# Lightweight String Reasoning for OCL

Fabian Buettner, Jordi Cabot

# Lightweight String Reasoning for OCL

Fabian Büttner* and Jordi Cabot

AtlanMod, École des Mines de Nantes - INRIA, Nantes, France
{fabian.buettner,jordi.cabot}@inria.fr

**Abstract.** Models play a key role in assuring software quality in the model-driven approach. Precise models usually require the definition of OCL expressions to specify model constraints that cannot be expressed graphically. Techniques that check the satisfiability of such models and find corresponding instances of them are important in various activities, such as model-based testing and validation. Several tools to check model satisfiability have been developed but to our knowledge, none of them yet supports the analysis of OCL expressions including operations on Strings in general terms. As, in contrast, many industrial models do contain such operations, there is evidently a gap.
There has been much research on formal reasoning on strings in general, but so far the results could not be included into model finding approaches. For model finding, string reasoning only contributes a sub-problem, therefore, a string reasoning approach for model finding should not add up front too much computational complexity to the global model finding problem. We present such a lightweight approach based on constraint satisfaction problems and constraint rewriting. Our approach efficiently solves several common kinds of string constraints and it is integrated into the EMFtoCSP model finder.

**Keywords:** OCL, String data type, Model Finding

## 1 Introduction

Model-driven Engineering (MDE) is a popular approach to the development of software based on the use of models as primary artifacts. To precisely describe the conceptual structure of a model, the Object Constraint Language (OCL) [23] has been widely accepted as a de-facto standard. In a nutshell, OCL allows to express model constraints using a first-order logic like language for objects.

Naturally, the increased precision comes along with an increased complexity of the models. This raises the need for systematic approaches to model validation, model verification, and model-based testing. Model finding (also called model instantiation) is an important problem in this context. It considers the question if a given model (including constraints) is satisfiable, and if it is satisfiable, to identify one instance of the model. While in model verification, model finders are typically used to show unsatisfiability when reasoning about implications between different constraints, the focus in model-based testing is typically on finding satisfying instances, which can be used to test a system which is based on the model.

* This work has been partly funded by the European Project CESAR

The community has developed several approaches and tools for automated model finding for OCL-annotated models. To deal with the computational complexity of the problem (which is undecidable in general), most of them are based on some underlying formalism for which sophisticated decision procedures and tools exist, such as first-order logic and satisfiability modulo theory (SMT), relational logic, propositional logic, and constraint satisfaction problems (CSP).

While these approaches cover an extensive subset of OCL, to our knowledge none of them supports the String data type and its OCL operations. The primitive data types typically supported are Integer and Boolean. Given that, on the contrary, several 'real life' models actually do contain constraints over strings, there is evidently a gap that needs to be addressed. However, we need to be aware that, when compared to models that only contain Integer primitive values, adding strings to the subject of reasoning introduces another level of complexity.

There are several works that address string reasoning, some focused on checking grammar satisfiability as a stand-alone problem, others on path analysis for string-manipulating programs. However, in the context of model finding and, in particular, model-based testing, string reasoning only contributes a sub-problem to the overall search problem. Therefore, there is a trade-off between the completeness of the string reasoning procedures and the overall model finding performance.

In this paper we present a lightweight approach to integrate constraints over bounded strings into model finding using constraint rewriting and Boolean and integer constraints. It is a two-step approach that first reasons about the lengths of the strings, then infers constraints on the individual elements of the strings (their character variables). In general, our approach can be implemented in any off-the-shelf solver that supports reasoning about linear constraints. We included it in the OCL model finder EMFtoCSP [15], the successor of UMLtoCSP [8], which is based on the ECL$^{\text{i}}$PS$^{\text{e}}$constraint logic programming environment.

For many common constraint constellations, our approach is scalable and shows good performance, and we claim that it is suited for several practical applications that do not pose hard, non-tractable string constraints. We provide experimental results that show that models with more than a thousand strings can be found within seconds.

*Paper Organization.* In Sect. 2, we first discuss the state of the art for the topic. Section 3 then formally presents our approach. In Sect. 4 we discuss its limits and scalability, and present experimental results of our implementation of the approach in the tool EMFtoCSP. Section 5 concludes our contribution and identifies future work.

## 2 State of the Art

In this section, we describe the state of the art in model finding for OCL-annotated models in general and its translation into constraint satisfaction problems, and we put our work in the context of general formal reasoning techniques for strings.

### 2.1 Model Finding

The model finding (or model instantiation) problem for models with constraints can be defined as follows: Let $\mathcal{M}$ be a model defining structural elements such as classes,

associations, and attributes. Let $C_{\mathcal{M}}$ be a set of constraints over $\mathcal{M}$. Let $I(\mathcal{M})$ denote the (possibly infinite) set of models (instances) of $\mathcal{M}$. The pair $(\mathcal{M}, C_{\mathcal{M}})$ is called satisfiable iff there exists a least one instance $\sigma \in I(\mathcal{M})$ such that $\sigma \models \hat{c_i}$ holds for each $c_i \in C_{\mathcal{M}}$, where we assume $\hat{c_i}$ to be a logical representation of $c_i$ that can be evaluated on $\sigma$. A model that is not satisfiable is called unsatisfiable. If a model is satisfiable, one is typically interested in a satisfying instance of it, too.

Model finding is important in several tasks within the model-driven approach. It is required in the validation and testing of systems based on the model (to systematically specify test cases), validation and testing of model transformations [3, 24], as well in the validation and verification of the model itself. Model finding has also been applied to verify the correctness of model transformations as transformation models (e.g., [7, 6]).

The community has developed several approaches and tools for automated model finding for models with OCL constraints. To deal with the computational complexity of the problem (which is undecidable for OCL in general), most of them are based on some underlying formalism for which sophisticated decision procedures and tools exists, such as, first-order logic and SMT [10], relational logic [2, 20], propositional logic [26], genetic algorithms [1], graph grammars [28] and constraint satisfaction problems [8]. All of these approaches support a more or less extensive subset of OCL (e.g., including quantifiers and collections), but, to our knowledge, the support of the String data type is very limited. [13] supports strings, but requires the user to specify an explicit procedure for the construction of potential strings and is not a black box model finding approach. [1] considers strings, but the approach, which is based on genetic algorithms, is focused on test case generation only and is (intentionally) not exhaustive. A promising approach is our view is [19], where the authors propose an encoding of OCL strings for the relational logic solver Kodkod. However, to our knowledge, this encoding still requires a constant maximum number of boolean variables, equal for all string variables (even when this length is only required for a single string).

We want to emphasize that most of the underlying formalism and solvers employed in the aforementioned approaches do support bounded bitvectors or sequences. Thus, theoretically, strings can be translated straightforwardly into these formalisms. However, unless considerably small upper bounds are imposed on string lengths, this quickly leads to extreme search spaces, with considerable effects on the runtime of the solvers and the decidability of the search problem in practice. Furthermore, this also prevents them from taking into account the String semantics on the symbolic level to reduce the search space up front.

### 2.2 Model Finding as a CSP

A constraint satisfaction problem (CSP) can be defined by a tuple

$$(V, C)$$

where $V = \{v_1 \in D_1, \ldots, v_m \in D_m\}$ is a set of variables $v_i$ and their domains $D_i$, and $C$ is a set of constraints over $V$. Where clear from the context, we will omit variable domains in the following. A constraint has the form $P(x_1, \ldots, x_n)$ where $P$ denotes an

$n$-ary predicate on $x_1, \ldots, x_n \in V$. An assignment $\beta$ of values to variables *satisfies* a constraint $P(x_1, \ldots, x_n)$ if $P\big(\beta(x_1), \ldots, \beta(x_n)\big)$ is true. If $\beta$ satisfies all constraints in $C$, it is a *solution* to $P$. We say a CSP is *consistent* (or feasible) if it has a solution, and *inconsistent* (or infeasible) if it does not.

Typical constraints employed in CSPs include a combination of arithmetic expressions, mathematical comparison operators and logical operators. Common techniques for the resolution of CSPs are based on backtracking and constraint propagation. CSPs can be represented in constraint logic programming, which embeds constraints into a logic program.

Eventually, the notion of the model finding problem in MDE is similar to the notion of a CSP, but it is based on a more complex structure (the variables are objects, links, etc.). In [8] a translation of the model finding problem for OCL-annotated models into a CSP is described. Given a $\mathcal{M}$ and a set of OCL constraints $C_{\mathcal{M}}$, the approach defines how to infer a CSP $P$ that is consistent iff. $\langle \mathcal{M}, C_{\mathcal{M}} \rangle$ is satisfiable. The solutions to $P$ correspond to instances of $\mathcal{M}$. Technically, the derived CSP $P$ consists of two sub-problems

$$P_{\text{structure}} = (\ Cardinalities, CardinalityConstraints\ )$$

and

$$P_{\text{global}} = (\ Instance, InstanceConstraints\ )$$

where the solutions of the first sub-problem are (potentially) valid sizes for the sets of objects and links. The variables of the second sub-problems include lists of objects and links. Iterating the solutions to the first sub-problem (by backtracking), these lists are instantiated (i.e., a length is assigned to them) in the second sub-problem.

In a nutshell, two kinds of constraints are employed in the second sub-problem. The first kind includes constraints over Boolean and integer arithmetic, for which constraint propagation is available in most constraint programming languages. The second kind includes specific constraints to represent, for example, navigation operations. The derivation of these constraints can be implemented, for example, using *suspended goals*.

We have implemented our String reasoning approach in EMFtoCSP [11], which is the successor of UMLtoCSP [8] and supports EMF metamodels as well as UML models. Other approaches that also express model finding in terms of constraint programming include [22, 21, 9].

### 2.3 Formal String Reasoning

Reasoning on strings has been performed in various formalisms, both for bounded and unbounded strings. Solvers for Satisfiability Modulo Theories (SMT) commonly support theories that can be used to represent strings, such as arrays and bit-vectors. Also, as a string is only a specific case of a list, any theory of lists can be applied, too. Recently, a working group for the development of a theory for strings and regular languages has been formed [4].

In addition to the theory-based approaches, several approaches reason about string using finite automata and regular expression, e.g., [18, 17, 14, 27]. [16] incorporates

regular languages into a CSP. In [5] the authors perform path analysis for String-manipulating programs using SMT, employing a two step approach similar to ours, determining an approximation of string lengths in the first step first.

In comparison to these approaches, our approach is less complex in the sense that it performs symbolic reasoning only over the lengths of strings. Due to its two-step nature, it is not suited to solve real hard (NP-hard), non-tractable string problems in reasonable time. However, it efficiently solves several less hard string problems with minimal overhead, which makes it suitable for embedding into a more general model finding procedure.

## 3 Lightweight String Reasoning for OCL

We now present our approach for solving OCL constraints that use the OCL data type String and its operations. Our approach can be easily integrated into existing approaches to satisfiability checking (model finding) of models and OCL constraints such as the ones presented in Sect. 2.

At its core, we formulate the problem as a CSP. We provide five constraint predicates for strings that can be used to translate OCL constraints containing string expressions. These five predicates correspond directly to five core operations of the OCL data type String, namely equality (=), size, substring, concat, and indexOf. Our five constraints predicates are resolved into constraints on integers and Booleans in two rewriting steps (described as constraint handling rules, which we introduce below). Fig. 1 depicts this process and will be explained in the following.

We assume an OCL model and its constraints have been translated into a CSP $P = (V, C)$ (e.g., as described in [8], c.f. Sect. 2.2). $V$ includes variables of type String and $C$ includes constraints over these variables. In the first step, we infer additional constraints on the lengths of the individual string variables. The second step then translates the string constraints into constraints on the elements of the strings (i.e., their characters). The result of this two-step process is a CSP that can be solved using off-the-shelf solvers for linear constraints.

More formally, we first employ a rewriting operation $C \rightsquigarrow C_{\text{length}}$ (described in Sect. 3.4) that extends $C$ by additional constraints over the lengths of all string variables in $V$. Then, we require that a solution to the *length sub-problem* is chosen, which introduces a potential choice point in a backtracking search process. We refer to the set of constraints in which the length variables are bound to fixed lengths as $C_{\text{length},i} \rightsquigarrow C_{\text{inst},i}$, where $i$ shall number the different solutions of the length sub-problem.

The second rewriting operation $C_{\text{length},i} \rightsquigarrow C_{\text{inst},i}$ then (a) unifies all symbolic string variables $s_i$ with lists of individual element variables $\langle s_{i,1}, s_{i,2}, \ldots \rangle$, and (b) rewrites the semantics of the string constraints into into Boolean and integer constraints over the element variables. The solutions to $(V, C_{\text{inst},i})$ are solutions to $P$. If $C_{\text{inst},i}$ has no solution, the next valid solution $i + 1$ to the length sub-problem must be selected. If there is not further solution to the length subproblem, $P$ is *inconsistent*.

The remaining section first describes the five OCL operations supported more precisely in Sect. 3.1. We provide our five String constraint predicates in Sect. 3.2. Sect. 3.3 gives a short primer on the constraint handling rules formalism, which we employ to

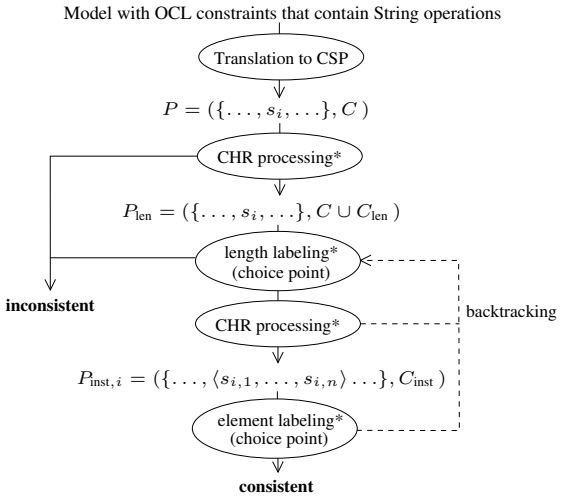


**Fig. 1.** The decision procedure for CSP with String constraints, including CHR processing. All processing steps (indicated with a star) are supposed to perform constraint propagation on linear and Boolean constraints.

define the two rewriting steps in Sections 3.4 and 3.5. We provide a complete derivation example in Sect. 3.6.

### 3.1 Considered OCL String Operations

The OCL specification [23] defines several operations for the data type String. In this work we consider a subset of five important core operations: the length of a string ($s_1$.concat($s_2$)), the concatenation of two strings ($s_1$.concat($s_2$) resp. $s_1 + s_2$), the indexed substring of a string ($s_1$.substring($i, j$)), and the containment of a string in another one ($s_1$.indexOf($s_2$)).

In accordance with the OCL specification, we assume the semantics of the core operations as follows: Let $\mathcal{A}$ be the set of characters, and $\mathcal{A}^*$ be the set of all strings (sequences of characters). The semantics of this core can be described by a set of interpretation functions $I(\text{size}) : \mathcal{A}^* \to \mathbb{N}$, $I(\text{concat}) : \mathcal{A}^* \times \mathcal{A}^* \to \mathcal{A}^*$, $I(=) : \mathcal{A}^* \times \mathcal{A}^* \to \mathbb{B}$, $I(\text{substring}) : \mathcal{A}^* \times \mathbb{N} \times \mathbb{N} \to \mathcal{A}^*$, and $I(\text{indexOf}) : \mathcal{A}^* \times \mathcal{A}^* \to \mathbb{N}$, as follows. $I(\text{size})(s)$ is defined to be the length of the sequence $s$ (and can be 0). $I(=)(s_1, s_2)$ is $\mathit{true}$ iff $s_1$ and $s_2$ have the same lengths and are equal element-wise, and $\mathit{false}$ otherwise. $I(\text{concat})(s_1, s_2)$ is $s_1 \circ s_2$ (concatenation of sequences). $I(\text{substring})(s, i, j)$ is the subsequence from $i$ to $j$, and is only defined for $1 \leq i \leq j \leq |s|$ according to the OCL specification. $I(\text{indexOf})(s_1, s_2)$ is the index of the first occurrence of $s_2$ in $s_1$ when $s_1$ is non-empty and 0 otherwise. In OCL, no string is a substring of the empty string (not even the empty string).

### 3.2 String Constraints

We define five string constraint predicates that are sufficient to express OCL constraints that use the core operations on strings as a CSP. They correspond directly to the operations (cf. Sect. 3.1). Examples for the encoding follow after this section.

Let $s$, $s_1, \ldots s_3$ denote string variables, and $l$, $i$ ,and $j$ denote integers. The five constraints are $\mathrm{len}(s, l)$ – the length of $s$ is $l$, $\mathrm{eq}(s_1, s_2, b)$ – $s_1$ and $s_2$ are equal iff. b is $\mathrm{true}$, $\mathrm{con}(s_1, s_2, s_3)$ – $s_3$ is the concatenation of $s_1$ and $s_2$, $\mathrm{sub}(s_1, i, j, s_2)$ – $s_2$ is the substring of $s_1$ from $i$ to $j$, and $\mathrm{idx}(s_1, s_2, i)$ – the number $i$ is either the first position at which $s_2$ occurs in $s_1$, or 0 if $s_2$ does not occur in $s_1$.

Notice that the string equality constraint has a *reified* form, which is necessary to deal with string equality in the linear and propositional reasoning. For example, the OCL constraint $s_1 = s_2 \,\mathrm{implies}\, s_1 = s_3$ would be expressed as

$$P = \big(\{s_1, s_2, s_3\}, \{(b_1 \rightarrow b_2), \mathrm{eq}(s_1, s_2, b_1), \mathrm{eq}(s_1, s_3, b_2)\}\big)$$

where $\rightarrow$ is assumed as a predefined constraint over two Booleans.

### 3.3 Constraint Handling Rules

To describe the rewriting operations on constraints, we employ Constraint Handling Rules (CHR), which is a well-known formalism and has several implementations available. However, our rewriting rules can also be implemented in other ways easily. As we make only very limited use of the formalism, we only introduce a simplified form here. For a thorough presentation of the formalism we refer to [12] and [25]. In our restricted context, a constraint handling rule has one of the three syntactic forms.

$$\begin{aligned} \mathit{rulename} \mathbin{@} c_1, \ldots, c_m &\iff c'_1, \ldots c'_n \\ \mathit{rulename} \mathbin{@} c_1, \ldots, c_m &\implies c'_1, \ldots c'_n \\ \mathit{rulename} \mathbin{@} c_1, \ldots, c_k \setminus c_{k+1}, \ldots, c_m &\implies c'_1, \ldots c'_n \end{aligned}$$

where $c_i$ and $c'_i$ are constraints that typically share some variables. The common semantics of these rules is that they match a pattern of constraints $c_1, \ldots, c_m$ in the constraint store (which, in our case, is the set of constraints in the CSP). The constraints in the pattern are related by their common variables, for example, as in the pattern $c_1(s, i), c_2(s, j)$. The first kind of rules above is called a *simplification* rule. It removes the matched constraints $c_1, \ldots, c_m$ from the constraint store and replaces them by new constraints $c'_1, \ldots c'_n$. The second kind is called a *propagation* rule, which also adds $c'_1, \ldots c'_n$ to the imposed constraints, but also keeps $c_1, \ldots, c_m$ in the store. The third kind is called a *simpagation* rule and is a mixture of the former two. It keeps $c_1, \ldots, c_k$, but replaces $c_{k+1}, \ldots, c_m$ by $c'_1, \ldots c'_n$. The execution of a set of CHR rules is terminated when no more rules can be applied. For propagation rules, the CHR environment ensures that such rules are applied only once per match of their pattern.

### 3.4 First Rewriting Step: The Length Sub-Problem

We now define the rules that infer linear additional constraints over the lengths of the string variables. This constitutes the first rewriting step in our approach (cf. Fig. 1).

**Definition 1.** *Length Inference Rules*

$$\begin{aligned}
\textit{len-dom} \ @\ \mathrm{len}(s,l) &\implies 0 \le l \le \mathit{MaxLen} \\
\textit{len-one} \ @\ \mathrm{len}(s,l_1) \setminus \mathrm{len}(s,l_2) &\iff l_1 = l_2 \\
\textit{eq-len} \ @\ \mathrm{eq}(s_1,s_2,r), \mathrm{len}(s_1,l_1), \mathrm{len}(s_2,l_2) &\implies r \rightarrow l_1 = l_2 \\
\textit{con-len} \ @\ \mathrm{con}(s_1,s_2,s_3), \mathrm{len}(s_1,l_1), \mathrm{len}(s_2,l_2), \mathrm{len}(s_3,l_3) &\implies l_3 = l_1 + l_2 \\
\textit{sub-len} \ @\ \mathrm{sub}(s_1,i,j,s_2), \mathrm{len}(s_1,l_1), \mathrm{len}(s_2,l_2) &\implies \begin{array}{l} l_2 = (j-i+1), \\ 1 \le i \le j \le l_1 \end{array} \\
\textit{idx-len} \ @\ \mathrm{idx}(s_1,s_2,i), \mathrm{len}(s_1,l_1), \mathrm{len}(s_2,l_2) &\implies \begin{array}{l} i \neq 0 \rightarrow l_1 \ge l_2, \\ l_1 = 0 \rightarrow i = 0 \end{array} \\
\textit{eq-refl} \ @\ \mathrm{eq}(s,s,b) &\iff b \leftrightarrow \mathrm{true} \\
\textit{eq-one} \ @\ \mathrm{eq}(s_1,s_2,b_1) \setminus \mathrm{eq}(s_1,s_2,b_2) &\implies b_1 \leftrightarrow b_2.
\end{aligned}$$

The propagation rule len-dom constrains all strings to be finite, using a maximum length *MaxLen* that can be any positive number. For example, given $\mathit{MaxLen} = 100$, the CSP $(\{s\}, \mathrm{len}(s,l)\})$ would be rewritten to $(\{s\}, \mathrm{len}(s,l), 0 \le l \le 100\})$. The simpagation rule len-one removes multiple lengths constraints for the same string and replaces them by linear constraints over the length variables. For example, $P = \big(\{s\}, \mathrm{len}(s,l_1), \mathrm{len}(s,4), l_1 \le 3\}\big)$ would be rewritten to $\big(\{s\}, \mathrm{len}(s,l_1), l_1 \le 3, l_1 = 4\big)$ – which a linear constraint solver would detect as unsatisfiable for any $l_1$. The propagation rule eq-len poses conditional equality. The rules con-len, sub-len, and idx-len generate the expected constraints in the same manner. Finally, the simplification rules eq-refl and eq-one include reflexivity and *tertium non datur* into the length inference.

Please note that we included the last two rules, eq-refl and eq-one for practical reasons only, as a performance optimization (these rules add only little overhead in terms of constraint processing). They are theoretically not required, because equality of strings will be translated into pair-wise equality of their elements (see below). On the contrary, we do not include transitivity (and neither symmetry) here, in order to keep the approach lightweight, as transitivity can lead to an exponential number of equality constraints. Theoretically, $\mathrm{eq}(s_1, s_2, \mathrm{true})$ is of course transitive (and $\mathrm{eq}(s_1, s_2, b)$ is symmetric), because the equality on the element variables has exactly these properties. We found, however, that turning these properties into rules produces too much overhead for the kind of (lightweight) string problems we consider.

When no more of the rules in Def. 1 can be applied (i.e., the execution of these rules has terminated), a ground assignment of all length variables has to be selected. In general, this introduces a choice point. Consider, for example, if we derived $P = \big(\{s_1, s_2\}, \{\mathrm{len}(s_1,l_1), \mathrm{len}(s_2,l_2), 1 \le l_1 \le 4, 1 \le l_2 \le 4, s_1 = s_2 - 1\}\big)$, the choices for $l_1, l_2$ would be $1, 2$ and $2, 3$. Assuming we select $1, 2$ first, we would proceed with the CSP $P = \big(\{s_1, s_2\}, \mathrm{len}(s_1, 1), \mathrm{len}(s_2, 2)\big)$

### 3.5 Second Rewriting Step: Resolve String Constraints To Element Constraints

Given that a solution to the length sub-problem has been selected (i.e., all length variables have ground values), the following rule unifies all string variables with lists of

element variables, where each element variable is constrained to be in the range of the alphabet $\mathcal{A}$, for which charnums is assumed to assign a corresponding set of integers from 1 to $|\mathcal{A}|$.

**Definition 2.** *Structural Instantiation Rule*

$$\begin{aligned} \textit{len-inst} \ @ \ & \mathrm{len}(s, n) \iff \\ & s = \langle x_1, \ldots, x_n \rangle, x_1 \in \textit{charnums}(\mathcal{A}), \ldots, x_n \in \textit{charnums}(\mathcal{A}) \end{aligned}$$

After all string variables have been instantiated using rule len-inst, all string constraints in $C$ are finally replaced ($\Leftrightarrow$) by linear and Boolean constraints on the individual element variables using the following rewrite rules.

**Definition 3.** *Linear Representation Rules*

$$\begin{aligned} \textit{eq-inst} \ @ \ & \mathrm{eq}(\langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_n \rangle, r) \iff \\ & r \leftrightarrow \left( \textstyle\bigwedge_{1 \le i \le n} x_i = y_i \right) \end{aligned}$$

$$\begin{aligned} \textit{con-inst} \ @ \ & \mathrm{con}(\langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_n \rangle, \langle z_1, \ldots, z_{m+n} \rangle) \iff \\ & \left( \left( \forall_{1 \le i \le m} \ x_i = z_i \right), \ \left( \forall_{1 \le j \le n} \ y_j = z_{m+j} \right) \right) \end{aligned}$$

$$\begin{aligned} \textit{sub-inst} \ @ \ & \mathrm{sub}(\langle x_1, \ldots, x_m \rangle, i, j, \langle y_1, \ldots, y_n \rangle) \iff \\ & \forall_{0 \le l \le (n-m)} \left( i = l + 1 \rightarrow \left( \textstyle\bigwedge_{1 \le k \le n} x_{k+l} = y_k \right) \right) \end{aligned}$$

$$\begin{aligned} \textit{idx-inst} \ @ \ & \mathrm{idx}(\langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_n \rangle, i) \iff \\ & \forall_{0 \le l \le (n-m)} \Big( \ r' \leftrightarrow \left( \textstyle\bigwedge_{1 \le k \le n} \ x_{k+l} = y_k \right), \\ & \qquad\qquad r' \rightarrow p_l = l + 1, \neg r' \rightarrow p_l = 0 \Big), \\ & i = \textit{min}^*(p_0, \ldots, p_{(n-m)}) \end{aligned}$$

*where $\forall$ is used to express a set constraints and $b \leftrightarrow \bigwedge(\ldots)$ is used to represent that $b$ is constrained to be the result of the conjunction of a set of Boolean values. $r'$ and $p_l$ are fresh variables and min$^*$ is the usual minimum function on natural numbers with the exception that $0$ is regarded as the largest number.*

The rule eq-inst poses one Boolean constraint that $r$ is equal to the Boolean value of the conjunction of the element-wise equality of both strings. The rule con-inst poses one equality constraint for each element of the concatenated string. The rule sub-inst poses one constraint for each possible offset $l$ of $y$ in $x$. Please notice that we can safely assume $m$ and $n$ to be ground values, whereas $i$ and $j$ can be variables. The rule does not pose further constraints on $j$, because $j$ has already been expressed dependent on $i$ by rule sub-len before (see Def. 1). Rule idx-inst is of similar nature, only that it introduces one integer variable $p_l$ per possible offset of $y$ in $x$, which is constrained to be either the position of $y$ in $x$ or 0. The result of $i$ is then expressed by the (modified) minimum of these values. This minimum can be rewritten syntactically into basic linear constraints.

After the rules of Def. 3 have been applied, the resulting CSP contains only relational and arithmetic constraints on Boolean and integer values. It can be solved using off-the-shelf solvers.

### 3.6 Derivation Example

To illustrate the definitions so far, we now provide a complete derivation example that shows (i) how an OCL constraint is expressed in terms of our string constraints, (ii) how the length inference takes place, and (iii) how the constraints are finally resolved for a given length assignment. We consider the following OCL constraint:

$$s_1 = (s_2 + s_3).\,\mathrm{substring}(2,3)\,\mathrm{and}(s_2 + s_3).\,\mathrm{size}() < 10$$

It can be represented as a CSP in a straight-forward manner using the constraints previously introduced. Because constraints predicates cannot be nested, the CSP requires two additional string variables $s_4$, $s_5$ to express the results of the subexpressions $s_2 + s_3$ and $(s_2 + s_3).\,\mathrm{substring}(2,3)$. This means that, while we are eventually interested in three strings, $s_1$,$s_2$, and $s_3$, we have to regard five strings in the CSP. Note that a $\mathrm{len}$ constraint with a free length variable is included for each string. Let $MaxLen = 1000$. We assume the built-in reified Boolean resp. linear constraints $\wedge(x, y, b)$ and $<(x, y, b)$, for which the third argument is the truth value of $x \wedge y$ resp. $x < y$. The resulting CSP is

$$\begin{aligned}\big(\{s_1,\ldots,s_5\},\{\ &\mathrm{len}(s_1,l_1),\mathrm{len}(s_2,l_2),\mathrm{len}(s_3,l_3),\mathrm{len}(s_4,l_4),\\ &\mathrm{len}(s_5,l_5),\wedge(b_1,b_2,\mathrm{true}),\mathrm{eq}(s_1,s_5,b_1),\\ &\mathrm{con}(s_2,s_3,s_4),\mathrm{sub}(s_4,2,3,s_5),<(l_4,10,b_2)\}\big)\end{aligned} \tag{1}$$

We assume that propagation on Boolean predicates will unify $b_1 = \mathrm{true}$ and $b_2 = \mathrm{true}$ from $\wedge(b_1, b_2, \mathrm{true})$. The rewriting rules for lengths apply as follows: len-dom infers a range constraint for each string length, eq-len infers $l_1 = l_5$, con-len infers a linear constraint $l_4 = l_2 + l_3$, sub-len infers the linear constraints $l_5 = 2$ and $1 \le i \le j \le l_4$. Assuming that the linear constraints propagate their bounds, the simplified resulting CSP is

$$\begin{aligned}\big(\{s_1,\ldots,s_5\},\{\ &\mathrm{len}(s_1,2),\mathrm{len}(s_2,l_2),\mathrm{len}(s_3,l_3),\mathrm{len}(s_4,l_4),\\ &\mathrm{len}(s_5,2),\mathrm{eq}(s_1,s_5,\mathrm{true}),\mathrm{con}(s_2,s_3,s_4),\\ &\mathrm{sub}(s_4,2,3,s_5),0 \le l_2 < 10,0 \le l_3 < 10,\\ &l_4 = l_2 + l_3, 3 \le l_4 < 10\}\big).\end{aligned}$$

with $l_1$ and $l_5$ being removed from the variables (as $l_1 = l_5 = 2$), At this point, no more rules are applicable until we select a solution to the length sub-problem

$$\big(\{l_2,\ldots l_4\},\le l_2 \le 10, 0 \le l_3 < 10, l_4 = l_2 + l_3, 3 \le l_4 < 10\}\big) \tag{2}$$

We assume the assignment $\{l_1 \mapsto 2, l_2 \mapsto 2, l_3 \mapsto 3, l_4 \mapsto 3, l_5 \mapsto 2\}$ is chosen and apply rule len-inst. The CSP to solve is now

$$\begin{aligned}\big(\{\ &s_{1,1},s_{1,2},s_{2,1},s_{2,2},s_{3,1},s_{4,1},s_{4,2},s_{4,3},s_{5,1},s_{5,2}\},\\ \{\ &\mathrm{eq}(s_1,s_5,\mathrm{true}),\mathrm{con}(s_2,s_3,s_4),\mathrm{sub}(s_4,1,3,s_5),\\ &\mathrm{sub}(\langle s_{4,1},s_{4,2}\rangle,1,3,\langle s_{5,1},s_{5,2}\rangle),\\ &s_{1,1},s_{1,2},s_{2,1},s_{2,2},s_{3,1},s_{4,1},s_{4,2},s_{4,3},s_{5,1},s_{5,2} \in \mathsf{charnums}|\mathcal{A}|\}\big)\end{aligned}$$

with $s_1 = \langle s_{1,1}, s_{1,2} \rangle$, $s_2 = \langle s_{2,1}, s_{2,2} \rangle$, $s_3 = \langle s_{3,1} \rangle$, $\langle s_{4,1}, s_{4,2}, s_{4,3} \rangle$, and $\langle s_{5,1}, s_{5,2} \rangle$.

For this example, the resolving of the rules eq-inst, con-inst, and sub-inst finally unifies several variables, leaving a CSP that is *solved* (i.e., a CSP for which every assignment of character numbers to the element variables is a solution).

$$\big(\{s_{1,1}, s_{1,2}, s_{2,1}, s_{2,2}, s_{3,1}\}, \{s_{1,1}, s_{1,2}, s_{2,1}, s_{2,2}, s_{3,1} \in \text{charnums}|\mathcal{A}|\}\big)$$

All $|\mathcal{A}|^5$ solutions to this CSP are solutions to the original CSP (1) under the assignment of $\{l_2 \mapsto 2, l_3 \mapsto 1, l_4 \mapsto 3)\}$ to the length subproblem (2) with $(s_1, \langle s_{1,1}, s_{1,2} \rangle)$, $(s_2 = \langle s_{2,1}, s_{2,2} \rangle$, $s_3 = \langle s_{3,1} \rangle$, $s_4 = \langle s_{2,1}, s_{1,1}, s_{1,2} \rangle$, $s_5 = \langle s_{1,1}, s_{1,2} \rangle$. Naturally, in other cases not all assignments to the final CSP are solutions to the problems. In general, *search* must be performed for a solution. The impact of this search on the scalability of our approach is discussed in the next section.

## 4 Limits and Scalability

The rewriting rules presented in the previous section do not constitute a self-contained decision procedure. They have to be combined with a solver that is capable to reason about propositional logic and bounded integer constraints, for which decision procedures are available in various solvers. The presented constraint handling rules are terminating and confluent. This means that, in theory, every CSP on strings with bounded lengths can be solved using our approach if it can be expressed using the provided string constraint predicates and other decidable constraints. In this section, we first provide some experimental results that we gathered using our implementation of the approach, then we discuss scalability aspects in more generality.

### 4.1 Experimental Results

As mentioned before, we have implemented the described reasoning approach in our model finder EMFtoCSP, using the constraint handling rules support of the ECL$^i$PS$^e$solver. We now employ the model depicted class diagram in Fig. 2 and the following OCL constraints to illustrate performance and scalability aspects of our approach and implementation.

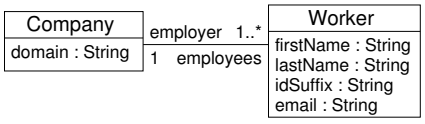


**Fig. 2.** Example Class Diagram annotated with OCL Constraints for its Strings

The example contains (conditional) string equality, concatenation, and and containment constraints. Due to the existential and universal quantifiers in EmailsUnique (all workers within a company must have the distinct email addresses) and OneSame (at

least two workers must share first and last name), a quadratic number of of conditional equality constraints is posed. Both invariants are considerably hard in terms of computational complexity, which is why we have added them to our experiment. The constraint EmailStructured determines the structure of the email attribute in terms of concatenation.

```
context Company inv EmailsUnique:
   worker->forAll(w1,w2 | w1.email = w2.email implies w1 = w2 )
context Company inv OneSame:
  worker->exists(e1,e2 | e1 <> e2 and
                 e1.firstName = e2.firstName and
                 e1.lastName = e2.lastName )
context Worker inv EmailStructured:
  email = firstName + '.' +  lastName + '.' +
          idSuffix + '@' + employer.domain
context Worker inv NoAt:
  firstName.indexOf('@') = 0 and
  lastName.indexOf('@') = 0
```

EMFtoCSP translates the model and its constraints into a CSP as described in Sect. 2.2. In our extended version, the OCL constraints are translated using the string constraints introduced in the previous section. Recall that the original approach consists of two sub-problems *Structure* and *Global*. In the version with string support, the length inference is included into the global sub-problem (i.e., it adds further constraints to this sub-problem). The element labeling then constitutes a new, third sub-problem *Strings*.

Table 1 show the runtimes and linear constraint propagations performed by ECL$^i$PS$^e$ for different instance sizes of the above example. The tests were conducted on a typical office 2.2Ghz laptop running ECL$^i$PS$^e$ 6.0 on Windows 7. All tests used a maximum string lengths of 1,000.

The table differentiates the runtimes and propagation counts for the CHR processing and the final labeling (i.e., the assignment of ground character values to the elements of all strings). For each test cases we constrained the workers to be evenly distributed among the companies. The table illustrates several aspects. First, we can see that the actual character labeling does not consume a significant amount of time once the constraint handling rules have been processed and the linear constraints have been posted.

Furthermore, we can observe that the runtimes for the cases where few companies have many workers consume the most processing time. This is due to the quadratic number of equality constraints that results from the OCL invariants EmailsUnique and OneSame. For the second row in Table 1 more than 30,000 conditional equality constraints are posed in the *Global* sub-problem. If the number of workers per company is less high, EMFtoCSP scales better, for example, when the ratio is 1:10, as in the third row.

### 4.2 General Discussion

In general, we can distinguish three categories for the solving of the CSP, where the distinction between the second and third case depends on the capabilities of he solver employed. In a nutshell, our approach works very efficiently in the first category, and

less efficient in the others. We have conducted several experiments using our implementation of the approach in EMFtoCSP and the ECL$^i$PS$^e$ solver, and found that several typical patterns of constraints encountered in models fall into the first case (and so does the previously presented example).

1. The *optimal case*: No backtracking is required, every valid length assignment yields a *solved* CSP on the string elements after applying our rewriting rules and performing constraint propagation. In this case, even very high numbers for the maximum string length (e.g., 1000) can be chosen.
2. In the *length search case*, not every valid length assignment yields a consistent CSP on the string elements, but the inconsistency of a chosen length assignment is detected by the solver without actually labeling the elements. A simple example for this case is given by the OCL constraint `s.indexOf('@') = 0 and s.indexOf('@') <> 0`, whose unsatisfiability is not recognized before instantiating the $s$ to its element variables[1]. The ECL$^i$PS$^e$ solver, for example, however detects that the resulting CSP (the third one in Fig. 1) is consistent without backtracking through the possible assignments of values to the element variables, leaving $MaxLen$ choices for the solver before reporting inconsistency. For constraints that fall into this case, the maximum string length must be set to a reasonable small value for practical applications.
3. The *labeling trap*: In this case, the inconsistency is not detected before actually labeling the element values. To ease the labeling trap in practice, the search procedure can be split to perform a two-pass run, where the first pass tries at most one assignment to the element variables for each solution to the length problem. In the second pass, the element labeling is repeated without restrictions. This tweak to the search space traversal helps to circumnavigate the labeling trap for insufficient string lengths.

However, for most constraint patterns typically encountered in 'real live' situations, the labeling trap is not a problem. In fact, as stated before, several common constraint patterns fall even into the first category. Summarizing, we state that our approach is perfectly suited to efficiently handle lightweight string problems that have many solutions, while it is not suited to solve non-tractable string problems (which, in general, are NP-hard), that only have few solutions, and for which the employed solver runs into the labeling trap.

## 5 Conclusion

In the previous sections, we have presented an approach that translates OCL String constraints into a constraint satisfaction problem that can be solved using off-the-shelf solvers, and which can be integrated easily into existing model finding approaches and tools for OCL without adding too much overhead to the underlying decision procedures. Our approach is lightweight in the sense that it efficiently finds solutions for many common OCL constraint constellations and it is suited to handle models with thousands of

[1] We assume the constraint is translated straightforwardly using two separate idx constraints.

| Instance | | | CHR Processing | | Character Labeling | |
|---|---|---|---|---|---|---|
| companies | workers | strings | cpu time | propagations | cpu time | propagations |
| 1 | 10 | 41 | 0.1s | 3,112 | $\leq$ 0.1s | 3,360 |
| 1 | 100 | 401 | 10.1s | 220,192 | 1.1s | 963,600 |
| 10 | 100 | 410 | 0.7s | 31,120 | $\leq$ 0.1s | 33,600 |
| 50 | 100 | 450 | 0.4s | 14,000 | $\leq$ 0.1s | 3,200 |
| 150 | 300 | 1,350 | 2.3s | 42,000 | $\leq$ 0.1s | 9,600 |
| 10 | 300 | 1,210 | 11.7s | 219,520 | 0.6s | 440,800 |

**Table 1.** Experimental Results

strings. Due to its two-step nature, we can efficiently handle strings of potentially long lengths, which otherwise would lead to search space explosion when directly encoding all strings as bitvectors of the maximum length. We therefore claim that our approach is suited for various practical ('real world') applications. It is, however, not suited to tackle hard, non-tractable string constraints that only have few solutions. These must be addressed either by one-step bitblasting approaches or by formal regular language reasoning and theorem provers that can handle an appropriate theory.

So far we considered an important subset of OCL string operations. We expect that the remaining operations (e.g., toLowerCase, at, characters, $<$) can be encoded in the same manner. To our knowledge, we are the first ones that integrate reasoning on String constraints into model finding for OCL-annotated models. We have implemented our results into the EMFtoCSP (formerly UMLtoCSP) tool and provided some experimental results. While we used constraint handling rules as a formalism to define our constraint rewriting rules, our approach can be easily implemented in other ways, for example, using suspended goals in constraint logic programming.

As the next step, we will evaluate the effect of using different solvers, for our final CSPs on the element variables and compare their suitability with the constraint logic programming approach. In particular, we hope to apply our approach to the Kodkod encoding of OCL in [19], in order to directly compare the bit-blasting approach and our two-step approach. Furthermore, we will evaluate the applicability and performance of our approach using further, more extensive case studies.

## References


1. Ali, S., Iqbal, M.Z.Z., Arcuri, A., Briand, L.C.: A Search-Based OCL Constraint Solver for Model-Based Test Data Generation. In: QSIC. pp. 41–50 (2011)
2. Anastasakis, K., Bordbar, B., Georg, G., I.Ray: UML2Alloy: A Challenging Model Transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MoDELS 2007. LNCS, vol. 4735. Springer (2007)
3. Baudry, B., Ghosh, S., Fleurey, F., France, R.B., Traon, Y.L., Mottu, J.M.: Barriers to systematic model transformation testing. Commun. ACM 53(6), 139–143 (2010)
4. Bjorner, N., Nieuwenhuis, R., Veith, H., Voronkov, A. (eds.): Decision Procedures in Soft, Hard and Bio-ware - Follow Up, vol. 1(7). Dagstuhl Reports (2011)
5. Bjørner, N., Tillmann, N., Voronkov, A.: Path Feasibility Analysis for String-Manipulating Programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer (2009)

6. Büttner, F., Cabot, J., Gogolla, M.: On Validation of ATL Transformation Rules By Transformation Models. In: Weißleder, S., Lúcio, L., Cichos, H., Fondement, F. (eds.) MoDeVVa'2011, Proceedings. ACM Digital Library (2012), dOI 10.1145/2095654.2095666
7. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83(2), 283–302 (2010)
8. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) Automated Software Engineering, ASE 2007, Proceedings. ACM (2007)
9. Cadoli, M., Calvanese, D., De Giacomo, G., Mancini, T.: Finite Satisfiability of UML Class Diagrams by Constraint Programming. In: In Proc. of the CP 2004 Workshop on CSP Techniques with Immediate Application (2004)
10. Clavel, M., Egea, M., de Dios, M.A.G.: Checking Unsatisfiability for OCL Constraints. Electronic Communications of the EASST 24, 1–13 (2009)
11. The EMFtoCSP tool. Website., http://code.google.com/a/eclipselabs.org/p/emftocsp/
12. Frühwirth, T.W.: Constraint Handling Rules. In: Podelski, A. (ed.) Constraint Programming. LNCS, vol. 910, pp. 90–107 (1994)
13. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69(1-3), 27–34 (2007)
14. Golden, K., Pang, W.: Constraint Reasoning over Strings. In: Rossi, F. (ed.) Principles and Practice of Constraint Programming - CP 2003. LNCS, vol. 2833 (2003)
15. González, C.A., Büttner, F., Clarisó, R., Cabot, J.: EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In: Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA), workshop at ICSE 2012, Proc., to appear. (2012), http://www.formsera.org/FormSERA
16. Grahne, G., Nykänen, M., Ukkonen, E.: Reasoning about Strings in Databases. J. Comput. Syst. Sci. 59(1), 116–162 (1999)
17. Hooimeijer, P., Veanes, M.: An Evaluation of Automata Algorithms for String Analysis. In: Jhala, R., Schmidt, D.A. (eds.) VMCAI. LNCS, vol. 6538, pp. 248–262 (2011)
18. Kiezun, A., Ganesh, V., Guo, P.J., Ernst, M.D., Hooimeijer, P., Ganesh, V., Guo, P.J., Ernst, M.D.: HAMPI: A solver for string constraints. In: In International Symposium on Software Testing and Analysis (2009)
19. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In: 8th European Conference on Modelling Foundations and Applications (ECMFA), Proceedings. LNCS, Springer (2012), to appear
20. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 201. LNCS, vol. 6705, pp. 290–306. Springer (2011)
21. Malgouyres, H., Motet, G.: A UML model consistency verification approach based on meta-modeling formalization. In: Proceedings of the 2006 ACM symposium on Applied computing. pp. 1804–1809. SAC '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1141277.1141703
22. Maraee, A., Balaban, M.: Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In: Proceedings of the 3rd European conference on Model driven architecture-foundations and applications. pp. 17–31. ECMDA-FA'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1768765.1768767
23. OMG: Object Constraint Language Specification, version 2.3.1 (Document formal/2012-01-01) (2012)
24. Sen, S., Baudry, B., Mottu, J.M.: Automatic Model Generation Strategies for Model Transformation Testing. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563. Springer (2009)

25. Sneyers, J., Weert, P.V., Schrijvers, T., Koninck, L.D.: As time goes by: Constraint Handling Rules. TPLP 10(1), 1–47 (2010)
26. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 152–170. Springer (2011)
27. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic Regular Expression Explorer. In: ICST. pp. 498–507. IEEE Computer Society (2010)
28. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. Electr. Notes Theor. Comput. Sci. 211, 159–170 (2008)