Lightweight String Reasoning in Model Finding

Fabian Büttner, Jordi Cabot

Software and System Modeling manuscript No.
(will be inserted by the editor)

Lightweight String Reasoning in Model Finding

Fabian Büttner[1], **Jordi Cabot**[1] ⋆

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

April 18, 2013

Abstract Models play a key role in assuring software quality in the model-driven approach. Precise models usually require the definition of well-formedness rules to specify constraints that cannot be expressed graphically. The Object Constraint Language (OCL) is a de-facto standard to define such rules. 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 for these activities have been developed, but to our knowledge, none of them supports OCL string operations on scale that is sufficient for, e.g., model-based testing. As, in contrast, many industrial models do contain such operations, there is evidently a gap. We present a lightweight solver that is specifically tailored to generate large solutions for tractable string constraints in model finding, and that is suitable for directly express the main operations of the OCL datatype String. It is based on constraint logic programming (CLP) and constraint handling rules (CHR), and can be seamlessly combined with other constraint solvers in CLP. We have integrated our solver into the EMFtoCSP model finder, and we show that our implementation efficiently solves several common string constraints on a large instances.

Key words Model instantiation, OCL, String constraints, Constraint logic programming, Constraint handling rules

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

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

been widely accepted as a de-facto standard. In a nutshell, OCL allows expressing 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.

The community has developed several model finding approaches and tools for OCL-annotated models. To deal with the computational complexity of this 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 SAT, satisfiability modulo theory (SMT), relational logic, propositional logic, and constraint satisfaction problems (CSP).

While the results currently available cover an extensive subset of OCL, to our knowledge only the work of Kuhlmann et al. [19] support the String data type and its OCL operations, using Kodkod [29], the SAT-based relational logic solver of Alloy, representing strings as sequence relations. This way of encoding strings is well suited to verify computationally hard constraints on strings, and can thus be applied in verification activities. However, these approaches scale up only to a limited number of string variables, even when the string constraints are tractable. This poses a problem when model finding is to be employed to generate model instances on a larger scale, e.g., for model-based testing. Given that several 'real life' models actually do contain such constraints, and that model-based testing does actually require models of non-trivial size, there is evidently a gap that needs to be addressed.

In this paper we present a lightweight solver for string constraints using constraint logic programming (CLP). Our solver is suited to directly implement the main operations of the OCL datatype String, and it can be seamlessly integrated into CLP-based model finders. Our approach can solve several kinds of OCL string constraints by propagation, showing significantly better performance and scalability than Alloy/SAT-based approaches. Thus, it provides an alternative when we want to generate larger instances of models that have string constraints in their well-formedness rules. It is inferior to SAT on constraints that is cannot handle by propagation.

In a nutshell, we associate two meta-variables to each string variable in our approach: the potential length of the string and, optionally, its domain (a dictionary of possible string values). We use Constraint Handling Rules (CHR) to define the reasoning rules for the constraint store. Those string constraints that cannot be resolved using the CHR rules are finally unfolded into finite domain constraints. Both CHR and finite domain solvers are available in many CLP environments, so our approach is not tool-specific.

We have implemented our solver as a library for the ECL^{i}PS^e CLP environment [26] and integrated it into the EMFtoCSP[1] model finder [12], the successor of UMLtoCSP [5]. A first version of our solver has been presented in [4]. Our current article extends that version by adding reasoning rules about domain and by providing a comparison with Alloy-based approaches.

Paper Organization. In Sect. 2, we first introduce the necessary background on CLP and CHR. We then present our solver in Sect. 3. In Sect. 4, we show how it integrates into EMFtoCSP. In Sect. 5 we discuss the performance and scalability of our approach and compare it to Alloy using several examples. Section 6 discusses threats to validity and Sect. 7 puts our contribution in the context of related works. We conclude in Sect. 8.

2 Background

In this section, we provide the necessary background on CLP and CHR. For a more comprehensive introduction we refer to, e.g., Rossi et al. [25].

2.1 Constraint Logic Programming

Constraint logic programming (CLP) combines logic programming (LP, in our context: Prolog) and dedicated constraint solvers to handle complex constraint satisfaction problems. In pure LP, all literals of a goal are predicates that are defined by Horn clauses, and the goal is evaluated by resolution (backtracking). In CLP, additional constraint predicates may be used as literals. Constraint predicates encountered in the evaluation of a program are not resolved using resolution; instead they are put in a constraint store. Constraint-specific reasoning rules are applied to the store. In particular, if the store is found to be unfeasible, the evaluation of the program backtracks immediately. Furthermore, propagation of constraints might lead to unification of variables and values in the logic program.

Typical constraint predicates that are available in virtually all CLP systems are finite domain constraints. The difference between LP to CLP can be illustrated by the following two programs.

$$\text{member}(X,[1,2,3,4,5]),\ \text{member}(Y,[1,2,3,4,5]),\ Z \text{ is } X+Y,\ Z \geq 9.$$

$$X ::_{\text{fd}} [1,2,3,4,5],\ Y ::_{\text{fd}} [1,2,3,4,5],\ Z =_{\text{fd}} (X+Y),\ Z \geq_{\text{fd}} 9,\ \text{labeling}_{\text{fd}}([X,Y]).$$

In the first (logic) program, the first two goals each introduce 5 choice points in the evaluation (the standard Prolog predicate *member* will bind X and Y to each value of the provided list while backtracking). Thus, the goal $Z \geq 9$ is visited 18 times before it succeeds (on $X = 4$ and $Y = 5$).

In the second (constraint logic) program, we assume a finite domain (fd) solver library to be available. Such solvers are available in most constraint logic programming systems, such as ECL^{i}PS^e SWI-Prolog, and SICStus Prolog. In a nutshell, the finite solver provides a set of constraint predicates and a labeling procedure to solve them. To easily distinguish them, we attach the subscript *fd* to all

[1] available at `http://code.google.com/a/eclipselabs.org/p/emftocsp/`

predicates of the finite domain solver. Using the $::_{\text{fd}}$ predicate, we constrain the ranges of X and Y as before, but without introducing choice points. Instead, these predicates add constraints on X and Y to the constraint store. Similarly, the goals $Z =_{\text{fd}} (X + Y)$ and $Z \geq_{\text{fd}} 9$ add further constraints on X, Y, and Z, which leads to internal propagations in the store. Notice that, unlike in the LP version, X and Y are still variables as this point. Instead of being visited many times on backtracking, they are evaluated just once, recording the constraints in the store. It is only the last goal, $\text{labeling}_{\text{fd}}$, that finally assigns X and Y with values according to their domains, in order to resolve all constraints in the store. In our example CLP program, the domain of X will be restricted to 4..5 at this point, and selecting the first value 4 for X will immediately propagate $Y = 5$ in the store. In general, the processing of $\text{labeling}_{\text{fd}}$ (i.e., the solving of the constraint store) can also lead to backtracking in the logic program when the store is unfeasible. This cannot happen in the above example, though, since none of the goals before $\text{labeling}_{\text{fd}}$ introduces a choice point in the logic program.

2.2 Constraint Handling Rules

Constraint Handling Rules (CHR) are an abstract, high-level technique to describe inference rules for the constraints in the constraint store. Implementations of CHR are available in many CLP environments. For a thorough presentation of the formalism we refer to [9] and [27].

In a CLP environment that supports CHR, predicates can be declared to be CHR constraints. Unlike for the normal LP predicates, no deriving logic clauses are defined for them. When they are encountered in the evaluation of a goal, they are placed on the CHR constraint store. CHR rules define how to process the constraints in the store.

Unlike logic programming rules, which have only a single head (Horn clauses), CHR rules can have multiple heads. They are well suited to express inference and rewrite rules as well as axioms for the store. There are two concepts for rules in CHR, simplification and propagation. Simplification rules are used to replace constraints by simpler, equivalent representations. Propagation rules are used to add new, redundant constraints which may cause earlier failure or further simplifications. CHR rules are applied repeatedly until no more rules are applicable.

Below we show the three syntactic forms of CHR rules: simplification, propagation, and simpagation (which is a mixture of the two former forms).

$$\begin{aligned} \textit{rulename} \ @ \ c_1, \ldots, c_m &\iff g_1, \ldots, g_k \mid d_1, \ldots d_n. \\ \textit{rulename} \ @ \ c_1, \ldots, c_m &\implies g_1, \ldots, g_k \mid d_1, \ldots d_n. \\ \textit{rulename} \ @ \ c_1, \ldots, c_k \setminus c_{k+1}, \ldots, c_m &\implies g_1, \ldots, g_k \mid d_1, \ldots d_n. \end{aligned}$$

In the rules, c_i are CHR constraint predicates over logic variables, g_i are LP predicates, and d_i are either CHR constraint or logic predicates. The first part, to the left of the '@' sign, is the (optional) rule name. The next part is the rule head. The part left of the | is called the guard of the rule, and the last part is called the body

of the rule. When a rule head consists of more than one constraint, they typically share at least one variable.

The common semantics of these rules is that they match a pattern, given by the head, in the constraint store. The constraints in the pattern are related by their common variables, for example, as in the pattern $c_1(S,I), c_2(S,J)$. When a match for a rule is found in the store, the guard of the rule, which is an LP goal, is tested. When the guard succeeds, the rule executes, depending on its kind: The first kind, the simplification rule, removes the matched constraints $c_1, \ldots, c_m$ from the store and replaces them by new constraints $d_1, \ldots d_n$. When d_i is a CHR constraint, it is put on the store, when it is a LP goal, it is immediately evaluated. The second kind, the propagation rule, does essentially the same, but keeps the matched constraints of the head in the store. The third kind, the simpagation rule, is actually a mixture of the former two: It keeps a the first part of the head $c_1, \ldots, c_k$, but removes the second part $c_{k+1}, \ldots, c_m$. CHR rules are executed until no more rules can be applied. For propagation rules and simpagation rules, the CHR environment ensures that such rules are executed only once per constraint. The execution of CHR rules and the execution of the logic program that poses constraint are interwoven in several aspects:

1. The variables in the store are subject to unification in Prolog. That is, a rule can become applicable to constraints in the store as the result of a unification in the logic program.
2. When the CHR rules infer an inconsistency (e.g., the LP goal *fail*), the LP program backtracks.
3. Conversely, as the CHR rules can infer LP goals, the execution of CHR rules can lead to unification of logic variables, too.

The following example, taken from [10], illustrates CHR. It shows an excerpt of a simple solver (lets name him 'b') for Booleans, using a unary constraint predicate $\mathrm{bool_b}$ and a ternary constraint predicate $\mathrm{and_b}$. We define the following handling rules for them. Notice that $\mathrm{and_b}$ has a so-called 'reified' form [25]: Its last argument is the result of the conjunction – $\mathrm{and_b}(X,Y,Z)$ constrains the three variables such that $Z = 1$ iff $X = 1$ and $Y = 1$. This form allows us to construct Boolean expressions from Boolean sub-expressions.

$$\begin{aligned}
\mathrm{bool_b}(0) &\iff \text{true}. & \mathrm{bool_b}(1) &\iff \text{true}.\\
\mathrm{and_b}(0,X,Y) &\iff Y = 0. & \mathrm{and_b}(X,0,Y) &\iff Y = 0.\\
\mathrm{and_b}(1,X,Y) &\iff Y = X. & \mathrm{and_b}(X,1,Y) &\iff Y = X.\\
\mathrm{and_b}(X,Y,1) &\iff X = 1, Y = 1. & \mathrm{and_b}(X,X,Z) &\iff X = Z.\\
\mathrm{and_b}(X,Y,A) \setminus \mathrm{and_b}(X,Y,B) &\iff A = B. & \mathrm{and_b}(X,Y,A) \setminus \mathrm{and_b}(Y,X,B) &\iff A = B.
\end{aligned}$$

With these rules defined for the store, the goal

$$\mathrm{bool_b}(X), \mathrm{bool_b}(Y), \mathrm{bool_b}(Z), \mathrm{bool_b}(U), \mathrm{and_b}(X,Y,Z), \mathrm{and_b}(Z,U,1)$$

will resolve $\mathrm{and_b}(Z,U,1)$ to $Z = 1, U = 1$, (by applying the fifth rule) and consequently $X = 1$ and $Y = 1$ (by applying the same rule again). The next goal,

however, will fail as a result of the third and fifth rule:

$$\text{bool}_\text{b}(X), \text{bool}_\text{b}(Y), \text{and}_\text{b}(X,Y,0), \text{and}_\text{b}(Y,X,1),$$

In general, when a solver is not able to fully reduce all constraints, the remaining predicates have to be explored by search and backtracking (i.e., by labeling the remaining variables with possible values. A simple labeling predicate for the Boolean solver looks as follows:

$$\begin{aligned}&\text{labeling} \ :\!- \ \text{chr\_get\_constraint}(\text{bool}_\text{b}(X)), !, (X = 1; X = 0), \text{labeling}.\\ &\text{labeling}.\end{aligned}$$

The standard predicate chr_get_constraint will remove a constraint from the store. Our labeling procedure then tries to assign either 1 or 0 to that variable. Notice that as a result of labeling one variable, further rules might fire. For example

$$\text{bool}_\text{b}(X), \text{bool}_\text{b}(Y), \text{and}_\text{b}(X,Y,0), \text{labeling}$$

will first try $X = 1$, which in turn directly propagates $Y = 0$, without further labeling.

3 A Lightweight String Constraint Solver

We now introduce our CLP-based solver for string constraints. We define several constraint predicates for strings, and we define reasoning rules on them using CHR. We will show in the later Sect. 4 how to integrate this solver into the EMFtoCSP model finder.

Our approach addresses constraints on strings in a similar fashion as a finite domain solver addresses numeric variables. Essentially, we consider two attributes for each string variable: its potential length and its potential values. We reason about the constraints in the store and propagate domain and length information. At several points, we lift sub-problems of solving strings to finite domain solving.

This section first shortly explains how we represent the string sort in CLP (Section 3.1). Then we introduce the reasoning rules (Sections 3.2–3.4).

3.1 Representations of Strings and String Constraints in CLP

In our constraint system, string values are represented asflat lists of ordinal numbers of a given alphabet $\mathscr{A}$. For example, given $\mathscr{A} = \{\text{a}, \text{b}, \ldots, \text{z}\}$, the string "hello" will be represented as the list $[8,5,12,12,15]$. During its lifetime, a string can appear in three stages: (1) as a logic variable X, (2) unified with a list of elements $[X_1, \ldots, X_n]$ of which one or more are variables (each restricted to the ordinal numbers of $\mathscr{A}$), or (3) unified with a list of ordinal numbers. In the following, we refer to strings in these three stages as *string variables*, *instantiated strings*, and *ground strings*.

Two constraints associate fundamental meta-data to a string variable X: $\text{length}_\text{str}(X,N)$ restricts the finite domain variable N to be the length of X and

$\text{domain}_{\text{str}}(X, D)$ restricts X to be an element of the list of ground strings D. We require that a length constraint is posed for every string variable, because we need an upper length bound for each string in the search. Usually this will happen using a length variable, unless we want to work with fixed-length strings. The length variable allows us to express and reason about length constraints, even before the string is instantiated to a list. The domain constraint is optional. We can express further constraints over strings:

- Equality of two strings X and Y is expressed by $\text{eq}_{\text{str}}(X, Y, R)$. The result variable R is constrained to be 1 when X and Y are equal, and 0 when X and Y are not equal.
- The predicate $\text{concat}_{\text{str}}(X, Y, Z)$ constrains Z to be the concatenation of X and Y.
- The predicate $\text{substr}_{\text{str}}(X, I, J, Y)$ constrains Y to occur as a substring in X from positions I to J (starting at 1).
- The predicate $\text{indexof}_{\text{str}}(X, Y, I)$ constrains I to be the first index (starting at 1) of occurrence of Y in X or 0, if Y does not occur in X. The constraint mirrors the special requirement of OCL that no string – not even the empty string – is contained in the empty string [23, p. 152][2].

Notice that the semantics of these predicates matches exactly the semantics of the corresponding OCL string operations [23]. The string constraints can be seamlessly combined with other predicates from other solvers, in particular with those of a finite domain solver. For example, the OCL invariant

$$\text{inv} : X = Y \text{ implies } X = Z$$

can be expressed by the following CLP goal, assuming that X, Y, and Z are some string variables (e.g., attribute values of an object). Recall that we use the index *fd* to denote constraints that we require from an underlying finite domain solver.

$$\text{eq}_{\text{str}}(X, Y, R_1),\ \text{eq}_{\text{str}}(X, Z, R_2),\ (R_1) \rightarrow_{\text{fd}} (R_2).$$

Similarly,

$$\text{inv} : \text{not}(X.\text{indexOf}(\texttt{"Hello"}) > 0 \text{ and } Y.\text{indexOf}(X) = 0)$$

can be expressed as

$$\begin{aligned} &\text{indexof}_{\text{str}}(X, [8,5,12,12,15], I_1),\ \text{gt}_{\text{fd}}(I_1, 0, R_1), \\ &\text{indexof}_{\text{str}}(Y, X, I_2),\ \text{eq}_{\text{fd}}(I_2, 0, R_2), \\ &\text{and}_{\text{fd}}(R_1, R_2, 0). \end{aligned}$$

[2] There is an open issue regarding this aspect of indexOf in the OMG issue tracker for the OCL spec. (`http://www.omg.org/issues/ocl2-rtf.open.html#Issue17220`).

3.2 Handling Rules for String Constraints

The handling rules for string constraints are divided into three parts: rules that reason about them based on their associated length variable (Def. 1); rules that reason about them based on their domain (Def. 2); and rules that eventually unfold them into finite domain constraints for instantiated strings (Def. 3).

In Def. 1, all rules reason about a string for which a length attribute has been added to the store. In the length handling rules, the first rule *l_range* propagates a finite domain constraint ensuring that all string lengths are between 0 and a constant maximum[3] *MaxLength*. The rule *l_unify* absorbs all length constraints on a string into one, unifying the length variables (a string can have only one length). The rule *l_ground* unifies the length variable and the list length whenever a string becomes instantiated. For example, $\text{length}_{\text{str}}(X,N_1)$, $\text{length}_{\text{str}}(X,N_2)$, $X = [4,5]$ will unify N_1 and N_2 with 2, and $\text{length}_{\text{str}}(X,N)$, $N = 3$, $X = [X_1,X_2]$ will fail.

The remaining rules infer finite domain constraints on the length variables for the different string operations. For equality, rule *l_eq* ensures that equal strings have an equal length and that two empty strings are equal. For indexof, concat, and substr, the rules *l_indexof*, *l_concat*, and *l_substr* ensures that a (positive) index must be consistent with the respective string lengths. For the indexOf predicate, we furthermore ensure the OCL-specific rule that no string is contained the empty string (not even the empty string).

Definition 1 (Length handling rules)

$$
\begin{aligned}
\textit{l\_range} \;@\; & \text{length}_{\text{str}}(X,N) \implies N ::_{\text{fd}} 0..\textit{MaxLength}.\\
\textit{l\_unify} \;@\; & \text{length}_{\text{str}}(X,N_1) \setminus \text{length}_{\text{str}}(X,N_2) \iff N_1 = N_2.\\
\textit{l\_ground} \;@\; & \text{length}_{\text{str}}(X,N) \implies \text{is\_list}(X) \mid \text{length}(X,N).\\
\textit{l\_eq} \;@\; & \text{eq}_{\text{str}}(X,Y,R), \text{length}_{\text{str}}(X,N_X), \text{length}_{\text{str}}(Y,N_Y) \implies\\
& \quad R ::_{\text{fd}} 0..1,\\
& \quad (R =_{\text{fd}} 1) \rightarrow_{\text{fd}} (N_X =_{\text{fd}} N_Y), (N_X =_{\text{fd}} 0 \wedge_{\text{fd}} N_Y =_{\text{fd}} 0) \rightarrow_{\text{fd}} (R =_{\text{fd}} 1)\\
\textit{l\_concat} \;@\; & \text{concat}_{\text{str}}(X,Y,Z), \text{length}_{\text{str}}(X,N_X), \text{length}_{\text{str}}(Y,N_Y), \text{length}_{\text{str}}(Z,N_Z) \implies\\
& \quad N_Z =_{\text{fd}} (N_X + N_Y), N_X \geq_{\text{fd}} 0, N_Y \geq_{\text{fd}} 0.\\
\textit{l\_substr} \;@\; & \text{substr}_{\text{str}}(X,I,J,Y), \text{length}_{\text{str}}(X,N_X), \text{length}_{\text{str}}(Y,N_Y) \implies\\
& \quad I ::_{\text{fd}} 0..\textit{MaxLength}, J ::_{\text{fd}} 0..\textit{MaxLength},\\
& \quad N_Y =_{\text{fd}} (J - I + 1), N_X \geq_{\text{fd}} J, I \geq_{\text{fd}} 1, J \geq_{\text{fd}} I, N_X >_{\text{fd}} 0\\
\textit{l\_indexof} \;@\; & \text{indexof}_{\text{str}}(X,Y,I), \text{length}_{\text{str}}(X,N_X), \text{length}_{\text{str}}(Y,N_Y) \implies\\
& \quad I ::_{\text{fd}} 0..\textit{MaxLength},\\
& \quad (I \geq_{\text{fd}} 1) \rightarrow_{\text{fd}} (I + N_Y - 1 \leq_{\text{fd}} N_X),\\
& \quad (I \geq_{\text{fd}} 1) \rightarrow_{\text{fd}} (N_X \geq_{\text{fd}} 1).
\end{aligned}
$$

[3] The maximum length can be arbitrarily large, in our implementation we use 1000 as the default value.

In the domain handling rules (Def. 2), the first rule *d_empty* ensures that a string cannot have an empty domain. When the domain of a string variable has exactly one value, the variable can be unified with this string (rule *d_singleton*). Conversely, when a domain-constrained string variable becomes ground, the domain constraint can be replaced by a check that the ground value is in the domain (rule *d_ground*). When there are two domain constraints on one string variable (which can happen, e.g., as a result of unification), the two previous domains are replaced by their intersections by rule *d_unify*. For example,

$$\text{domain}_{\text{str}}(X,[[1,1],[1,2,3]]),\text{domain}_{\text{str}}(Y,[[1,2,3],[2,3,4]]),X=Y$$

will trigger first *d_unify* and then *d_singleton*, resulting to a ground assignment $X = Y = [1,2,3]$. We assume that the lists of strings the domain of a string variable are always sorted, and use the standard Prolog predicates on ordered lists for checking list membership (ord_memberchk), element removal (ord_del_element) and intersection (ord_intersect).

The remaining domain handling considers the string operation constraints. When the result variable of an equality constraint is 0 and one of the two operands is ground, the rules *d_neq_X* and *d_neq_Y* shrink the domain of the other operand accordingly to the intersection of both domains. For example,

$$\text{domain}_{\text{str}}(X,[[1,2],[2,3],[3,4]]),\text{eq}_{\text{str}}(X,[1,2],R),R=0$$

will resolve to $\text{domain}_{\text{str}}(X,[[2,3],[3,4]])$. Notice that both rules, as all other rules that restrict the domain of a string, contain a comparison of the old and the new domain in their guards, in order to avoid replacing a domain constraint by an equivalent version.

In a similar fashion as the *d_neq_X* and *d_neq_Y*, the two rules *d_substr_YIJ* and *d_substr_YI* reduce the domain of the super-string when the substring becomes ground (the first version takes furthermore a ground index into account, the second version considers substrings at any position). We assume two helper predicates keep_substr and keep_substr_idx (not shown here), to filter the domain accordingly.

For indexof, we employ a little trick to early reduce the domain of the super-string: When both the substring and the index become ground and greater than 0, we can reduce the domain by rule *d_indexofYI* similarly to *d_substr_YIJ*. However, the index value 0 plays an important role, as it expresses 'does not contain', and we want to react on the index even before it becomes ground. Therefore, the rule *d_indexof1* derives a new helper constraint (specific to indexof) that captures containment using a Boolean result variable, constrained to be 0 whenever the index is 0 and 1 otherwise. When this result variable becomes ground, the two rules *d_indexof1_f* and *d_indexof1_t* filter the domain accordingly. The trick here is that the result variable can become ground even when the index variable is not. For example,

$$\text{domain}_{\text{str}}(X,[[1,2],[1,3],[2,3]]),\text{indexof}_{\text{str}}(X,[3],I),I>_{\text{fd}}1$$

will infer $\text{domain}_{\text{str}}(X,[[1,3],[2,3]])$, assuming that the underlying finite domain solver infers $\text{indexof1}_{\text{str}}(X,[[1,2],[1,3],[2,3]],0)$ from $I >_{\text{fd}} 1$. For the filtering of the domain we assume a third helper predicate remove_substring (not shown).

Definition 2 *Domain handling rules*

$$d\_empty \ @\ \mathrm{domain}_{\mathrm{str}}(X,[]) \iff \mathrm{fail}.$$
$$d\_singleton \ @\ \mathrm{domain}_{\mathrm{str}}(X,[V]) \iff X = V.$$
$$d\_ground \ @\ \mathrm{domain}_{\mathrm{str}}(X,D) \iff \mathrm{ground}(X) \mid \mathrm{ord\_memberchk}(X,D).$$
$$d\_unify \ @\ \mathrm{domain}_{\mathrm{str}}(X,D_1), \mathrm{domain}_{\mathrm{str}}(X,D_2) \iff \mathrm{ord\_intersect}(D_1,D_2,D), \mathrm{domain}_{\mathrm{str}}(X,D_2).$$
$$d\_neq\_X \ @\ \mathrm{eq}_{\mathrm{str}}(X,Y,0) \setminus \mathrm{domain}_{\mathrm{str}}(Y,D) \iff \mathrm{ground}(X), \mathrm{ord\_del\_element}(D,X,D_1), D \neq D_1 \mid \mathrm{domain}_{\mathrm{str}}(Y,D_1).$$
$$d\_neq\_Y \ @\ \mathrm{eq}_{\mathrm{str}}(X,Y,0) \setminus \mathrm{domain}_{\mathrm{str}}(X,D) \iff \mathrm{ground}(Y), \mathrm{ord\_del\_element}(D,Y,D_1), D \neq D_1 \mid \mathrm{domain}_{\mathrm{str}}(X,D_1).$$
$$d\_substr\_YIJ \ @\ \mathrm{substr}_{\mathrm{str}}(X,I,J,Y) \setminus \mathrm{domain}_{\mathrm{str}}(X,D_1) \iff \mathrm{ground}([Y,I,J]), \mathrm{keep\_substr\_idx}(D_1,I,Y,D_2), D_1 \neq D_2 \mid \mathrm{domain}_{\mathrm{str}}(X,D_2).$$
$$d\_substr\_Y \ @\ \mathrm{substr}_{\mathrm{str}}(X,I,J,Y) \setminus \mathrm{domain}_{\mathrm{str}}(X,D_1) \iff \mathrm{ground}(Y), \mathrm{keep\_substr}(D_1,I,Y,D_2), D_1 \neq D_2 \mid \mathrm{domain}_{\mathrm{str}}(X,D_2).$$
$$d\_indexof\_YI \ @\ \mathrm{indexof}_{\mathrm{str}}(X,Y,I) \setminus \mathrm{domain}_{\mathrm{str}}(X,D_1) \iff \mathrm{ground}([Y,I]), I \neq 0, \mathrm{keep\_substr\_idx}(D_1,I,Y,D_2), D_1 \neq D_2 \mid \mathrm{domain}_{\mathrm{str}}(X,D_2).$$
$$d\_indexof1 \ @\ \mathrm{indexof}_{\mathrm{str}}(X,Y,I) \implies (I =_{\mathrm{fd}} 0) \rightarrow_{\mathrm{fd}} (R =_{\mathrm{fd}} 0), (I \neq_{\mathrm{fd}} 0) \rightarrow_{\mathrm{fd}} (R =_{\mathrm{fd}} 1), \mathrm{indexof1}_{\mathrm{str}}(X,Y,R).$$
$$d\_indexof1\_f \ @\ \mathrm{indexof1}_{\mathrm{str}}(X,Y,0), \mathrm{domain}_{\mathrm{str}}(X,D) \iff \mathrm{ground}(Y), \mathrm{remove\_substring}(D,Y,D_2), D \neq D_2 \mid \mathrm{domain}_{\mathrm{str}}(X,D_2).$$
$$d\_indexof1\_t \ @\ \mathrm{indexof1}_{\mathrm{str}}(X,Y,1), \mathrm{domain}_{\mathrm{str}}(X,D) \iff \mathrm{ground}(Y), \mathrm{keep\_substr}(D,Y,D_2), D \neq D_2 \mid \mathrm{domain}_{\mathrm{str}}(X,D_2).$$

The last package of CHR rules (Def. 3) covers the instantiation of strings by unfolding the constraints on the level of the individual element variables. However, as a shortcut, the first two rules *i_eq_f* and *i_eq_t* reduce ground equalities and ground inequalities directly to their prolog term pendants for performance reasons. For the other cases, unfold_eq(X,Y,R) unfolds the equality of strings into a big conjunction of element-wise equalities using the helper predicate given in Def. 4. For example, $\mathrm{eq}_{\mathrm{str}}([X_1,X_2,X_3],[Y_1,Y_2,Y_3],R), X_1 = 1, Y_1 = 2$ infers $R = 0$.

Definition 4 assumes that the lengths of the strings are consistent, as enforced by the rules in Def. 1. The concat operation is unfolded directly by appending the lists. The substring constraint is unfolded into a set of implications: For each potential value of I, the corresponding conjunction of element-wise equality is posed. For example, $\mathrm{substr}_{\mathrm{str}}([X_1,X_2,X_3],I,[Y_1,Y_2],)$ unfolds to $(I =_{\mathrm{fd}} 1) \rightarrow_{\mathrm{fd}} (X_1 =_{\mathrm{fd}} Y_1 \wedge_{\mathrm{fd}} X_2 =_{\mathrm{fd}} Y_2)$, $(I =_{\mathrm{fd}} 2) \rightarrow_{\mathrm{fd}} (X_2 =_{\mathrm{fd}} Y_1 \wedge_{\mathrm{fd}} X_3 =_{\mathrm{fd}} Y_2)$. When the superstring is instantiated and the index parameters are ground, the substring can be directly unified with the elements in the superstring.

Notice that the unfolding of the indexOf constraint is more complex than the one for substrings, since we want to capture (a) the deterministic 'first match' se-

mantics that OCL gives to the indexOf operation and (b) the meaning of 0 as 'does not contain'. Hence, the unfolding captures all the potential matching indexes as variables in the list P (using a value of $MaxLength+1$ to capture a no-match for the respective position in X). The result R is then constrained to be the minimum of P, if it less or equal than $MaxLength$, and to be 0, otherwise.

Definition 3 *Instantiation rules*

$$\begin{aligned}
i_eq_cut_f \;@\; & \mathrm{eq}_{\mathrm{str}}(X,Y,0) \iff \mathrm{ground}([X,Y]) \mid X \neq Y.\\
i_eq_cut_t \;@\; & \mathrm{eq}_{\mathrm{str}}(X,Y,1) \iff \mathrm{ground}([X,Y]) \mid X = Y.\\
i_eq_var \;@\; & \mathrm{eq}_{\mathrm{str}}(X,Y,R) \iff \mathrm{is_list}(X),\mathrm{is_list}(Y) \mid \mathrm{unfold_eq}(X,Y,R).\\
i_concatXY \;@\; & \mathrm{concat}_{\mathrm{str}}(X,Y,Z) \iff \mathrm{is_list}(X),\mathrm{is_list}(Y) \mid \mathrm{append}(X,Y,Z).\\
i_concatXZ \;@\; & \mathrm{concat}_{\mathrm{str}}(X,Y,Z) \iff \mathrm{is_list}(X),\mathrm{is_list}(Z) \mid \mathrm{append}(X,Y,Z).\\
i_concatYZ \;@\; & \mathrm{concat}_{\mathrm{str}}(X,Y,Z) \iff \mathrm{is_list}(Y),\mathrm{is_list}(Z) \mid \mathrm{append}(X,Y,Z).\\
i_substrXIJ \;@\; & \mathrm{substr}_{\mathrm{str}}(X,I,J,Y) \iff\\
& \qquad \mathrm{is_list}(X),\mathrm{ground}(I),\mathrm{ground}(J) \mid N \text{ is } J-I+1,\mathrm{sublist}(X,I,N,Y).\\
i_substrXY \;@\; & \mathrm{substr}_{\mathrm{str}}(X,I,J,Y) \iff \mathrm{is_list}(X),\mathrm{is_list}(Y) \mid \mathrm{unfold_substr}(X,I,J,Y).\\
i_indexof \;@\; & \mathrm{indexof}_{\mathrm{str}}(X,Y,I) \iff \mathrm{is_list}(X),\mathrm{is_list}(Y) \mid \mathrm{unfold_indexof}(X,Y,I).\\
i_indexof1 \;@\; & \mathrm{indexof1}_{\mathrm{str}}(X,Y,R) \iff \mathrm{is_list}(X),\mathrm{is_list}(Y) \mid \mathrm{true}.
\end{aligned}$$

Definition 4 *Unfolding of constraints We assume X and Y be instantiated strings of element variables that are constrained by* $\mathrm{icdomain}_{\mathrm{fd}}$ *to the range of the alphabet* $\mathscr{A}$*. We assume I to be constrained to be a number by* $I ::_{\mathrm{fd}} 1..MaxLength$ *and R to be constrained to be a truth value by* $R ::_{\mathrm{fd}} 0..1$*. The predicates* $\mathrm{unfold\_eq}(X,Y,R)$*,* $\mathrm{unfold\_substr}(X,I,Y)$ *and* $\mathrm{unfold\_indexof}(X,Y,I)$ *unfold the semantics of the corresponding CHR predicates using finite domain solver constraints.*

$$\begin{aligned}
&\mathrm{unfold_eq}(X,Y,R) \;:-\\
&\quad (\mathrm{foreach}(X_0,X),\mathrm{foreach}(Y_0,Y),\mathrm{fromto}(1,R_1,R_2,R)\ \mathrm{do}\\
&\qquad \mathrm{and}_{\mathrm{fd}}(X_0 =_{\mathrm{fd}} Y_0,R_1,R_2)\ ).\\
&\mathrm{unfold_substr}(X,I,Y) \;:-\\
&\quad \mathrm{length}(X,N_X),\mathrm{length}(Y,N_Y),D \text{ is } N_X-N_Y,\\
&\quad (\mathrm{count}(F,0,D),\mathrm{param}(I,X,Y,N_Y)\ \mathrm{do}\\
&\qquad (\mathrm{count}(K,1,N_Y),\mathrm{fromto}(1,R_1,R_2,R_3),\mathrm{param}(F,X,Y)\ \mathrm{do}\\
&\qquad\quad L \text{ is } K+F,\mathrm{nth1}(L,X,X_0),\mathrm{nth1}(K,Y,Y_0),\\
&\qquad\quad \mathrm{and}_{\mathrm{fd}}(R_1,X_0 =_{\mathrm{fd}} Y_0,R_2)\ ),\\
&\qquad (I =_{\mathrm{fd}} F+1)\ \rightarrow_{\mathrm{fd}}\ (R_3 =_{\mathrm{fd}} 1)).\\
&\mathrm{unfold_indexof}(X,Y,I) \;:-\\
&\quad \mathrm{length}(X,N_X),\mathrm{length}(Y,N_Y),D \text{ is } N_X-N_Y,\\
&\quad (\mathrm{count}(F,0,D),\mathrm{param}(I,Y,X,N_Y),\mathrm{fromto}(0,R_1,R_5,R_6)\ \mathrm{do}
\end{aligned}$$

$$
\begin{array}{l}
\quad (\mathrm{count}(J,1,N_Y),\mathrm{fromto}(1,R_2,R_3,R_4),\mathrm{param}(Y,X,F)\ \mathrm{do}\\
\qquad K\ \mathrm{is}\ J+F,\mathrm{nth1}(J,Y,X_0),\mathrm{nth1}(K,X,Y_0),\\
\qquad \mathrm{and}_{\mathrm{fd}}(R_2,X_0 =_{\mathrm{fd}} Y_0,R_3)\ ),\\
\quad ((R_1 =_{\mathrm{fd}} 0) \wedge_{\mathrm{fd}} (R_4 =_{\mathrm{fd}} 1)) =_{\mathrm{fd}} (I =_{\mathrm{fd}} F+1),\\
\quad \mathrm{or}_{\mathrm{fd}}(R_1,R_4,R_5))\\
(I =_{\mathrm{fd}} 0) =_{\mathrm{fd}} (R_6 =_{\mathrm{fd}} 0).
\end{array}
$$

3.3 Labeling

In some cases, processing the presented rules can directly yield a solution (i.e., ground values for all string variables). In general, however, we have to explore the search space using backtracking at some point. In Def. 5 below we provide a standard labeling predicate $\mathrm{labeling}_{\mathrm{str}}$ that consists of three steps. First, all strings that have a domain constraint attached are labeled. Then, all remaining length variables are labeled and the string variables are instantiated to lists accordingly. Finally, the remaining elements are labeled.

To label a variable that has a domain constraint, the constraint is taken out of the store and the variable is bound to an element of the domain. On backtracking, all values are tried. Similary, to label a length constraint, all values of the domain are tried on backtracking. The actual labeling of the length variable is done by the finite domain solver. During backtracking through the labeling goal, constraints are put back on the store. When a ground integer value is assigned to a length variable, the string variable is instantiated to a list of that length, and all element variables are restricted to the domain of the alphabet $\mathscr{A}$.

Notice that unifying string variables with ground strings or instantiated strings can fire further handling rules. For example, in

$$D = [[1,2],[2,3]],\mathrm{domain}_{\mathrm{str}}(X,D),\mathrm{domain}_{\mathrm{str}}(Y,D),\mathrm{eq}_{\mathrm{str}}(X,Y,0),\mathrm{labeling}_{\mathrm{str}}$$

the labeling procedure will first remove $\mathrm{domain}_{\mathrm{str}}(X,D)$ and assign $[1,2]$ to X. This in turn triggers the rule *d_neg_X*, which reduces the domain of Y to a singleton which in turn replaces the domain constraint for Y into $Y = [2,3]$. Analogously, for the length labeling phase, instantiating string variables fires in particular the rules from Def. 3.

Definition 5 *Standard labeling predicate*

$$
\begin{array}{l}
\mathrm{labeling}_{\mathrm{str}}\ \mathrm{:\!-}\ \mathrm{label\_domains}_{\mathrm{str}},\mathrm{label\_lengths}_{\mathrm{str}}(Cs),\mathrm{labeling}_{\mathrm{fd}}(Cs).\\
\mathrm{label\_domains}_{\mathrm{str}}\ \mathrm{:\!-}\\
\quad \mathrm{chr\_get\_constraint}(\mathrm{domain}_{\mathrm{str}}(X,D)),!,\\
\quad \mathrm{member}(X,D),\\
\quad \mathrm{label\_domains}_{\mathrm{str}}.\\
\mathrm{label\_domains}_{\mathrm{str}}.
\end{array}
$$

$$\begin{aligned}
&\text{label_lengths}_{\text{str}}(C) \ :- \\
&\quad \text{chr_get_constraint}(\text{length}_{\text{str}}(X,N)), !, \\
&\quad N ::_{\text{fd}} 1..\text{maxOrdinal}(\mathscr{A}), \text{labeling}_{\text{fd}}([N]), \text{length}(X,N), \\
&\quad \text{label_lengths}_{\text{str}}(C_1), \\
&\quad \text{append}(X,C_1,C). \\
&\text{label_lengths}_{\text{str}}([]).
\end{aligned}$$

3.4 Extension: An alldifferent Constraint for Strings

A common constraint in CSP is alldifferent($[X_1,\ldots,X_n]$), which states that all variables X_i are mutually different. Several models contain such invariants for strings, more specifically, for names and identifiers (e.g., of types within a namespace, of attributes within a class). While the alldifferent constraint can be blasted into several inequalities, $\text{eq}_{\text{str}}(X_i,X_j,0)$ for $i \neq j$ in our case, it can be implemented more efficiently using *n outof* constraints for strings that have domains. We adopt this standard technique by introducing two additional CHR constraints: $\text{outof}_{\text{str}}(X,A,B)$ expresses that the string X does not occur in any element of the two sets of strings A and B, and $\text{domainhole}_{\text{str}}(X,Y)$ expresses that the domain of X does not contain the ground string Y. We can unroll $\text{alldifferent}_{\text{str}}(L)$ on a list of domain-restricted strings as follows. For each string variable, A and B are the variables before and after that variable.

$$\begin{aligned}
&\text{alldifferent}_{\text{str}}(L) \ :- \ \text{alldifferent}_{\text{str}}(L,[]) \\
&\text{alldifferent}_{\text{str}}([X|B],A) \ :- \ \text{outof}_{\text{str}}(X,A,B), \text{alldifferent}_{\text{str}}(B,[X|A]). \\
&\text{alldifferent}_{\text{str}}([],A).
\end{aligned}$$

The handling rules for *outof* are given in Def. 6. Whenever one of the strings in an alldifferent constraint becomes ground, the rule *outof_ground* excludes this string from all other variables by posing $n-1$ $\text{domainhole}_{\text{str}}$ predicates (using the helper operations given in Def. 7). When the string of this constraint has a domain, the domain is reduced accordingly by rule *outof_hole_domain*. When the string is ground, *outof* is simplified to a inequality check by rule *outof_hole_ground*. Given that the domain contains enough values, alldifferent can be successfully processed without backtracking.

Definition 6 *Rules for alldifferent constraint*

$$\begin{aligned}
\textit{outof_ground} \ @ \ &\text{outof}_{\text{str}}(X,L,R) \iff \\
&\quad \text{ground}(X) \mid \text{exclude_value}(L,X), \text{exclude_value}(R,X) \\
\textit{outof_hole_domain} \ @ \ &\text{domainhole}_{\text{str}}(X,V), \text{domain}_{\text{str}}(X,D_1) \iff \\
&\quad \text{ord_del_element}(D_1,V,D_2), D_1 \neq D_2 \mid \text{domain}_{\text{str}}(X,D_2)
\end{aligned}$$

$$\mathit{outof\_hole\_ground} \ @ \ \mathrm{domainhole}_{\mathrm{str}}(X,V) \iff \mathrm{ground}(X) \mid X \neq V$$

Definition 7 *Helper predicates for all different*

$$\mathrm{exclude\_value}([],Y).$$

$$\mathrm{exclude\_value}([X|Y],N) \ :- \ \mathrm{domainhole}_{\mathrm{str}}(X,N), \mathrm{exclude\_value}(Y,N).$$

4 Integration into the EMFtoCSP Model Finder

We now show how our string constraint solver integrates a CLP-based model finder. We discuss this for our tool EMFtoCSP [13,5], but our string solver is not specific to it. EMFtoCSP is available as an open source plugin for Eclipse. It provides an API as well as a graphical user interface (available as a context action on *.ecore* and *.uml* files). Figure 1 shows the main configuration dialogs and a search result for the Entity-Relationship example bundled with the installation. EMFtoCSP internally uses the open source ECL$^{\mathrm{i}}$PS$^{\mathrm{e}}$ CLP environment, which provides the required solvers for finite domain constraints and constraint handling rules.

Finding instances of a model can be considered as a special kind of a constraint satisfaction problem (CSP), with the sets of objects and properties in the model as values and the well-formedness rules as the constraints over them. A satisfying assignment of such a CSP is a valid instance of the model. Like our string constraints, model finding CSPs be solved using CLP, too. Cabot et al. describe such

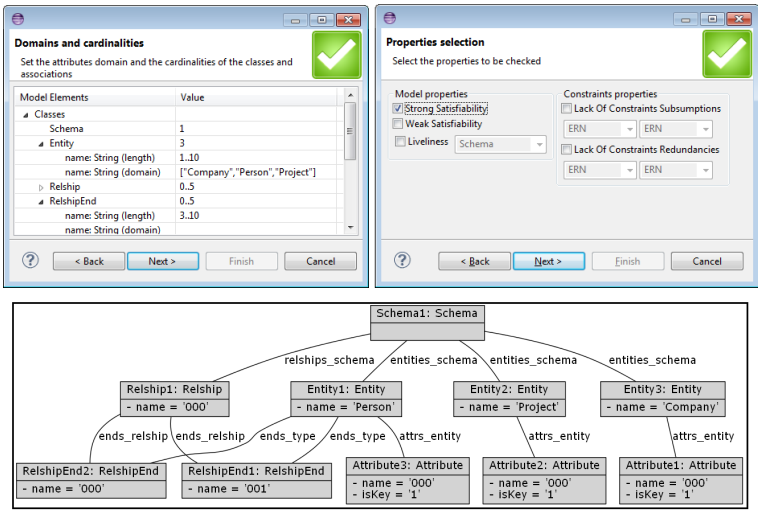


Fig. 1: EMFtoCSP: Search bound selection, configuration, and result.

a translation in [5]. Given a model and a set of OCL well-formedness rules, their approach defines how to infer a CLP P that succeeds exactly if the constrained model is satisfiable. The solutions that are returned by P on success correspond to valid instances of the model.

Technically, the derived CLP P solves two sub-problems, the *cardinality problem*, and the *instance problem*. The variables of the first sub-problem are sizes for the extents of objects and associations. For each solution of the cardinality problem, a potential instance (representing a partially instantiated object diagram) can be constructed, in which the various links and attribute values of the objects are the variables. These variables are considered in the second, dependent sub-problem. Figure 2 illustrates the search process of EMFtoCSP (including the strings part explained below), showing how an instance I gradually gets more and more instantiated. The generated CLP program may be shown abstractly as follows:

$$\begin{aligned}\text{solution}(I) \;:\!-\; &\text{validCardinalities}(I,C),\ \text{labeling}(C),\\ &\text{initInstanceVariables}(I,V),\\ &\text{validInstance}(I),\ \text{labeling}(V).\end{aligned}$$

In the first step, the solution I consists of uninstantiated lists of objects and links. The cardinality sub-problem is unfolded into finite domain solver constraints over the lengths variables C of these lists. The predicate validCardinalities particularly reflects all multiplicity, inheritance, and composition constraints of the model. The solutions of the sub-problem are iterated by labeling. For each valid cardinality assignment, a structure for a potential model is instantiated as lists of numeric variables V, and the constraints of the second sub-problem are posed on the potential model (again, taking advantage of the underlying finite domain solver). Thus, the predicate validInstance(I) encodes all remaining OCL constraints. The variables encode both the links between the objects and their attribute values.

The string constraints that we presented in previous section can be seamlessly 'plugged' into validInstance(I). In the extended EMFtoCSP version, the CLP program becomes:

$$\begin{aligned}\text{solution}(I) \;:\!-\; &\text{validCardinalities}(I,C),\ \text{labeling}(C),\\ &\text{initInstanceVariables}(I,V_{numeric}),\\ &\text{validInstance}(I),\ \text{labeling}(V_{numeric}),\\ &\text{labeling}_{\text{str}}.\end{aligned}$$

The predicate validInstance(I) now also encodes all OCL string operations, although we decide to label them after labeling the links and numeric attributes. However, CHR rules may fire before the explicit labeling of the strings, due to propagations on length variables. Note that EMFtoCSP keeps all integer variables explictly (in $V_{numeric}$), whereas our string solver extracts them from the constraint store. Hence there is no list of variables passed to $\text{labeling}_{\text{str}}$.

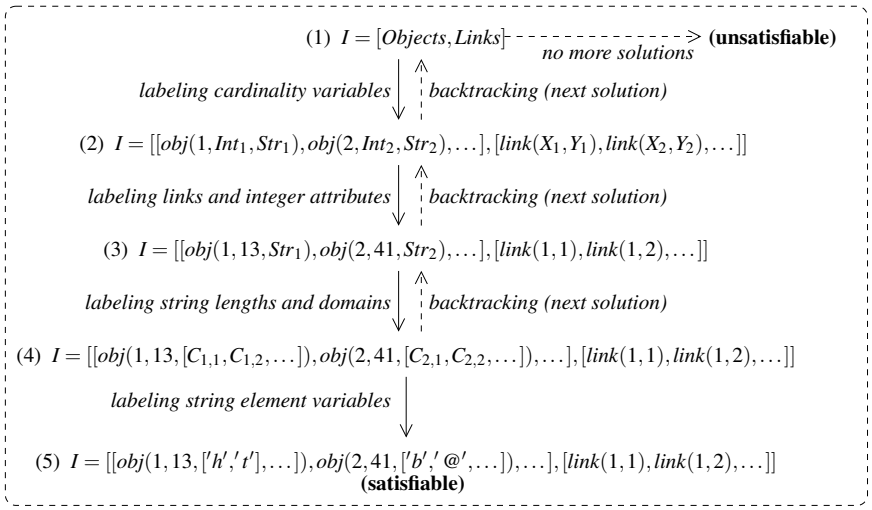


Fig. 2: EMFtoCSP's search process, showing the different instantiation levels of the solution I: (1) cardinalities; (2) partial object diagram with potential links; (3) with ground links and numeric attributes; (4) with strings instantiated as lists; and (5) fully ground (i.e., satisfiable).

5 Performance and Scalability

We are aware that the computational complexity of satisfiability solving for string constraints is high even for apparently simple fragments – for example, the satisfiability problem over the theory of strings with equality, containment, negation, and conjunction, is already NP-hard [17]. Thus, there cannot be a perfect solution for all problems. Our hypothesis is that many string problems in MDE model instantiation are tractable and solvable by propagation, although they are often larger in the number of variables. Our approach explicitly addresses those problems.

We have evaluated the performance and scalability of our library using several problems that our solver can handle by propagation, and compared the results with the popular relational model finder Alloy [16], which is based on Boolean satisfiability solvers (SAT solvers) internally [29], and which has the reputation to be efficient on many combinatorial problems.

Using the presented generic labeling predicate (Def. 5), string CSPs can be categorized into two scalability classes for our approach. The class on which it scales very well comprises those problems that can be solved completely by propagation, without backtracking. In this class, our approach can in general handle thousands of string variables, a number that is impossible to reach with Alloy.

The second class comprises those problems that require backtracking in at least one of the labeling phases. When backtracking occurs only in length or domain labeling, the search space can still be traversed exhaustively when the number of string variables, the size of their domains, and the allowed ranges for string lengths are small enough. However, labeling all elements of all instantiated strings

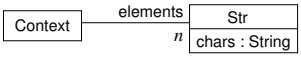


Fig. 3: Example model. The value for n is varied in the following tests.

is usually unfeasible (although a little modification of the labeling procedure can help to skip a specific 'local labeling traps', as we explain below).

The remaining section illustrates both classes and provides performance results for several test cases. For the CLP version, we have translated the OCL invariants into our string constraint version using our extended version of EMFtoCSP. For simple syntactic patterns, alldifferent constraints are generated when all strings are declared with a domain. For Alloy, we have represented the strings using the built-in sequence datatype. We tested our implementation on all test cases in two modes:

1. Using string variables with a range for their lengths between l and u, but without giving them domains of ground values. We use a to denote the size of the alphabet.
2. Using a domain of w different ground strings as the domain for all string variables.

We have conducted all performance tests using ECL$^{\text{i}}$PS$^{\text{e}}$ 6.0 (for the CLP version) and Java 1.7 and Alloy 4.2 (for the Alloy version). All tests were run on an Intel Dual Core 2.2 Ghz processor with 4GB RAM.

The first test case MIN_LENGTH is trivially satisfiable and gives an impression of the capabilities to handle many string variables. The only restriction, posed on all string variables, is that their length is at least 2. In OCL:

context Str **inv**: self.chars.size() >= 2

In our CLP approach, this means posing n constraints of the form $\text{length}_{\text{str}}(X_i, N_i), N_i \geq_{\text{fd}} 2$. The listing (a) in Fig. 4 shows the Alloy version of this problem. We have added listing (b) as a reference, too, although it uses an opaque representation of strings, and we cannot express the length constraint this way. It will show, however, the maximum number of variables we can reasonable handle with Alloy. Figure 5 shows the runtimes of the CLP and the Alloy versions for varying values of u (5 and 10) and a (26 and 127). The runtime for the CLP version is only a few milliseconds. We can see that, even for short sequences and a small alphabet, Alloy does not scale far using a translation of strings into sequences of characters, and even using an opaque representation of strings will not allow to handle more than a few hundred strings. The missing numbers for Alloy are due to OutOfMemoryExceptions (given 4GB).

A second, more complicated, but still tractable problem, MUTUALLY_DIFFERENT, that we study is mutual inequality on a set of n string variables. This is a common invariant, e.g., on *name* attributes in many models. Using the class diagram from Fig. 3, this constraint can be formulated in OCL as follows:

```
context Context inv: self.elements->forAll(e1,e2 |
  e1 <> e2 implies e1.chars <> e2.chars)
```

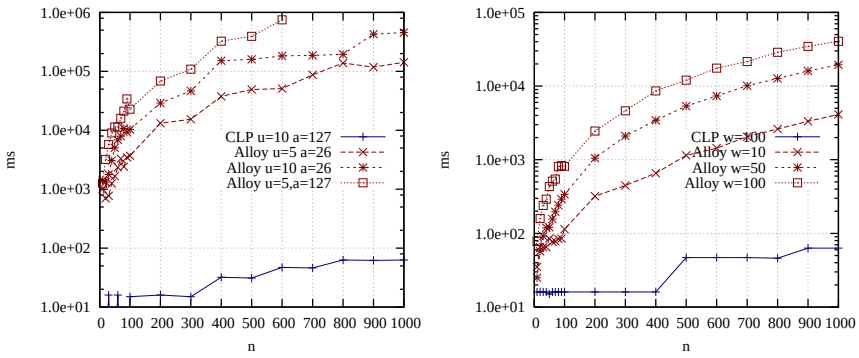
```
sig Alphabet {}
sig Str { chars : seq Alphabet}
fact { all s : Str | #(s.chars) >= 2 }
run {} for u seq, exactly n Str, exactly a Alphabet
```

(a) without domains

```
sig Word {}
sig Str { chars : one Word }
run {} for exactly w Word, exactly n Str
```

(b) with domains

Fig. 4: Alloy specifications for the MIN_LENGTH test case.



(a) without domains (b) with domains

Fig. 5: Performance results for the MIN_LENGTH test case.

For the CLP version, this invariant is unrolled by EMFtoCSP into $n(n-1)$ constraints of the form $\mathrm{eq}_{\mathrm{str}}(X,Y,0)$ (one for each pair of variables X and Y). Optionally, simple syntactic forms as the given OCL invariant can be recognized and translated into an alldifferent constraint (when string domains are used). Figure 6 shows the Alloy versions. We have omitted, again, the length constraint for the case with string domains in Alloy.

As before, we have tested our implementation on this case in two modes, with and without domains. When using domains, we have used $w = 1.5n$ as the size of the domain dictionary. The entries in the dictionary of the CLP version were randomly generated (but all different from each other) with random lengths between 1 and 15.

The first plot in Fig. 7 shows the results for the domain-free case. For the tests, we have varied the size of the alphabet (26 and 127) and the allowed range for lengths (2..5, 2..10, 2..30). We can see that the runtime of Alloy becomes very large even for small string lengths and a small alphabet size. In contrast, our ap-

```
sig Alphabet {}
sig Str { chars : seq Alphabet}
fact { all s1, s2: Str | s1 != s2 implies s1.chars != s2.chars }
fact { all s : Str | #(s.chars) >= l }
run {} for u seq, exactly n Str, exactly a Alphabet
```

(a) without domains

```
sig Word {}
sig Str { chars : one Word }
fact { all s1, s2 : Str | s1 != s2 implies s1.chars != s2.chars }
run {} for exactly w Word, exactly n Str
```

(b) with domains

Fig. 6: Alloy specifications for the MUTUALLY_DIFFERENT test case.

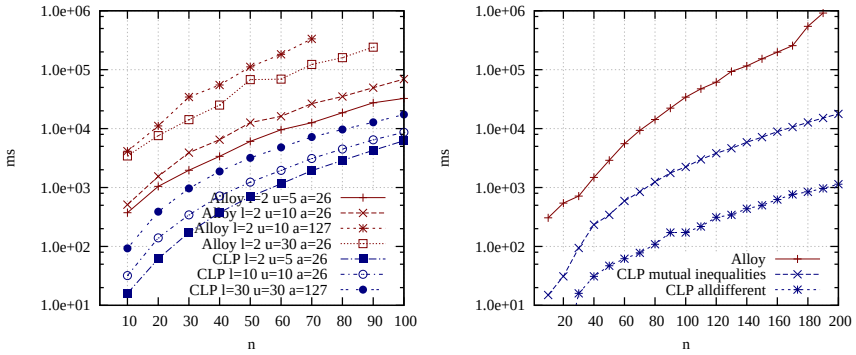
proach handles 100 mutually different strings in less than a second for small string sizes and alphabets. It performs still in less than 10 seconds on the extreme case that all strings have a length of 50, which does not allow for any shortcuts in the propagation due to different string lengths, and which imposes a large number of finite domain constraints on the elements of the strings.

The second plot in Fig. 7 shows the results for the domain dictionary case. Again, the Alloy version scales worst, followed by the unrolled equalities CLP version. The alldifferent version scales best, and can generate several hundred strings in a few seconds.

We have to say clearly that this test case is only tractable for our approach when it is satisfiable. When, the dictionary contains less than n elements (thus, the strings cannot be mutually different), the search space has to be exhaustively traversed. In this case, even for small values for n, our approach becomes unfeasible (for $n = 10$, detecting unsat takes about 30 seconds).

Similarly, when n is larger than the alphabet size a, a labeling procedure that first labels all string length to 1, the element labeling (by the finite domain solver) will try to label the elements in $n!$ combinations (i.e., backtracking steps). In practice, we have modified our labeling procedure to be slightly more elaborated than the one in Def. 5. In our implementation, we perform two labeling passes: In the first pass, we only give the element labeling a backtracking credit of 1 for small values of n (exactly to skip such local labeling). Only when we cannot find a solution in this first pass, we backtrack and perform the element labeling without restrictions. In general, however, the problem remains for unsatisfiable problems, that our approach does not have symmetry breaking, as SAT solvers have. Thus, when the string constraints are unsatisfiable, it will search a (potentially very large) number of symmetric constellations.

Our third test case, DEPENDENT, poses several string constraints to make their content dependent on each other: Given n different strings, at least one string must contain the substring '@' (think of an email address), at least one substring



(a) without domains (b) with domains

Fig. 7: Performance results for the MUTUALLY_DIFFERENT test case.

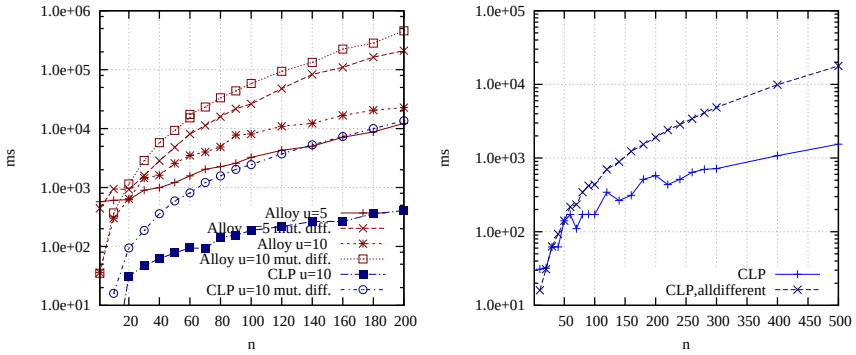
must start with 'http:' (think of a website), and every string that starts with 'http:' must not contain '@'. In OCL:

```
context Context inv: elements->exists(e | e.chars.indexOf('@') <> 0)
context Context inv: elements->exists(e | e.chars.substring(1,5) = 'http:')
context Context inv: elements->forAll(e |
  e.chars.substring(1,5) = 'http:' implies e.chars.indexOf('@') = 0)
```

As a fourth case, DEPENDENT_MUTUALLY_DIFFERENT, we furthermore request that all strings are pairwise different, as in the first test case. We have again translated the OCL invariants into our constraint system using EMFtoCSP, and into Alloy using the default operations on sequence (not shown here). Figure 8 shows the result using strings without domains (left) and using a dictionary containing $w = 1.5n$ random entries, including entries to satisfy the problem (right). We have set $a = 26$ and $l = 2$. For the test case DEPENDENT, both the free labeling and the dictionary version scale very well. The translation into sequences for Alloy, in contrast, becomes again unfeasible for even small numbers of strings, small alphabets, and short lengths. Notice that an Alloy dictionary version without using strings, as for MUTUALLY_DIFFERENT is not possible here, as we consider the individual elements in the second and third test cases. For the DEPENDENT_MUTUALLY_DIFFERENT, both the free labeling and the dictionary version scale still much better than the Alloy version, but, as expected, becomes impracticle sooner than in the second test case.

As for the previous example, if we consider an unsatisfiable version DEPENDENT_UNSAT of the previous test case as shown below, our approach times out for small values of n, u, and a, whereas Alloy can still handle (and detect the unsatisfiability of) those cases, since the unsatisfiability is not detected in the constraint store before actually labeling all individual string elements.

```
context Context inv: elements->forAll(e | e.chars.indexOf('@') <> 0)
context Context inv: elements->forAll(e |
  e.chars.substring(1,5) = 'http:' implies e.chars.indexOf('@') = 0)
context Context inv: elements->exists(e | e.chars.substring(1,5) = 'http:')
```



(a) without domains (b) with domains

Fig. 8: Performance results for the DEPENDENT and DEPENDENT_MUTUALLY_DIFFERENT test cases.

6 Threats to Validity

We see two threats to the validity for the scalability results for our approach. First, our approach scales well on problems that can be solved by propagation, while it degrades massively on problems that require exhaustive search. However, the set of of problems that are tractable in this sense is not explicitly defined. In particular, we do not provide a procedure that could statically determine this a priori for a given set of OCL constraints. Even if such a procedure could be designed in a computational cheap way (which we doubt, given the NP-hard nature of even simple string constraints), it would be dependent of the characteristics of the underlying finite domain solver. Thus, we cannot exclude that there might be 'obviously easy' string constraints that still cannot be handled by our approach. Future work therefore requires to conduct more extensive case studies and might require the incorporation of further rules or heuristics to optimize the backtracking for such cases.

Second, EMFtoCSP might perform bad even for models with tractable string constraints, when the non-String related constraints cannot be efficiently handled. Our scalability tests intentionally focused on models that are structurally tractable for EMFtoCSP, in order to measure to analyze the performance of the string solver.

7 Related Work

The community has developed several approaches and tools for automated solving for OCL-annotated models. 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 [7], relational logic [2,19,20], Boolean satisfiability [28], genetic algorithms [1], graph grammars [31,8], logic programming [24] and CLP [5].

All of these works support a more or less extensive subset of OCL (e.g., including quantifiers and collections), but, to our knowledge, only Kuhlmann et al. [19] provide real support for string operations. Their work is based on Kodkod (the relational solver behind Alloy). Since their implementation of the string operations has not been available at the time of writing, we could not evaluate the performance of their encoding on large instances. However, since it uses an (index, character) relation to encode strings, it very closely resembles the representation of sequences in Alloy, with the exception that they furthermore include support for the undefined string, which we do not consider. From the perspective of performance and scalability, their approach should thus behave either similarly or worse (because of the unfolding of OCL's multi-valued logic) as Alloy on our examples. The work of Ali et al. [1] considers strings, too, but the approach, which is based on genetic algorithms, is not exhaustive.

Outside of MDE, reasoning on strings has been performed in various formalisms, both for bounded and unbounded strings. Several solvers for Satisfiability Modulo Theories (SMT) support theories that can be used to represent strings, such as arrays and bit-vectors. For example, Bjorner et al. perform path analysis for String-manipulating programs using SMT [3]. In addition to the theory-based works, several approaches reason about string constraints using finite automata, e.g., [18,14,15,11,30]. These approaches are much stronger than our solver in exhaustively checking even NP-hard string constraints (e.g., using symbolic reasoning). But, to our knowledge, they have not been applied to model instantiation, where string constraints are only one, simple, part of the overall problem (but with a potentially large number of string variables).

We expect that our string solver can be ported in a straightforward manner to other CLP environments that support CHR and finite domain constraints, too, such as SWI-Prolog and SICStus Prolog do. In general, our handling rules could be implemented using lower level CLP concepts, such as suspended goals and meta-attributes, too, using the CHR rules as specification. We expect that our constraint handler can also be integrated into other CLP-based model finders such as [22,21, 6].

8 Conclusion

In the context of OCL-annotated models, systematic approaches ('model finders') are required to check their satisfiability (in verification activities) and to generate instances of them (in, e.g., testing and validation activities). While both checking satisfiability and generating instances are very similar to each other from a theoretical perspective, the practical requirements are different. In general, model verification calls for exhaustive exploration of the search space, and for support to check even intractable constraints in a reasonable time (given the computational complexity of the problem). On the other hand, model-based testing and validation often do not pose intractable string constraints but call for sufficiently large instances.

When adding support for OCL well-formedness rules to model finding, the tension between both requirements becomes even more pronounced: On the one hand, even simple theories of strings are already NP-hard, and on the other hand, strings

of reasonable length introduce a very large number of variables (when viewed on the element level).

We have presented a CLP-based solver, defined using CHR, that is suited to efficiently handle large, lightweight string problems. It can be combined seamlessly with other constraint solvers. Our constraints are suited to directly encode the operations of the main OCL string operations. Using our solver, we have implemented an extension of the EMFtoCSP model finder that now supports OCL string operations.

Our implementation scales better than our reference, the popular relational model finder Alloy, on tractable constraints that our approach can handle by propagation. It provide a way to automatically instantiate such models (e.g., as test cases) on a scale that cannot be handled by a simple relational/SAT encoding. On intractable constraints, it performs worse. Thus, it complements these approaches for model finding and closes a gap in model finding.

References

1. S. Ali, M. Z. Z. Iqbal, A. Arcuri, and L. C. Briand. A Search-Based OCL Constraint Solver for Model-Based Test Data Generation. In *QSIC*, pages 41–50, 2011.
2. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.
3. N. Bjørner, N. Tillmann, and A. Voronkov. Path Feasibility Analysis for String-Manipulating Programs. In S. Kowalewski and A. Philippou, editors, *TACAS 2009*, volume 5505 of *LNCS*, pages 307–321. Springer, 2009.
4. F. Büttner and J. Cabot. Lightweight String Reasoning for OCL. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. S. Kolovos, editors, *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Proceedings*, volume 7349 of *LNCS*, pages 244–258. Springer, 2012.
5. J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *Automated Software Engineering, ASE 2007, Proceedings*. ACM, 2007.
6. M. Cadoli, D. Calvanese, G. De Giacomo, and T. Mancini. Finite Satisfiability of UML Class Diagrams by Constraint Programming. In *In Proc. of the CP 2004 Workshop on CSP Techniques with Immediate Application*, 2004.
7. M. Clavel, M. Egea, and M. A. G. de Dios. Checking Unsatisfiability for OCL Constraints. *Electronic Communications of the EASST*, 24:1–13, 2009.
8. K. Ehrig, J. Küster, and G. Taentzer. Generating instance models from meta models. *Software and Systems Modeling*, 8:479–500, 2009.
9. T. W. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming*, volume 910 of *LNCS*, pages 90–107, 1994.
10. T. W. Frühwirth. Constraint handling rules: the story so far. In A. Bossi and M. J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), July 10-12, 2006, Venice, Italy*, pages 13–14. ACM, 2006.
11. K. Golden and W. Pang. Constraint Reasoning over Strings. In F. Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003*, volume 2833 of *LNCS*, 2003.

12. C. A. González Pérez, F. Buettner, R. Clarisó, and J. Cabot. EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In *Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*, Zurich, Switzerland, 2012.
13. C. A. González, F. Büttner, R. Clarisó, and J. Cabot. 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.*, 2012.
14. P. Hooimeijer and M. Veanes. An Evaluation of Automata Algorithms for String Analysis. In R. Jhala and D. A. Schmidt, editors, *VMCAI*, volume 6538 of *LNCS*, pages 248–262, 2011.
15. P. Hooimeijer and W. Weimer. StrSolve: solving string constraints lazily. *Autom. Softw. Eng.*, 19(4):531–559, 2012.
16. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
17. S. Jha, S. A. Seshia, and R. Limaye. On the Computational Complexity of Satisfiability Solving for String Theories. *CoRR*, abs/0903.2825:1–11, 2009.
18. A. Kiezun, V. Ganesh, S. Artzi, P. H. Philip Guo, and M. Ernst. HAMPI: A Solver for Word Equations over Strings, Regular Expressions and Context-Free Grammars. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(4), 2012.
19. M. Kuhlmann and M. Gogolla. From UML and OCL to Relational Logic and Back. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 415–431. Springer, 2012.
20. M. Kuhlmann, L. Hamann, and M. Gogolla. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In J. Bishop and A. Vallecillo, editors, *TOOLS 201*, volume 6705 of *LNCS*, pages 290–306. Springer, 2011.
21. H. Malgouyres and G. Motet. A UML model consistency verification approach based on meta-modeling formalization. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1804–1809, New York, NY, USA, 2006. ACM.
22. A. Maraee and M. Balaban. 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*, ECMDA-FA'07, pages 17–31, Berlin, Heidelberg, 2007. Springer-Verlag.
23. OMG. *Object Constraint Language Specification, version 2.3.1 (Document formal/2012-01-01)*, 2012.
24. A. Queralt and E. Teniente. Verification and Validation of UML Conceptual Schemas with OCL Constraints. *ACM Trans. Softw. Eng. Methodol.*, 21(2):13, 2012.
25. F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier Science, 2006.
26. J. Schimpf and K. Shen. ECLiPSe – From LP to CLP. *Theory and Practice of Logic Programming*, 12:127–156, 2012.
27. J. Sneyers, P. V. Weert, T. Schrijvers, and L. D. Koninck. As time goes by: Constraint Handling Rules. *TPLP*, 10(1):1–47, 2010.
28. M. Soeken, R. Wille, and R. Drechsler. Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In M. Gogolla and B. Wolff, editors, *TAP 2011*, volume 6706 of *LNCS*, pages 152–170. Springer, 2011.
29. E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 -*

April 1, 2007, Proceedings, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

30. M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *ICST*, pages 498–507. IEEE Computer Society, 2010.
31. J. Winkelmann, G. Taentzer, K. Ehrig, and J. M. Küster. 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.