a]	—[b]	10	—[b]	1.2
`makeDescending` (`size` = 20)	0.6	—[a]	—[b]	340	—[b]	2.0

[a] An exception is thrown at runtime.
[b] Animation was not possible due to unsupported language features.

Figure 3. Experimental results: execution times in seconds

```
context Linear::solve(c : Integer) :
post: (-500 <= a and a <= 500) and (-500 <= b and b <= 500)
post: b > -200 and a > b + c
```

(a) In OCL

```
MACHINE Linear
   VARIABLES a, b
   INVARIANT a : (-500..500) & b : (-500..500)
   INITIALISATION a, b := 1, 1
   OPERATIONS
      solve(c) =
         PRE c: (1..1000) THEN
            ANY x1, x2
            WHERE x1 : (-500..500) & x2 : (-500..500)
                      & x2 > -200 & x1 > x2 + c
            THEN a, b := x1, x2
            END
         END
END
```

(b) In B

```
public class Linear {
   public int a, b;

   /*@ assignable a, b;
       ensures -500 <= a && a <= 500 && -500 <= b && b <= 500
                && b > -200 && a > b + c; */
   public void solve(int c);
}
```

(c) In JML

```
module linear
pred solve(a: Int, b: Int)
{
   -500 <= a and a <= 500 and -500 <= b and b <= 500
   b > -200 and a > b + 500
}
run solve for 11 int
```

(d) In Alloy

Figure 4. A contract with linear integer constraints

- ProB [5], a development tool for the B language that provides interactive animation among other functionality.
- The JML-Testing-Tools animator [7], an interactive animation tool for JML.
- The Alloy Analyzer [19]. Although Alloy focuses more on specification validation than prototyping, we include it in the comparison because it is a popular tool for object-oriented analysis.

We evaluate these tools on the following operation contracts:

```
context IntList::fill12(size: Integer):
post: list->size() = size
post: list->forAll(x | x = 1 or x = 2)
post: Sequence{0..size-2}
        ->forAll(i | list->at(i) <= list->at(i+1))
```

Figure 5. An OCL operation contract for creating a sorted list of ones and twos

```
context IntList::makeDescending(size: Integer):
pre: size >= 0 and size <= 30
post: list->size() = size
post: list->forAll(x | x >= 1 and x <= 30)
post: Sequence{0..size-2}
        ->forAll(i | list->at(i) > list->at(i+1))
```

Figure 6. An OCL operation contract for creating a strictly descending list of integers between 1 and 30

- The operation contract in Fig. 1 is executed using the system state in Fig. 2(a).
- The operation `solve` (Fig. 4) solves a linear constraint on integers.
- The operation `fill12` (Fig. 5) creates a list that consists of ones and twos such that any one appears before any two on the list. The size of the list is passed as argument.
- The operation `makeDescending` (Fig. 6) constructs a strictly descending list of integers between 1 and 30 whose size is passed as an argument.

The results of the evaluation are listed in Fig. 3. The execution times were measured on a machine with 256 MB RAM and a 1.4 GHz Intel celeron CPU. The SAT solver used was MiniSat [22], a well-known state-of-the-art SAT solver.

OCLexec scales well for all operation contracts. The JML-Testing-Tools animator can solve the quantifier-free constraint in Fig. 4 fast, but could not process quantifiers in the version available to the authors[7], so it could not be applied to the other operation contracts. ProB scales well for some operations like `fill12`, but scales poorly for similar contracts like `makeDescending` and solves the quantifier-free constraint relatively slowly. Alloy generally scales well. Recall that Alloy uses a SAT solver as does our tool. However, Alloy handles large integers inefficiently. This is why Alloy performs relatively poorly for the quantifier-free constraint.

7 Conclusions and Future Work

We presented an approach to executing underspecified OCL operation contracts with a SAT solver. OCL postconditions and class invariants are translated via arithmetic formulas to a satisfiability problem. A SAT solver is used to compute the operation results. The approach has been implemented in a prototype tool that generates Java

[7] We used a binary obtained in 2005.

method bodies from OCL operation contracts. Experimental results show that our SAT-based technique can be more efficient than existing approaches.

Not all features of UML/OCL are supported by the current prototype. Some language features of OCL will require a bit of thought to implement, such as calls to query operations which are themselves specified by postconditions. It is likely that the efficiency of the generated code can still be improved. The external software that our constraint solving back-end uses is not tailored for integer constraints. Since our approach is based on integer arithmetic, it appears promising to redesign the back-end. Maybe an efficiency improvement can also be achieved by using Satisfiability Modulo Theories (SMT) solvers instead of conventional SAT solvers. Finally, we would like to refine the rules for determining the parts of the system state that can be modified during operation execution. This could free the modeler from writing many postconditions that preserve parts of the system state.

References

1. Object Management Group: Object Constraint Language Specification, Version 2.0. Specification, OMG (2006) `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.
2. Hußmann, H., Demuth, B., Finger, F.: Modular Architecture for a Toolset Supporting OCL. Sci. Comp. Prog. **44**(1) (2002) 51–69
3. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based Specification Environment for Validating UML and OCL. Sci. Comp. Prog. **69**(1–3) (2007) 27–34
4. Dick, A., Krause, P., Cozens, J.: Computer Aided Transformation of Z into Prolog. In Nicholls, J., ed.: Proc. 4th Z Users Workshop. Workshops in Computing, Oxford, Springer (1989) 71–85
5. Leuschel, M., Butler, M.: ProB: An Automated Analysis Toolset for the B Method. Int. J. Softw. Tools Tech. Trans. **10**(2) (2008) 185–203
6. Grieskamp, W.: A computation model for Z based on concurrent constraint resolution. In Bowen, J.P., Dunne, S., Galloway, A., King, S., eds.: Proc. 1st Int. Conf. B and Z Users (ZB'00). Volume 1878 of Lect. Notes Comp. Sci., Springer (2000) 414–432
7. Bouquet, F., Dadeau, F., Legeard, B., Utting, M.: Symbolic animation of JML specifications. In: Proc. 13th Int. Conf. Formal Methods 2005 (FM'05). Volume 3582 of Lect. Notes Comp. Sci., Springer (2005) 75–90
8. Krause, B., Wahls, T.: jmle: A Tool for Executing JML Specifications Via Constraint Programming. In Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J., eds.: Rev. Sel. Papers 5th Int. Wsh. Parallel and Distributed Methods for Verification (PDMC'06). Volume 4346 of Lect. Notes Comp. Sci., Springer (2006) 293–296
9. Servat, T.: BRAMA: A New Graphic Animation Tool for B Models. In Julliand, J., Kouchnarenko, O., eds.: Proc. 7th Int. Conf. B Users (B'07). Volume 4355 of Lect. Notes Comp. Sci., Springer (2007) 274–276
10. Büning, H.K., Zhao, X., eds.: Proc. 11th Int. Conf. Theory and Applications of Satisfiability Testing (SAT'08). Volume 4996 of Lect. Notes Comp. Sci., Springer (2008)
11. Doma, V., Nicholl, R.A.: EZ: A System for Automatic Prototyping of Z Specifications. In Prehn, S., Toetenel, W.J., eds.: Proc. 4th Int. Symp. VDM Europe (VDM'91). Volume 551 of Lect. Notes Comp. Sci. (1991) 189–203
12. Utting, M.: Data Structures for Z Testing Tools. In Schellhorn, G., Reif, W., eds.: Proc. 4th Wsh. Tools for System Design and Verification (FM-TOOLS'00), Technical Report 2000-07, Universität Ulm (2000)

13. Bouquet, F., Legeard, B., Peureux, F.: CLPS-B — A Constraint Solver to Animate a B Specification. Int. J. Softw. Tools Tech. Trans. **6**(2) (2004) 143–157
14. Oliver, I., Kent, S.: Validation of Object Oriented Models using Animation. In: Proc. 25th Conf. EUROMICRO, IEEE Computer Society (1999) 2237–2242
15. Gray, J., Schach, S.: Constraint animation using an object-oriented declarative language. In Turner, A.J., ed.: Proc. 38th ACM Southeast Reg. Conf., ACM (2000) 1–10
16. Kim, S., Zhang, H.: ModGen: Theorem Proving by Model Generation. In: Proc. 12th Nat. Conf. Artificial Intelligence (AAAI'94), AAAI Press (1994) 162–167
17. McCune, W.: MACE 2.0 Reference Manual and Guide. Comp. Res. Rep. **6** (2001) `http://arxiv.org/abs/cs.LO/0106042`.
18. Claessen, K., Sörensson, N.: New Techniques that Improve MACE-style Finite Model Finding. In: Proc. Wsh. Model Computation — Principles, Algorithms, Applications, Miami, Florida (2003)
19. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
20. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In Grumberg, O., Huth, M., eds.: Proc. 13th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07). Volume 4424 of Lect. Notes Comp. Sci., Springer (2007) 632–647
21. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: Proc. 10th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'07). Volume 4735 of Lect. Notes Comp. Sci., Springer (2007) 436–450
22. Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: Sel. Rev. Papers 6th Int. Conf. Theory and Applications of Satisfiability Testing (SAT'03). Volume 2919 of Lect. Notes Comp. Sci., Springer (2004) 502–518