Electronic Communications of the EASST
Volume 44 (2011)

Proceedings of the Workshop on OCL and Textual Modelling (OCL 2011)

Extending OCL Operation Contracts with Objective Functions

Matthias P. Krieger and Achim D. Brucker

18 pages

ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Extending OCL Operation Contracts with Objective Functions

Matthias P. Krieger[1]* **and Achim D. Brucker**[2]

[1] krieger@lri.fr, http://www.lri.fr/~krieger/
Université Paris-Sud, Parc Club Orsay Université, 91893 Orsay Cedex, France

[2] achim.brucker@sap.com, http://www.brucker.ch/
SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany

Abstract: We explore the potential of adding objective functions to OCL operation contracts. If an operation contract includes an objective function, the operation has the obligation to yield results that make the objective function assume an optimal value. Thus, an objective function expresses a preference among the possible operation results that conform to the postconditions of the operation contract and any class invariants. Objective functions strictly increase the expressiveness of OCL operation contracts. While objective functions arise naturally in application domains like operations research, we argue that objective functions are a useful general-purpose specification instrument and discuss several application examples. As tool support for operation contracts with objective functions, we present an animator for OCL operation contracts with optimization capabilities. We ensure tool interoperability by specifying objective functions in a UML profile.

Keywords: OCL, Operation contracts, Model execution, Animation, SAT solvers

1 Introduction

OCL has been wisely conceived with executability in mind. The language omits constructs like unbounded quantifiers ranging over all integers that make expression evaluation intractable or entirely impossible. Recursion used for defining operations in postconditions is restricted by the OCL standard [Obj10] to be finite, so uncomputable operations are avoided. The collection constructors and operations are designed in a way that prevents an uncontrolled explosion of collection size. As a result, evaluators for OCL that check the conformance of an implementation to its OCL specification at runtime could be implemented without resorting to sophisticated reasoning techniques (e.g., [HDF02]). This is in contrast to other specification languages like Z [ISO] that offer more powerful constructs. For such languages constraint evaluation can be highly nontrivial, not to mention more difficult kinds of specification analysis like animation or test-case generation.

Naturally, the choice to restrict the expressive power of OCL comes at a price: some specification tasks may be impossible to accomplish or require considerably higher effort. Working around limitations of the language may also lead to specifications that obscure the problem that was to be specified originally. This may be one of the reasons why the use of OCL operation

* This work was partially supported by the Digiteo Foundation.

contracts appears to still not have gained widespread acceptance, although OCL has established itself as a language for model well-formedness rules and is also widely employed for queries in model-transformation and action languages.

A shortcoming of OCL in this respect that we identified is the difficulty to express optimization tasks. Such problems ask for operation results for which an objective function assumes an optimal value. Optimization problems arise naturally in application domains like operations research and constitute some of the most elementary algorithmic problems. A basic example is the problem of finding a shortest path in a graph. Figure 1 lists fundamental optimization algorithms covered in a classic introductory algorithms textbook [Sed88].

We propose to facilitate the specification of such operations in OCL by adding objective functions to operation contracts. Thus, rather than enriching the expression language of OCL, we introduce an additional constituent of operation contracts. Objective functions in operation contracts would provide immediate support in OCL for specifying optimization problems. Moreover, objective functions can also be used as a convenient means to specify that an operation should return a solution to a certain constraint if a solution exists: make the objective function evaluate whether the constraint holds and return the optimal value only if this is the case. Altogether, we think that this extension would make OCL operation contracts more attractive by facilitating the specification of many operations.

Objective functions strictly increase the expressiveness of OCL operation contracts. With the presence of an objective function, it is no longer decidable whether a set of returned operation results conforms to an operation contract. We propose to achieve tool interoperability by specifying objective functions in a UML profile. Existing tools can simply ignore the additional information represented by objective functions. Thus, existing applications of OCL are not compromised by the introduction of objective functions.

The paper is organized as follows. In Section 2 we show how objective functions can be included in operation contracts. Section 3 discusses applications of objective functions to different specification tasks by means of an example specification. We see that objective functions help not only for specifying ordinary optimizations problems, but also in other situations. In Section 4 we present an animation tool that supports operation contracts with objective functions and give experimental results. Finally, we discuss related work and conclude.

- Closest pair among a set of points
- Minimum spanning tree of a graph
- Shortest path in a graph
- Maximum network flow
- Maximum matching of a graph
- Regression: Least Squares
- Knapsack problem
- Linear programming

Figure 1: Optimization algorithms covered in a classical algorithms textbook [Sed88]

2 Operation Contracts with Objective Functions

2.1 Syntax

Together with class invariants and different kinds of value definitions, operation contracts are a major ingredient of OCL specifications. Recall that an OCL operation contract consists of pre- and postconditions which are Boolean expressions. We propose to allow objective functions as a further element of operation contracts. Obviously, the type of an objective function must support comparison, so we restrict objective functions to expressions of type `Integer` or `Real`.[1] This corresponds to the restrictions imposed on body expressions of the `sortedBy` iterator. Furthermore, every operation contract may include at most one objective function.

Another useful extension proposed to OCL operation contracts are invariability clauses [Kos06] that specify which parts of the system state an operation may modify. We will also consider this extension since invariability clauses are essential for animation support that we will discuss later.

In all, an OCL operation contract for an operation op with the arguments $x_1,\dots,x_n$ with these extensions has the form:

$$\begin{aligned}
&\texttt{context}\ C\ ::\ op(x_1,\dots,x_n) : T\\
&\quad \texttt{pre} : \phi(\mathit{self},x_1,\dots,x_n)\\
&\quad \texttt{pre} : \dots\\
&\quad \texttt{post} : \psi(\mathit{self},x_1,\dots,x_n,\mathit{result}) \qquad (1)\\
&\quad \texttt{post} : \dots\\
&\quad \texttt{minimize} : \theta(\mathit{self},x_1,\dots,x_n,\mathit{result})\\
&\quad \texttt{modifies only} : t_1(\mathit{self},x_1,\dots,x_n) :: a_1,\dots,t_m(\mathit{self},x_1,\dots,x_n) :: a_m
\end{aligned}$$

Here, the OCL expressions ϕ and ψ are of type Boolean and θ is an OCL expression of type `Integer` or `Real`. The OCL expressions $t_1,\dots,t_m$ denote sets of objects in the pre-state. We require that **`@pre`** does not occur in ϕ or $t_1,\dots,t_m$. The operation contract (1) requires the operation to minimize the function θ. Thus, we extend the concrete syntax of operation contracts with the keyword **`minimize`**. Of course, a corresponding **`maximize`** keyword can be introduced as syntactic sugar as well. In short, the **`modifies only`** clause in (1) specifies that the operation may only change the attribute a_i for the objects in t_i. Attributes not mentioned in the **`modifies only`** clause may not be changed for any object. A richer syntax for **`modifies only`** clauses is presented in [Kos06].

OCL operation contracts are often defined in UML models by storing OCL expressions as specifications of `Constraint` model elements. `Operation` elements can then reference such constraints through associations provided for by the UML metamodel. In order to avoid metamodel incompatibilities and to ensure the interoperability with OCL tools that do not use our operation contract extensions, we define the new operation contract elements in a UML profile. Figure 2 shows a UML profile for extending operation contracts with objective functions and invariability clauses. The objective function is stored as a string and can be parsed when

[1] A further syntax extension could allow user-defined comparison functions.

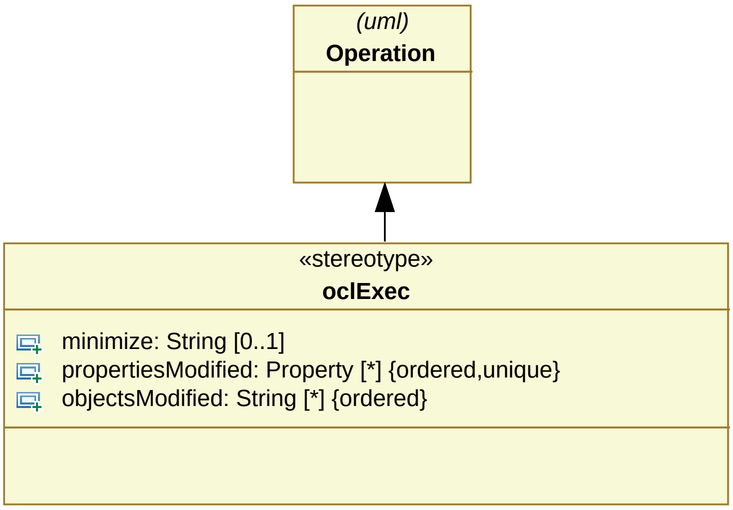
(uml)
Operation
«stereotype»
oclExec
minimize: String [0..1]
propertiesModified: Property [*] {ordered,unique}
objectsModified: String [*] {ordered}

Figure 2: A UML profile for operation contract extensions

needed. The **modifies only** clause is defined by listing the expressions $t_1,\dots,t_m$ and the attributes $a_1,\dots,a_m$ in separate attributes of the stereotype.

It may be more systematic to rather model objective functions and **modifies only** clauses as separate stereotypes of the `Constraint` metaclass. However, the profile in Figure 2 is simple and can be applied easily in UML editors.

2.2 Semantics

When discussing the semantics of operation contracts we assume without loss of generality that there is exactly one precondition ϕ and one postcondition ψ. Following Annex A of the OCL standard [Obj10], we define the *semantics* of an operation contract to be a relation R between pre-*environments* r_{pre} and post-environments r_{post}. An environment includes the system state and values of parameters to the operation. R is defined by

$$R = \left\{ \left(r_{\text{pre}}, r_{\text{post}}\right) \mid \phi\left(r_{\text{pre}}\right) \land \psi\left(r_{\text{pre}}, r_{\text{post}}\right) \right\}. \tag{2}$$

Thus, a transition from a pre- to a post-environment is permitted by the semantics if both the pre- and the postcondition are satisfied. The behavior of an operation implementation can be described by a function f that maps pre-environments to post-environments. The operation implementation *satisfies* the contract if and only if the graph of f is contained in R ($\text{graph}(f) \subseteq R$).

We show how this original definition of R can be modified in order to take an objective function θ into account. Let the contract require that the operation minimizes θ. This is expressed by the following definition of the modified semantics R':

$$\begin{aligned} R' = \Big\{ \left(r_{\text{pre}}, r_{\text{post}}\right) \mid \phi\left(r_{\text{pre}}\right) \land \psi\left(r_{\text{pre}}, r_{\text{post}}\right) \\ \land \quad \forall\, r'_{\text{post}}.\ \psi\left(r_{\text{pre}}, r'_{\text{post}}\right) \implies \theta\left(r_{\text{pre}}, r_{\text{post}}\right) \le \theta\left(r_{\text{pre}}, r'_{\text{post}}\right) \Big\}. \end{aligned} \tag{3}$$

Thus, a pair of a pre- and post-environment $\left(r_{\text{pre}}, r_{\text{post}}\right)$ can only belong to R' if there is no other post-environment r'_{post} satisfying the postcondition with a smaller objective value. Hence, the objective function constrains the set of permitted transitions and may forbid transitions that are allowed by the original semantics R. Note that there only is a difference to the original semantics if the contract is underspecified, i.e., there is a valid pre-environment for which there is more than one post-environment satisfying the postcondition. Otherwise, $r'_{\text{post}} = r_{\text{post}}$ whenever $\psi\left(r_{\text{pre}}, r'_{\text{post}}\right)$, and the additional condition in (3) could never be violated. Thus, the objective function selects preferred post-states in case several are permitted by the postconditions.

Also note that the addition of the objective function increased the expressiveness of OCL operation contracts since the new semantics R' cannot in general be obtained by simply adding a postcondition $\theta\left(r_{\text{pre}}, r_{\text{post}}\right) \leq c\left(r_{\text{pre}}\right)$ for some OCL expression c. This only is an alternative if there exists such an expression c that computes the optimal value of the objective function from the pre-environment. However, this cannot always be the case, since the values of OCL expressions are effectively computable, but the existence of a post-environment r'_{post} violating (3) is in general undecidable. Even in cases in which it is possible to designate such an expression c, it is likely that this expression is much more complex than the addition of an objective function to the operation contract.

Finally, note that objective functions in operation contracts differ considerably from the `min` and `max` operations on collections that are provided by the OCL standard library. While these operations select the minimum and maximum from a finite collection, objective functions operate on the set of all possible post-states, which tends to be much larger than an OCL collection or can even be infinite.

If a subclass redefines the operation, the semantics of the redefined operation must conform to the Liskov substitution principle. Specifically, the set of possible post-states r_{post} that can result from calling the redefined operation in a certain pre-state r_{pre} satisfying the precondition ϕ must be a subset of the set of post-states reachable from this pre-state r_{pre} by calling the operation in the superclass. The objective function can be redefined in the subclass as long as this requirement is met.

The **`modifies only`** clause further constrains the semantics of the operation contract. However, unlike for objective functions, it is also possible to express this restriction through postconditions. Such a transformation is described in [Kos06].

3 Applications of Objective Functions

In this section we demonstrate the usefulness of objective functions by presenting several application examples. As running example we use the simplified model of a build tool shown in Figure 3. A `Project` comprises several `CompilationUnits`. In general, the task of the build tool is to arrange all `CompilationUnits` into adequate `CompilationJobs`. A `CompilationJob` designates an ordered sequence of `CompilationUnits` that are to be processed in order to complete the job. The assignment of `CompilationUnits` to `CompilationJobs` may be driven by various considerations. In particular, there can be dependencies between `CompilationUnits`, which is modeled by an association. Moreover, `CompilationUnits` can have different `sizes`.

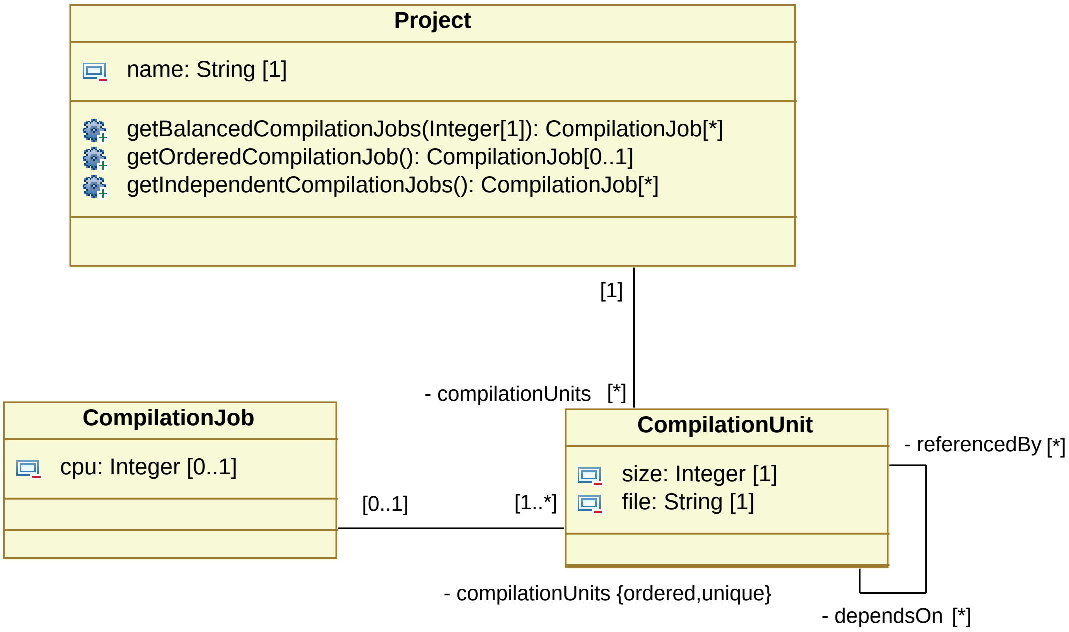
Project
name: String [1]
getBalancedCompilationJobs(Integer[1]): CompilationJob[*]
getOrderedCompilationJob(): CompilationJob[0..1]
getIndependentCompilationJobs(): CompilationJob[*]
[1]
- compilationUnits [*]
CompilationJob
cpu: Integer [0..1]
CompilationUnit
size: Integer [1]
file: String [1]
- referencedBy [*]
[0..1]
[1..*]
- compilationUnits {ordered,unique}
- dependsOn [*]

Figure 3: Excerpt from a possible UML model of a build tool

3.1 Ordinary Optimization Problems

As as first example, we consider a plain optimization task that is representative for many similar ones with an operations research background. The goal is to achieve efficient parallel processing of `CompilationJobs` by assigning `CompilationUnits` to `CompilationJobs` such that the longest execution time of any `CompilationJob` is minimized and the entire process is completed as early as possible. This is accomplished by the operation `getBalancedCompilationJobs` specified in Figure 4, which returns a set of `CompilationJobs` that arrange the `CompilationUnits` of the `Project` such that total execution time is minimized. The parameter `n` indicates the maximal number of `CompilationJobs` created and would usually correspond to the number of CPUs available. The postcondition `allUnitsReturned` specifies that the set of `compilationUnits` included in the returned `CompilationJobs` are exactly the `CompilationJobs` of the `Project` for which the operation is called. The postcondition `jobLimitMet` limits the number of returned `CompilationJobs` to the argument passed to the operation. In this operation contract, most of the behavior is specified in the objective function. In the objective function, we first compute the total sizes of all returned `CompilationJobs` by applying `collectNested` and `sum`. We use `collectNested` here in order to make explicit for readability that no flattening is performed. Then the value of the objective function is defined to be the maximum of all total job sizes, which is an estimate of the total execution time. Finally, the **`modifies only`** clause specifies that the attribute `compilationJob` may only be changed for the `compilationUnits` belonging to the `Project` for which the operation is called and that the values of all other attributes may not be affected by the operation. This invariability clause make some previous **`@pre`** decorations

```
context Project::getBalancedCompilationJobs(n: Integer): Set(CompilationJob)

post allUnitsReturned:
   result->collect(compilationUnits)->asSet() = compilationUnits@pre

post jobLimitMet: result->size() <= n

minimize: let
             jobSizes: Bag(Integer)
                = result->collectNested(job |
                   job->compilationUnits->collectNested(size@pre)->sum())
          in
            jobSizes->max()

modifies only: compilationUnits::compilationJob
```

Figure 4: Operation contract for evenly distributing compilation units among processes

superfluous, but we decided to keep these in order to already make explicit in the postconditions that these are values from the pre-state.

This problem of optimizing parallel execution is NP-hard since the subset sum problem can be reduced to it. Thus, even a relatively simple implementation is likely to be substantially more complex than the operation contract.

3.2 Problems that do not always have a Solution

The task of the next operation `getOrderedCompilationJobs` that we specify is to find a compilation order that respects the dependencies of the `CompilationUnits`. In other words, we desire a topologically sorted sequence of the compilation units. The contract is shown in Figure 4. An interesting aspect of this operation is that there is no valid compilation order if the dependency graph has a cycle. In this case the operation should return `null`. The postcondition `allUnitsReturned` specifies that if the result is not `null`, i.e., a solution exists, then the set of `CompilationJobs` returned are exactly the `CompilationJobs` of the `Project`, as in the previous contract. The postcondition `resultSorted` states that, if the result is not `null`, no `CompilationUnit` depends on another unit occurring later in the returned sequence.

This operation does not solve an optimization problem at first sight. The objective function of the contract expresses that the operation should return a non-null result whenever possible, i.e., when it is not excluded by the postconditions. The presence of the objective function is essential for the contract to be complete, since otherwise an implementation that always returns `null` even if a valid compilation order exists would satisfy the contract. Note that it is not sufficient to simply add the additional postcondition **post: not** `result.oclIsUndefined()`, since this would make the postconditions unsatisfiable in case there is no valid compilation order, and the contract would not be implementable. An admissible alternative would be to add a precondition expressing that the postconditions are satisfiable, i.e., that the dependency graph is acyclic. This choice would be possible for this operation because the satisfiability of the postconditions is decidable in this case. It is clear, however, that such a precondition testing whether the depen-

```
context Project::getOrderedCompilationJobs(): CompilationJob

post allUnitsReturned: (not result.oclIsUndefined())
                       implies
                       result.compilationUnits->asSet()
                          = compilationUnits@pre

post resultSorted:
   (not result.oclIsUndefined())
   implies
   let
      units: OrderedSet(CompilationUnit) = result.compilationUnits
   in
      Sequence{2..units->size()}
         ->forAll(i | Sequence{1..i-1}
           ->forAll(j | units->at(j).dependsOn@pre->excludes(units->at(i))))

minimize: if result.oclIsUndefined() then 1 else 0 endif

modifies only: compilationUnits::compilationJob
```

Figure 5: Operation contract for ordering compilation units by dependencies

dency graph has a cycle would be much more lengthy then the objective function in Figure 4. Thus, the possibility of adding an objective function to the contract helped considerably to specify the operation in a concise and comprehensible manner. This technique of preferring non-null results by means of an objective function is applicable in general to problems that do not always have a solution.

3.3 Other Disguised Optimization Problems

Next we consider another task that is usually not regarded as an optimization problem. We are seeking a set of `CompilationJobs` that can be processed independently, e.g., for allowing parallel execution as discussed above for the operation `getBalancedCompilationJobs`. But this time we do not arrange the `CompilationUnits` based on their size but according to their dependencies. We require that there is no dependency between any two `CompilationUnits` belonging to distinct `CompilationJobs`. In order to be as flexible as possible for execution, we desire to have as many independent `CompilationJobs` as possible. This amounts to finding the connected components of the dependency graph. The operation `getIndependentCompilationJobs`, whose contract is shown in Figure 6, is specified to return a set of `CompilationUnits` that meets these requirements. As for the previous operations, the first postcondition expresses that the `CompilationUnits` included in the returned `CompilationJobs` are exactly the `CompilationJobs` of the `Project`. The next postcondition `jobsIndependent` states that every `CompilationJob` returned includes all `CompilationUnits` that depend on any unit in the job. This rules out any dependencies between any two `CompilationUnits` belonging to distinct `CompilationJobs`.

In order to ensure that the returned `CompilationJobs` correspond to the connected compo-

```
context Project::getIndependentCompilationJobs(): Set(CompilationJob)

post allUnitsReturned:
   result->collect(compilationUnits)->asSet() = compilationUnits@pre

post jobsIndependent:
   result->forAll(compilationUnits->forAll(
      compilationUnits->includesAll(dependsOn@pre)))

minimize: -result->size() -- this maximizes the size of the result set

modifies only: compilationUnits::compilationJob
```

Figure 6: Operation contract for grouping compilation units into independent jobs

nents of the dependency graph, we still need to specify that there actually is a dependency between every pair of `CompilationUnits` that belong to the same `CompilationJob`. Otherwise, an implementation may always return just a single `CompilationJob` that includes all `CompilationUnits`. However, we find this requirement difficult to express using postconditions, since two `CompilationUnits` may depend on each other via several other intermediate `CompilationUnits`. This is where the objective function comes in handy. The objective function defined in Figure 6 requires the operation to return a maximal number of `CompilationJobs`. This implies that every job corresponds to exactly one connected component of the dependency graph. Thus, the possibility of adding an objective function to the contract helped again to keep the contract simple.

4 Tool Support for Animation

Tool support is important for the adoption of specification languages by users. In this section we show how tool support for *animation* can be accomplished for OCL operation contracts that include objective functions. Animation [DKC89] is the task of performing computations that comply with the specification based on user-provided input data. For animating an operation contract, the user constructs a system state in which the operation is called and provides any arguments to the operation. Animation yields a new state and if necessary a return value that satisfy the postconditions, the objective function and any other restrictions stated in the specification such as class invariants.

If support for animation is available, users can validate requirements by animating the specification on sample sets of input data (*scenarios*). This is possible if the implementation is partially or even entirely unavailable. For incomplete, faulty or inadequate specifications, animation will typically lead to strange and alarming results. In contrast to automatically generated test cases, which result from exploring input data structures according to certain criteria, animation based on user-supplied input avoids too artificial scenarios. Animation can help users gain confidence in the specification by allowing the execution of scenarios that are common for the application domain.

Animation 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 the complete implementation of some parts is not yet available. It may even be possible to entirely omit the manual implementation of certain features if they can be animated efficiently enough. We think that animation 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.

In the remainder of this section, we first give an overview of our animation tool OCLexec[2]. See [KKW10] for a more detailed description. Then we show how the tool can be extended to deal with objective functions and present some experimental results.

4.1 Overview of OCLexec

OCLexec generates Java method bodies that enforce the postconditions of the operation and all class invariants. It serializes an intermediate representation 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 modeler can use her favorite tool for the overall code generation and then use OCLexec only for selected method bodies.

As intermediate representation, OCLexec uses a language of arithmetic expressions with bounded quantifiers and uninterpreted functions. A bounded quantifier has the form $\forall t_1 \leq x \leq t_2 \,.\, p(x)$. Here x is the bound variable, t_1 is the lower bound and t_2 is the upper bound of the quantifier. 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 that evaluate to 1. The underlying state of an OCL expression is represented by uninterpreted function symbols for attributes. Object references can be encoded as integer values. We have currently not implemented support for real numbers. Thus, we will later require objective functions do be of type `Integer`.

For animating an operation, we translate the postconditions of the operation and all relevant invariants to arithmetic constraints with bounded quantifiers. 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 construct a Boolean circuit that computes the validity of the formula. The Boolean circuit is encoded in conjunctive normal form (CNF), which can then be solved by an off-the-shelf satisfiability (SAT) solver. A solution to the Boolean satisfiability problem yields a model for the original arithmetic formula. In order to generate a Boolean formula from an arithmetic formula with bounded quantifiers, the quantifiers occurring in the arithmetic formula need to be eliminated. Some quantifiers can be removed by skolemization.

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

```
procedure solve($\varphi$):
  bounds := initial_bounds($\varphi$)
  forever do
    CNF := generate_CNF($\varphi$, bounds)
    (solved, solution) := call_SAT_solver(CNF)
    if solved then
      return solution
    end if

    increase_bounds(bounds)
  end forever
end procedure
```

Figure 7: Basic constraint solving procedure for operation contract animation

For the remaining quantifiers, we substitute the quantified variable for every value within the range of the quantifier, translate the resulting formulas to Boolean circuits and feed them into the respective gate ($\wedge$ or $\vee$). Thus, the quantifier ranges need to be small enough in order to make this quantifier elimination manageable. It is clearly not feasible to always perform the substitution for the largest possible quantifier range, which may include e.g. all 32-bit integers. Therefore we may at first restrict the values of certain function symbols in 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. 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.

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. This basic procedure is depicted in Figure 7. It always terminates when a solution exists. If no solution exists, the procedure may terminate unsuccessfully after the bounds have reached a limit, e.g., a maximal 32-bit number, or if the formula has no function symbols that require bounding. However, due to the large number of possible system states, the procedure will not always terminate within reasonable time.

4.2 Extension to Support Objective Functions

We now show how this basic animation procedure can be extended to support objective functions. The fundamental problem is that it is generally undecidable whether a better solution exists that improves the value of the objective function. Such a superior solution may only be constructible using much larger bounds. Therefore we encourage the user to supply a time limit for animation with an objective function in order to ensure termination. In order to promote early termination, we also consider over-approximations of the satisfiability problem based on the same bounds used for under-approximation as described above. If we determine that an over-approximation of the problem of finding an improved solution is unsatisfiable, we know that such a solution does not exist for any bounds and can stop the search. However, solving over-approximations is always incomplete, so unfortunately user-provided time limits are in general unavoidable.

We construct over-approximations by adding fresh Boolean variables to formulas resulting from the unfolding of quantifiers outlined above. These new variables represent the unknown value of the quantified formula in case the quantifier bounds exceed the bounds used for the translation. For a universal quantifier $\forall t_1 \leq x \leq t_2 \,.\, p(x)$ we construct the Boolean formula

$$\begin{array}{llllll} & (t_1 \leq \mathrm{lb}(t_1) & \wedge & \mathrm{lb}(t_1) & \leq t_2 \implies & p(\mathrm{lb}(t_1))) \\ \wedge & (t_1 \leq \mathrm{lb}(t_1)+1 & \wedge & \mathrm{lb}(t_1)+1 & \leq t_2 \implies & p(\mathrm{lb}(t_1)+1)) \\ \dots & & & & & \\ \wedge & (t_1 \leq \mathrm{ub}(t_2) & \wedge & \mathrm{ub}(t_2) & \leq t_2 \implies & p(\mathrm{ub}(t_2))) \\ \wedge & (t_1 < \mathrm{lb}(t_1) & \vee & \mathrm{ub}(t_2) & > t_2 \implies & a) \end{array} \tag{4}$$

where $\mathrm{lb}(t)$ and $\mathrm{ub}(t)$ are a lower and upper bound, respectively, of the values the term t can assume when the function values that t depends on are within the bounds used for the translation. The Boolean variable a is a fresh variable that represents the unknown value of the quantified formula in case the lower bound t_1 of the quantifier is smaller than $\mathrm{lb}(t_1)$ or the upper bound t_2 is larger than $\mathrm{ub}(t_2)$. The new variable a does not have any effect if $t_1 \geq \mathrm{lb}(t_1)$ and $t_2 \leq \mathrm{ub}(t_2)$ or the quantified formula evaluates to false due to the values of $p(\mathrm{lb}(t_1)), \dots, p(\mathrm{ub}(t_2))$.

Example 1 *Consider a quantified formula*

$$\varphi = \forall 0 \leq x \leq f+g \,.\, h(x) = 1. \tag{5}$$

Here f and g are constants (nullary functions) and h is a unary function. If we use bounds that restrict f and g to assume values in $[-2,2]$, we obtain through interval arithmetic an upper bound of 4 for $f+g$. We apply the scheme (4) with $t_1 = 0$ and $t_2 = f+g$. Since the lower bound t_1 is constant, comparisons with it can be eliminated, and we obtain the following unfolding of φ:

$$\begin{array}{lll} & (0 \leq f+g & \implies h(0) = 1) \\ \wedge & (1 \leq f+g & \implies h(1) = 1) \\ \wedge & (2 \leq f+g & \implies h(2) = 1) \\ \wedge & (3 \leq f+g & \implies h(3) = 1) \\ \wedge & (4 \leq f+g & \implies h(4) = 1) \\ \wedge & (4 > f+g & \implies a). \end{array} \tag{6}$$

The overall procedure for animating with an objective function is shown in Figure 8. We can test the existence of a solution meeting a certain bound on the value of the objective function by calling the SAT solver with an appropriate set of Boolean literals as *assumptions*. When called with assumptions, a SAT solver restricts the search to models in which the assumptions are true. Clauses learned by the solver remain and can be used when solving later under different assumptions. Both MiniSat [ES04] and SAT4J [BP10] are SAT solvers that provide an interface for solving under assumptions.

Operation	$n = 3$	$n = 5$	$n = 7$	$n = 10$	$n = 15$	$n = 20$
`getBalancedCompilationJobs(2)`	3.0	3.0	50	>1000	>1000	>1000
`getOrderedCompilationJob()`	2.5	12	37	>1000	>1000	>1000
`getIndependentCompilationJobs()`	3.0	9	10	140	>1000	>1000
`getCompilationOrder()` (input cyclic)	1.0	1.0	1.0	1.5	1.5	>1000
`getCompilationOrder()` (input acyclic)	1.0	1.0	1.0	1.5	1.5	>1000

Table 1: Experimental results (execution times in seconds)

For every choice of bounds we compute a solution by binary search that is optimal among all solutions within these bounds. We assume that the objective function assumes values within a certain interval [f_min, f_max] that we determine e.g. by restricting the values of all function symbols to 32-bit numbers. This has the additional benefit of preventing infinite looping in case there is no optimal solution.

We maintain a lower bound on the value of the objective function, which is to be minimized, with the hope of observing at some point that we have obtained a solution that is optimal unconditionally. After having searched for a solution that is as good as possible for the chosen bounds, we again perform a binary search for determining the best lower bound. For determining the lower bound we can reuse the same CNF since we can also control over-approximation over the assumption interface. We terminate after a solution has been found and the timeout is reached or a solution has been found whose value of the objective function matches the lower bound.

4.3 Experimental Results

We evaluated our animation tool on the specification presented in section Section 3 for random inputs. When generating an input with n compilation units, we added every possible dependency between compilation units with probability $0.1/n$, so it was sufficiently unlikely that a cyclic dependency graph was generated. File sizes were sampled uniformly between 0 and 10000. Unfortunately, our over-approximation scheme was not able to identify optimal solutions for any of the operations, so user-defined timeouts were necessary for animating these operations. Table 1 shows the total execution times for animating the operations obtained by adding the best attainable value of the objective function as an additional postcondition. 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 [ES04], a well-known state-of-the-art SAT solver.

We conclude that the animation of many interesting scenarios involving objective functions is possible. However, for large inputs animation becomes infeasible due to quickly increasing runtimes. These runtimes may seem disappointing, considering that polynomial-time algorithms exist for topological sorting and the identification of connected components. Recall that the we are processing a high-level specification in a relatively general-purpose language.

In order to evaluate our approach to over-approximation, we modified the contract of the operation `getOrderedCompilationJob` as shown in Figure 9. Over-approximation succeeds for this modified contract, which we animated with dependencies added with probability $10/n$, so the dependency graph very likely had a cycle. Table 1 shows the time required to determine that 1 is the optimal value of the objective function and the operation can return an empty result.

```
procedure solve($\varphi$, $f$, max_time):
  initialize timer for timeout
  bounds := initial_bounds($\varphi$)
  found_solution := false
  best_value := f_max + 1
  lower_bound := f_min
  do
    CNF := generate_CNF($\varphi$, bounds)
    local_min := lower_bound
    do
      mid := local_min + (best_value − local_min) div 2
      (solved, solution)
        := call_SAT_solver(CNF, $f \leq$mid, under−approximate)
      if solved then
        found_solution := true
        best_solution := solution
        best_value := mid
      else
        local_min := mid + 1
      end if
    while best_value > local_min

    local_max := best_value
    while lower_bound < local_max
      mid := lower_bound + (local_max − lower_bound) div 2
      (solved, solution)
        := call_SAT_solver(CNF, $f \leq$mid, over−approximate)
      if not solved then
        lower_bound := mid + 1
      else
        local_max := mid
      end if
    end while

    increase_bounds(bounds)
  while not found_solution or timeout(max_time)
          or lower_bound = best_value

  return best_solution
end procedure
```

Figure 8: Constraint solving procedure for animating with an objective function f

```
context Project::getCompilationOrder(): OrderedSet(CompilationUnit)

post allUnitsReturned: (not result->isEmpty())
                       implies
                       result->asSet() = compilationUnits@pre

post resultSorted:
   (not result->isEmpty())
   implies
   Sequence{2..result->size()}
      ->forAll(i | Sequence{1..i-1}
         ->forAll(j | result->at(j).dependsOn@pre->excludes(result->at(i))))

minimize: if result->isEmpty() then 1 else 0 endif

modifies only: nothing
```

Figure 9: “Tuned” variant of the operation contract in Figure 5

The table also show the time consumed by animating the modified contract when processing the acyclic graphs used for animating the other contracts, which are essentially identical to the runtimes for animating the cyclic dependency graphs with the same number of vertices. We note that animation of the modified contract in Figure 9 is more efficient than the original contract in Figure 5. The reason may be that, in contrast to the original contract, the modified contract does not specify the creation of new objects.

5 Related Work

Objective functions are widely used in mathematical modeling languages like AMPL [FGK02] or GAMS [GAM]. Such modeling languages can be processed by suited optimization engines. Modeling languages are often tailored to the kind of solver that reads them and usually favor convex or polynomial constraints and objective functions. Sugar [TTB10] is an example of a constraint programming language that features objective functions. Modelica [Mod] is an object-oriented mathematical modeling language.

We extended the UML/OCL language by means of a UML profile. A more comprehensive approach to extending OCL based on modularization is presented in [AZH08].

Pioneering work on animating UML/OCL can be found in [OK99, GS00]. Animators for operation contracts have also been implemented, for example, for the specification language JML [BDLU05, KW06, CW09]. The jmle animator [KW06, CW09] works by generating a prototype implementation in Java, as does our animator OCLexec. Another recent tool along these lines is Squander [MRYJ11] that animates operation contracts in a tailored specification language.

An alternative to our approach of encoding the constraints as a Boolean SAT problem is to employ solvers based on the lazy SMT paradigm [Seb07] like Z3 [MB08]. These solvers integrate 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. The Z3 solver supports a notion of so-called *contexts* that provide similar functionality as the assumption interface of MiniSat [ES04] and SAT4J [BP10]. Another type of solver applicable to this kind of optimization problem are MAX-SAT solvers, e.g., one of the solvers in the SAT4J package. Such solvers are likely to be more efficient than optimizing via assumptions since they are tailored to optimization. Sugar [TTB10] is a SAT-based constraint solver that also performs optimization over an assumption interface to a SAT solver. In the implementation of OCLexec, we use an adapted version of the Kodkod solver [TJ07] for constructing the CNF. Kodkod is a SAT-based solver for relational logic and includes effective algorithms for constructing and compressing Boolean representations. We do not make use of higher-level features of Kodkod such as encoding of relations. Kodkod also features a form of optimization based on appropriate support by the SAT solver used. Nitpick [BN10] is a SAT-based model finder that deals with problematic quantifiers by computing an undefined value when the value of the quantified formula is unknown. Our over-approximation approach can detect slightly more unsatisfiable constraints. The technique of under- and over-approximation that we use corresponds to abstraction-refinement techniques in verification [CGJ+03]. Such techniques can be more powerful if the under-approximation is computed according to the result of the over-approximation and vice versa. We do not use this kind of adaptive approximation in our approach.

6 Conclusions and Future Work

We showed how OCL can be extended with objective functions. Objective functions strictly increase the expressiveness of OCL operation contracts. Optimization problems arise naturally in application domains like operations research. Moreover, objective functions are also useful in various other situations and thus can be a valuable aid in writing concise and comprehensive operation contracts. We ensure tool interoperability by specifying objective functions in a UML profile.

As tool support for objective functions we present an animator for OCL operation contracts that supports objective functions. We perform optimization through the assumptions interface of the SAT solver used for constraint solving. Over-approximations of the constraint satisfaction problem are constructed for identifying optimal solutions. Since over-approximation is inevitably incomplete, user-defined timeouts are generally needed for animation with an objective function.

As future work we consider adding symmetry breaking predicates to the constraints solved for animating. This could improve the effectiveness of our over-approximation technique and may also improve the efficiency of constraint solving in general. We would also like to integrate our animation method into a comprehensive code generation system.

References

[AZH08] D. H. Akehurst, S. Zschaler, W. G. J. Howells. OCL: Modularising the Language. *ECEASST* 9, 2008.

[BDLU05] F. Bouquet, F. Dadeau, B. Legeard, M. Utting. Symbolic Animation of JML Specifications. In *Proc. 13th Int. Conf. Formal Methods 2005 (FM'05)*. Lect. Notes Comp. Sci. 3582, pp. 75–90. Springer, 2005.

[BN10] J. Blanchette, T. Nipkow. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In Kaufmann and Paulson (eds.), *Interactive Theorem Proving*. Lect. Notes Comp. Sci. 6172, pp. 131–146. Springer, 2010.

[BP10] D. L. Berre, A. Parrain. The Sat4j library, release 2.2. *JSAT* 7(2-3):59–64, 2010.

[CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50(5):752–794, 2003.

[CW09] N. Cataño, T. Wahls. Executing JML specifications of Java card applications: a case study. In Shin and Ossowski (eds.), *SAC*. Pp. 404–408. ACM, 2009.

[DKC89] A. Dick, P. Krause, J. Cozens. Computer Aided Transformation of Z into Prolog. In Nicholls (ed.), *Proc. 4th Z Users Workshop*. Workshops in Computing, pp. 71–85. Springer, Oxford, 1989.

[ES04] N. Eén, N. Sörensson. An Extensible SAT-solver. In Giunchiglia and Tacchella (eds.), *Sel. Rev. Papers 6th Int. Conf. Theory and Applications of Satisfiability Testing (SAT'03)*. Lect. Notes Comp. Sci. 2919, pp. 502–518. Springer, 2004.

[FGK02] R. Fourer, D. M. Gay, B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002.

[GAM] General Algebraic Modeling System (GAMS). http://www.gams.com.

[GS00] J. Gray, S. Schach. Constraint animation using an object-oriented declarative language. In Turner (ed.), *Proc. 38th ACM Southeast Reg. Conf.* Pp. 1–10. ACM, 2000.

[HDF02] H. Hußmann, B. Demuth, F. Finger. Modular Architecture for a Toolset Supporting OCL. *Sci. Comp. Prog.* 44(1):51–69, 2002.

[ISO] ISO/IEC 13568. Information technology — Z formal specification notation — Syntax, type system and semantics.

[KKW10] M. P. Krieger, A. Knapp, B. Wolff. Automatic and efficient simulation of operation contracts. In Visser and Järvi (eds.), *GPCE*. Pp. 53–62. ACM, 2010.

[Kos06] P. Kosiuczenko. Specification of Invariability in OCL. In Nierstrasz et al. (eds.), *MoDELS*. Lect. Notes Comp. Sci. 4199, pp. 676–691. Springer, 2006.

[KW06] B. Krause, T. Wahls. jmle: A Tool for Executing JML Specifications Via Constraint Programming. In Brim et al. (eds.), *Rev. Sel. Papers 5th Int. Wsh. Parallel and Distributed Methods for Verification (PDMC'06)*. Lect. Notes Comp. Sci. 4346, pp. 293–296. Springer, 2006.

[MB08] L. M. de Moura, N. Bjørner. Z3: An Efficient SMT Solver. In Ramakrishnan and Rehof (eds.), *TACAS*. Lect. Notes Comp. Sci. 4963, pp. 337–340. Springer, 2008.

[Mod] Modelica Association. Modelica Language Specification.

[MRYJ11] A. Milicevic, D. Rayside, K. Yessenov, D. Jackson. Unifying execution of imperative and declarative code. In Taylor et al. (eds.), *ICSE*. Pp. 511–520. ACM, 2011.

[Obj10] Object Management Group. Object Constraint Language Specification, Version 2.2. Specification, OMG, 2010. http://www.omg.org/spec/OCL/2.2.

[OK99] I. Oliver, S. Kent. Validation of Object Oriented Models using Animation. In *Proc. 25th Conf. EUROMICRO*. Pp. 2237–2242. IEEE Computer Society, 1999.

[Seb07] R. Sebastiani. Lazy Satisability Modulo Theories. *JSAT* 3(3-4):141–224, 2007.

[Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley, Second edition, 1988.

[TJ07] E. Torlak, D. Jackson. Kodkod: A Relational Model Finder. In Grumberg and Huth (eds.), *Proc. 13th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*. Lect. Notes Comp. Sci. 4424, pp. 632–647. Springer, 2007.

[TTB10] N. Tamura, T. Tanjo, M. Banbara. Solving Constraint Satisfaction Problems with SAT Technology. In Blume et al. (eds.), *FLOPS*. Lect. Notes Comp. Sci. 6009, pp. 19–23. Springer, 2010.