Partial Evaluation of OCL Expressions

Bastian Ulke
Fernuniversität in Hagen, Germany
bastian.ulke@gmail.com

Friedrich Steimann
Fernuniversität in Hagen, Germany
steimann@acm.org

Ralf Lämmel
Universität Koblenz-Landau, Germany
rlaemmel@acm.org

***Abstract*—In the academic literature, many uses of the Object Constraint Language (OCL) have been proposed. By contrast, the utilization of OCL in contemporary modelling tools lags behind, suggesting that leverage of OCL remains limited in practice. We consider this undeserved, and present a scheme for partially evaluating OCL expressions that allows one to capitalize on given OCL specifications for a wide array of purposes using a single implementation: a partial evaluator of OCL.**

I. Introduction

The Object Constraint Language (OCL) [1] is a specification language designed to be usable by stakeholders with only little formal background. Using OCL, assertions (invariants, preconditions, and postconditions) and queries can be expressed in a form familiar to programmers acquainted with Smalltalk and other object-functional programming languages. Being a standard of the Object Management Group (OMG), OCL is heavily used in other OMG standards, most prominently for specifying the semantics of the Meta Object Facility (MOF) [2] and the Unified Modeling Language (UML) [3]; but also for specifying the semantics of OCL itself [1].

Most early OCL tools focused on *evaluating* OCL expressions, e.g. to check the validity, or well-formedness, of a model instance with respect to the constraints associated with its model. Much to our surprise, however, we found that none of the OCL implementations we investigated[1] use OCL to check the well-formedness of the OCL expressions they evaluate, despite the fact that this well-formedness is specified using OCL [1]. Given that meta-circular bootstrapping is commonly regarded as the High Mass of language implementation, we find this remarkable.

One might argue that the programming-language like syntax of OCL makes it easy to manually translate OCL expressions to expressions of the host language of a modelling tool (e.g., Java), obviating the need for embedding an OCL interpreter in the tool. However, this holds only for *evaluating* OCL expressions, as required for checking the validity of model instances. For most of OCL's other uses, which include

- checking consistency of a specification (by finding an instance satisfying it; "strong satisfiability" or "model finding") [5], [6], [7], [8], [9] or, more generally, verification [10], [11];
- identifying redundancy (i.e., constraints being implied by others so that they can be removed) [7],
- validation [7] or completion [12] of partial models,
- test data and test case generation [13], [14], and
- model refactoring [15] and repair [12],

this is not sufficient: rather, OCL expressions need to be translated to other formalisms that come with powerful reasoning mechanisms, usually some kind of logic (see Section III for examples).

However, even if we abstract from the differences in logic used by each particular work (which we will), we note that the different uses of OCL still require different translations:

- *model finding* and *test data generation* require the translation of an OCL specification and the model it is associated with to a maximally variable representation in which neither classifier instances nor their properties are known in advance;
- *model completion*[2] requires a fixed representation of the current model with variables in the places to be completed;
- *model repair* requires identifying fixed parts of a model that make it malformed, and replacing them with variables whose sought values make the model well-formed; and
- *model refactoring* requires the propagation of (local) changes through an otherwise fixed model so that neither its well-formedness nor its semantics change.

In addition, the implementer of each translator must make some effort to show that translation is consistent with evaluation, so that all found, completed, repaired, or refactored models (or instances) are valid as judged by an OCL evaluator. This is not self-evident if the OCL evaluator and the OCL translator are different programs.

With our work presented here, we aim to contribute to the more widespread use of OCL in the modeller's toolchain. Specifically, we make the following contributions:

- We present a systematic approach towards developing a partial evaluator for OCL that can be used for model validation, model finding, model completion, model repair, and other possible use cases.

[1] the *UML Specification Environment* USE (https://sourceforge.net/projects/useocl/) [4], the *Dresden OCL Toolkit* (http://www.dresden-ocl.org), and the implementation of the Eclipse Model Development Tools (https://projects.eclipse.org/projects/modeling.mdt.ocl), the reference implementation of OCL

[2] Model completion, as well as model refactoring and model repair, are actually applied to models as instances of meta-models, on which the invariants are defined. However, our work is completely oblivious to the level it operates on.

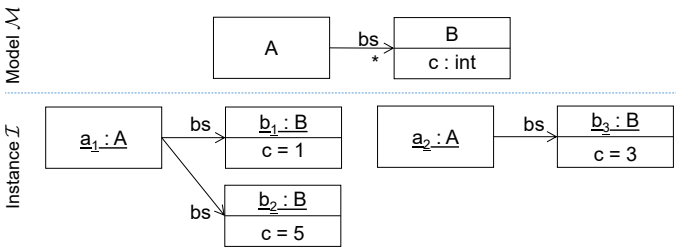

Fig. 1. Model $\mathcal{M}$ and instance $\mathcal{I}$ used as a running example.

- We present a Java implementation of our OCL partial evaluator that is built upon the Eclipse Model Development Tools (MDT)[3] infrastructure.
- We present results obtained from applying our implementation to 14 non-trivial (with up to one quarter of a million model elements) open-source models for the purposes of model completion and model repair, showing that it is indeed practically feasible.

Our approach relies on the adaptation of existing recipes in the field of partial evaluation [16], [17] to the situation of OCL. A particular challenge is the handling of variable path expressions that are common in OCL, but absent from the usual target formalisms of OCL translation. Our formal development has been mechanized in Haskell and is available online.[4]

The remainder of this paper is organized as follows. In Section II, we provide an overview of the problem, setting the stage also for the related work discussed in Section III. In Section IV, we briefly review regular evaluation of OCL expressions (using big-step operational semantics), before we develop its translation into a target formalism (an intermediate constraint language) by means of symbolic evaluation (Section V). In Section VI, we show how evaluation is intertwined with translation, giving us our desired partial evaluator. Section VII is dedicated entirely to the special problem of handling variable path expressions in the context of partial evaluation as developed thus far. Section VIII briefly sketches our Java implementation, which serves the evaluation in Section IX. Notes on lessons learnt and future work conclude.

NB: This work is not on formalizing OCL. Instead, it is on using one formalization (formal semantics) of OCL to suit different purposes, without requiring different implementations for this. The presented coverage of OCL is not complete; our partial evaluation, however, is orthogonal to extensions in that it offers a scheme that can likewise be applied to formalizations that cover OCL more completely.

II. Problem Statement

We state the problem we are addressing by means of a small, artificial example. It is comprised of the OCL expression

$$\textbf{context}\ A\ \textbf{inv}\ I : \textbf{self}.bs{->}\textbf{exists}(b|b.c < 5) \qquad (1)$$

which is formulated with reference to the model $\mathcal{M}$ shown in Figure 1. Using this example, we can identify the following problems and solutions:

- An OCL *interpreter* accepting (1) and the model instance $\mathcal{I}$ also shown in Figure 1 as input[5] will produce the value $true$, by iterating over the instances of A (a_1 and a_2) and the values of $a_1.bs$ (b_1 and b_2) and $a_2.bs$ (b_3), resp., aggregating the truth values of the conditions $b_i.c < 5$ as

  $$(1 < 5 \vee 5 < 5) \wedge (3 < 5) \equiv true$$

  The mechanical basis of the derivation of this result will be delivered in Section IV.
- A *model finder* will use the model $\mathcal{M}$ of Figure 1 and the invariant (1) to generate a constraint satisfaction problem (CSP, not shown) solved by the assignment

  $$[A] := \{o_1^A\}, [B] := \{o_1^B\}, o_1^A.bs := \{o_1^B\}, o_1^B.c := 0$$

  where $[C]$ denotes the extension (set of instances) of classifier C and o_1^C denotes a (first) anonymous instance (object) of C. The mechanical basis of the derivation of this result will be delivered in Section V; note that a model finder will eventually (via backtracking) also find instance $\mathcal{I}$ of Figure 1, but only if the domain of the property c is bounded. In fact, to ensure that the model finder always terminates, $[C]$ also needs to be bounded for all classifiers C ("bounded verification" [7], [18]). Model finding is also relevant in the context of test-data generation on models that involve constraints [19].
- To see how (1) is used for *model completion*, assume that we have $\mathcal{I}'$ identical to $\mathcal{I}$ of Figure 1, except that $a_2.bs$ has not yet been assigned any of b_1, b_2, or b_3. Clearly, $\mathcal{I}'$, as far as it has been specified, is well-formed according to invariant (1), but not all of its conceivable completions are. An automatic model completion tool will provide only the appropriate choices, which can be derived by finding all values for $a_2.bs$ in $2^{[B]}$ that satisfy the constraint $\exists b \in a_2.bs : b.c < 5$, that is, by solving $b_1 \in a_2.bs \vee b_3 \in a_2.bs$ for the variable $a_2.bs$ (b_2 is ruled out because $b_2.c \not< 5$). The mechanics of this derivation will be delivered in Section VI; model completion will be used to demonstrate the capabilities of our partial evaluator of OCL expressions in Section IX.
- For the purpose of *model repair*, assume we have a model $\mathcal{I}''$ identical to $\mathcal{I}$ except that $a_1.bs = \{b_2\}$ (check that this renders $\mathcal{I}''$ malformed with respect to (1)). An automatic model repair tool would offer changing $b_2.c$ to an integer less than 5, which can be computed by solving the constraint $b_2.c < 5$ for the variable $b_2.c$. If this should prove unsatisfactory, the tool can further offer replacing b_2 in $a_1.bs$ with either b_1 or b_3, or adding at least one of the two to $a_1.bs$. As above, these corrections can be computed by solving the constraint $\exists b \in a_1.bs : b.c < 5$ for the variable $a_1.bs$; even greater flexibility is granted

[3] http://www.eclipse.org/modeling/mdt/

[4] https://github.com/softlang/yas/tree/master/languages/BOL/Haskell/

[5] Actually, an OCL interpreter will typically also consume $\mathcal{M}$, to check the well-formedness of expression (1).

by allowing new values for $b_1.c, ..., b_3.c$, too, which amounts to solving

$$\bigvee_{i=1,2,3} b_i \in a_1.bs \wedge b_i.c < 5$$

(but note that any of these changes may be subject to other constraints, which need to be considered by the repair). The mechanics of this derivation will also be delivered in Section VI, and model repair will also be performed in Section IX.

- For the purpose of refactoring, we refer the reader to our earlier work [15], which shows how well-formedness conditions can be lifted to conditions sufficient for behaviour preservation. For this, our running example is unsuited, since we cannot associate any meaning to be preserved with instance $\mathcal{I}$ of Figure 1.

Hence, we have that, depending on the specific use, the OCL expression (1) is rendered to widely differing forms which, as we will see in Section III, are currently obtained using very different implementations. What we will deliver in this paper is a single OCL treatment that covers all uses uniformly. This treatment will be based on partial evaluation.

III. Related Work

Cabot and Gogolla provide an excellent introduction to OCL and its many uses [20]. However, despite its undeniable versatility, [18] finds that OCL tools are currently not integrated into the modeller's tool chain.

Mapping OCL to other formalisms: As noted in the introduction, different works use different target formalisms, notably propositional (as in, e.g., [21]), sorted (e.g., [8]), description (e.g., [22]), first- (e.g., [6], [23], [24]), or higher-order logic (e.g., [10], [25]); or to constraint-logic programs (e.g., [7]). While such mapping is, for most parts, straightforward, the two features requiring special attention are the treatment of navigation (or path) expressions (which includes the problem of properties having $[0, 1]$ and *, rather than $[1, 1]$, multiplicities), and the handling of uncertainty, amounting (since OCL 2.4) to a four-valued logic [8]. Different multiplicities can be handled alike by a mapping that treats scalars as singleton sets and hence unifies the representation of $[0, 1]$, $[1, 1]$, and * (such as the relational logic of Alloy; but see [23], [26] for the problems that this causes). Adding four-valued logic to address uncertainty complicates the mapping to standard logic (which is required by standard reasoners), but is otherwise orthogonal to partial evaluation as we propose. A further challenge arises when property values are variable, thereby requiring special handling of chained property access. If the target formalism does not support indirection through unknowns, designated case analyses are to be introduced [15], which complicate matters considerably (we dedicate the entire of Section VII to this problem).

Verification: The use of OCL in verification of models has been systematically reviewed in [18]. One finding is that performance of verification tools drops dramatically when the models get larger. This, of course, is owing to the exponential nature of the underlying problem, and can only be fundamentally improved by giving up completeness or reducing the degree of variability in the sought models. The reduction of the combinatorial explosion plaguing translations from OCL to SAT problems (as undertaken for model finding) when some variable values are known in advance (specifically: when the metamodel marks some properties as unchanging for all instances), has also been addressed by recent work [9], but the treatment remains cursory (although some similarities can be identified, e.g., the use of shortcut optimizations; see Section VI). Cabot et al. have shown how OCL expressions can be translated to constraint-logic programs [7]; list variables of the target language are used to model the extensions of classes and associations where the number of elements in each list is bounded to ensure termination. By contrast, we cast the transformation of OCL expressions to first-order formulae in terms of partial evaluation, without tying it to a specific target formalism (such as SAT or SMT), let alone a concrete implementation.

Partial evaluation: A partial evaluator (or program specializer) is a metaprogram which processes a program and its known input (the so-called *static variables*), and generates a specialized program (the so-called *residual program*) parameterized in the remaining input (the so-called *dynamic variables*). In the residual program, computations on just static variables are eliminated [16], [17], [27], [28], [29]. For instance, a partial evaluator presented with the program `if x>y then (1+y)/x else y` and static input `0` for `y` produces the residual program `if x>0 then 1/x else 0`.An offline specializer performs an extra binding-time analysis to distinguish eliminable versus residual constructs [29]. An online specializer makes decisions in a single pass. Our development uses the flexible online scheme, as we did not encounter any termination issues that would call for offline partial evaluation. A special use case of partial evaluation is to specialize an interpreter by a program, thereby serving compilation [16].

Transferring the idea of partial evaluation to OCL would mean that some or all instances (objects) and some or all of their properties are 'dynamic'. While a regular OCL evaluator would return a Boolean value for checking an invariant, a partial evaluator would thus return a formula over appropriate instance and property variables. A central aspect of OCL specialization concerns iteration over sets (or collections; including quantification), which is similar to loop unrolling in program specialization [27]. A specific challenge of OCL is chained property access (in so-called path expressions), informal treatise of which we find in [9], and which we treat in terms of partial evaluation in Section VII. Partial evaluation for OCL has already been studied in a models@runtime context by Song et al. [30] where specialization helps with making constraints of adaptation policies more manageable for SMT use at runtime, without though exercising the scale of validation, finding, and completion with application to a larger-scale problem. The simplification of OCL constraints, with focus on redundancy, when constraints are instantiated from templates,

has also been addressed by techniques other than partial evaluation—in particular, by rule-based approaches [31], [32]. Partial evaluation combined with simple rule-based optimization directly provides an efficient model validation, finding, and completion approach, as demonstrated by our implementation (Section VIII) and case study (Section IX).

Previous work by the authors: In our own prior works ([12], [15]), we showed how first-order logic specifications paired with constraint solving can be used for model completion, model refactoring, and model repair. All three uses rely heavily on the assumption that most properties of models are fixed (static), and only few are variable (dynamic). However, constraint generation was delegated to Refacola, an infrastructure developed for the constraint-based refactoring of programs [33]. Refacola comes with its own constraint language (so-called constraint rules, which have been used for constraint-based refactoring from its beginnings [34] and which are vastly different from OCL invariants) and uses elaborate algorithms for generating only the constraints required. In the present paper, we use OCL as our specification language and we formalize constraint generation within the framework of partial evaluation, thereby arriving at a systematic generation process profiting from a long line of research done in the context of compiler optimization. An early draft of a rule-based translation process of OCL constraints into constraint-satisfaction problems was presented at an OCL workshop [35], without use of partial evaluation, though.

The evaluation scenario used in this paper (Section IX) also served the first two authors' recent work on repairing malformed programs using constraint attribute grammars [36]. Attribute grammars are vaguely related to OCL expressions in that they express equations (constraints) on attributes attached to the nodes of a syntax tree, where the nodes correspond to instances of the classifiers of a model, and the attributes correspond to properties attached to the instances. However, [36] neither used OCL, nor did it utilize partial evaluation to generate the constraints needed.

IV. Regular Evaluation

We begin the development of the partial evaluator for OCL with departing from a regular evaluator. For reasons of brevity and understandability, we limit the formal development here to the OCL fragment of Figure 3; our mechanized Haskell development covers a larger OCL subset; our implementation (Section VIII) covers OCL up to the point needed in the case study.

Inspired by [37], we specify regular evaluation for OCL by a big-step operational semantics; see the deduction rules in Figure 2. OCL expressions like (1) are evaluated in the context of an environment E which is actually of the form $\langle E_{\mathcal{I}}, E_{\mathcal{P}}, E_{\mathcal{V}} \rangle$, i.e., there are three function components: $E_{\mathcal{I}}(C)$ for the extensions of classifiers C, $[C]$ (i.e., C's instances, a set of objects); $E_{\mathcal{P}}(o, p)$ for the values of properties p of objects o; and $E_{\mathcal{V}}(x)$ for the values of OCL variables x, i.e., iteration variables and **self**.

Applied to invariant (1) and $\mathcal{I}$ of Figure 1, the operational semantics assigns $true$, meaning that $\mathcal{I}$ is indeed well-formed.

V. Symbolic Evaluation

The initial assumption of setting up partial evaluation for OCL is that the environment components $E_{\mathcal{I}}$ and $E_{\mathcal{P}}$ map classifiers C and properties of objects $o.p$ to unknowns, that is, to designated *dynamic variables* with appropriate domains, rather than to sets of instances and values, resp., as they did for regular evaluation. We write $[\![C]\!]$ for the variable to which C, and $[\![o.p]\!]$ for the variable to which $o.p$, is mapped. We use the term "dynamic variable" here in the sense of partial evaluation as a variable in a program (expression) that is not known at specialization time. By contrast, the OCL variables mapped by $E_{\mathcal{V}}$ correspond to static variables; they are known at specialization time. Specifically, the environment $E_{\mathcal{V}}$ holds OCL variables (iteration variables and **self**), just as in the case of regular evaluation.

Because of the required boundedness, we assume for each classifier C from $\mathcal{M}$ a finite repository $[C] = \{o_1^C, ..., o_n^C\}$ of instances; $E_{\mathcal{I}}$ thus maps C to a variable whose domain is $2^{[C]}$. Such variables parameterize over the instances of classifiers constituting a sought model $\mathcal{I}$. Analogously, since we do not know the values of properties, either, $E_{\mathcal{P}}$ maps a property $o_i^C.p$ to a variable with a domain given by $\mathcal{M}$, i.e., the domain of p as specified by its declaration in classifier C. We write $[\psi]$ for the domain (set of all possible values) of a variable ψ, and additionally for many-valued variables, $\lfloor \psi \rfloor$ for the base domain (the potential members of the set that is the value of ψ). For instance, we write $[[\![a_1.bs]\!]] = 2^{[B]}$ for the domain of the variable representing $a_1.bs$, and $\lfloor [\![a_1.bs]\!] \rfloor = [B]$ for the base domain of ψ.

Because of the introduced variables, the computations performed by the regular evaluator specified in Figure 2 are no longer feasible. For instance, as $o.c$ maps to a variable, we cannot compute a truth value for $o.c < 5$. Thus, we need to specify a *symbolic evaluator* which computes terms τ rather than values for OCL expressions and formulae Φ rather than truth values for OCL formulae (invariants). The syntax of the resulting formulae and terms is described in Figure 4; we refer to the corresponding language as our *Intermediate Constraint Language* (ICL).

ICL is an artefact of OCL evaluation in the presence of variables, as opposed to an arbitrary choice made by these authors. That is, ICL has constructs for the evaluation or symbolic representation of various OCL constructs, e.g., $<$, $\wedge$, and $\vee$. OCL's quantifiers in **inv** and **exists** are eliminated with the help of $\in$, $\Rightarrow$, and $=$ (which also have correspondences in OCL). *ICL is thus a quantifier-free, first order logic with theories* (integer, set), which is close to the formalisms used by various technologies serving the different uses of OCL (cf. Introduction and Section III). For the use of a particular technology (a specific constraint or SMT solver), ICL expressions may still need to be mapped to the corresponding input languages, but compared to mapping OCL to these languages, this mapping is a very small one.

$$\frac{(\langle E_{\mathcal{I}}, E_{\mathcal{P}}, E_{\mathcal{V}}[\textbf{self} \mapsto o]\rangle \vdash \phi \downarrow b_o)_{o \in E_{\mathcal{I}}(C)}}{E \vdash \boxed{\textbf{context}\ C\ \textbf{inv}\ I : \phi} \downarrow \bigwedge_{o \in E_{\mathcal{I}}(C)} b_o}\ \text{(E-INV)}$$

$$\frac{E \vdash e \downarrow \{o_1, ..., o_n\} \quad (\langle E_{\mathcal{I}}, E_{\mathcal{P}}, E_{\mathcal{V}}[x \mapsto o_i]\rangle \vdash \phi \downarrow b_i)_{1 \le i \le n}}{E \vdash \boxed{e\text{->}\textbf{exists}(x|\phi)} \downarrow b_1 \vee ... \vee b_n}\ \text{(E-EXISTS)} \qquad E \vdash \boxed{x} \downarrow E_{\mathcal{V}}(x)\ \text{(E-VARACC)}$$

$$E \vdash \boxed{\textbf{self}} \downarrow E_{\mathcal{V}}(\textbf{self})\ \text{(E-SELF)} \qquad \frac{E \vdash e \downarrow o}{E \vdash \boxed{e.p} \downarrow E_{\mathcal{P}}(o, p)}\ \text{(E-PROPACC)} \qquad \frac{E \vdash e_l \downarrow v_l \quad E \vdash e_r \downarrow v_r}{E \vdash \boxed{e_l < e_r} \downarrow v_l < v_r}\ \text{(E-LT)}$$

Fig. 2. Rules of a regular OCL evaluator (excerpt). v represents values, o objects (in fact, object ids) as a form of values, and b boolean values. Note that expressions on the right of $\downarrow$ are expressions over the semantic domains; e.g., '$\wedge$' means Boolean conjunction and '$<$' means integer less-than. Evaluation of these expressions is taken for granted. Also, note that we assume set-based semantics here for simplicity. In particular, the result of evaluating e in e->**exists**$(x|\phi)$ is viewed as a set (as usual, receiver expressions e with multiplicity $[0, 1]$ or $[1, 1]$ are implicitly coerced to sets). The treatment of aggregates would additionally require list-based semantics; see, e.g., [26] for a treatment of OCL's collection types in the context of logic.

Inv	::=		**invariants:**
		context C **inv** $I : \phi$	context condition
ϕ	::=		**formulae:**
		e->**exists**$(x\|\phi)$	exists
	|	$e < e$	less than
e	::=		**expressions:**
	|	$e.p$	property access
	|	x | **self**	variable access
	|	l	integer literal

Fig. 3. OCL syntax (excerpt)

Φ	::=		**formulae:**
		b	Boolean values
	|	$\tau < \tau$	less than
	|	$\neg\ \Phi$	negation
	|	$\Phi \wedge \Phi$	conjunction
	|	$\Phi \vee \Phi$	disjunction
	|	$\Phi \Rightarrow \Phi$	implication
	|	$o \in \psi$	membership constraint
	|	$\psi = o$	equality constraint
τ	::=		**terms:**
		l	integer literal
	|	o	object (id)
	|	ψ	dynamic variable

Fig. 4. The first-order logic (ICL) required for symbolic evaluation.

We specify symbolic evaluation for OCL by translation semantics mapping OCL constraints to ICL; see the rules in Figure 5. Rules (T-INV) and (T-EXISTS) state that symbolic evaluation unrolls quantified expressions without though assuming that the underlying sets are known, only assuming them to be bounded. Note how the rules for symbolic evaluation parallel those for regular evaluation (Figure 2); the main differences are that the sets of instances iterators range over are taken from the repository of potential instances with guards added to check their membership in variables with a power-set domain: In the case of **context** C **inv** $I : \phi$, the instances are drawn from the repository $[C]$; in the case of e->**exists**$(x|\phi)$, the instances are drawn from $\lfloor\psi\rfloor$, the potential members of the value of ψ, where ψ is the variable to which e is evaluated.

Applied to invariant (1) and the model $\mathcal{M}$ of Figure 1 with $|[A]| = 2$ and $|[B]| = 3$, we get the formula

$$\bigwedge_{i=1,2} \left(o_i^A \in [\![A]\!] \Rightarrow \bigvee_{j=1,2,3} o_j^B \in [\![o_i^A.bs]\!] \wedge [\![o_j^B.c]\!] < 5 \right) \quad (2)$$

Note that all quantifications have been unrolled; however, unlike for evaluation, they are not unrolled over sets derived by prior (recursive) evaluation, but over the full domain of variables (**self** and x in our example). For **inv** (corresponding to universal quantification), this means that each conjunct must be guarded with membership in the set of instances of the context classifier (which is a variable; $o \in E_{\mathcal{I}}(C) \Rightarrow \Phi_o$ in (T-INV)). For **exists**, this means that each disjunct must be conjoined with a membership constraint ($o \in \psi$ in (T-EXISTS)), meaning that the quantified constraint is applied only in those cases in which the object turns out to be a member of the set according to a variable assignment.

An ICL formula with the assumed domains for the involved variables defines a *constraint satisfaction problem* (CSP) [38]. A constraint or SMT solver can thus compute solutions for ICL expressions such as (2), of which $\mathcal{I}$ of Figure 1 is one.

Note that, because the partial evaluatoris obtained by a number of small-scope revisions on the regular interpreter, we can rest confident that it is soundwith respect to regular evaluation: assuming compositionality of the definitions, we are permitted to reason modularly about correctness, e.g., of the shortcut optimizations introduced next.

VI. PARTIAL EVALUATION

In a technical sense, the symbolic evaluator, as developed thus far, is already a degenerated partial evaluator, as it caters for dynamic variables ($E_{\mathcal{I}}$ and $E_{\mathcal{P}}$) and it constructs residual formulae according to the ICL syntax. The symbolic evaluator

$$\frac{(\langle E_{\mathcal{I}}, E_{\mathcal{P}}, E_{\mathcal{V}}[\textbf{self} \mapsto o]\rangle \vdash \phi \longrightarrow \Phi_o)_{o \in [C]}}{E \vdash \boxed{\textbf{context}\ C\ \textbf{inv}\ I : \phi} \longrightarrow \bigwedge_{o \in [C]} o \in E_{\mathcal{I}}(C) \Rightarrow \Phi_o} \quad \text{(T-INV)}$$

$$\frac{E \vdash e \longrightarrow \psi \quad (\langle E_{\mathcal{I}}, E_{\mathcal{P}}, E_{\mathcal{V}}[x \mapsto o]\rangle \vdash \phi \longrightarrow \Phi_o)_{o \in \lfloor \psi \rfloor}}{E \vdash \boxed{e\text{->}\textbf{exists}(x|\phi)} \longrightarrow \bigvee_{o \in \lfloor \psi \rfloor} o \in \psi \wedge \Phi_o} \quad \text{(T-EXISTS)}$$

$$E \vdash \boxed{x} \longrightarrow E_{\mathcal{V}}(x) \ \text{(T-VARACC)}$$

$$E \vdash \boxed{\textbf{self}} \longrightarrow E_{\mathcal{V}}(\textbf{self}) \ \text{(T-SELF)} \qquad \frac{E \vdash e \longrightarrow o}{E \vdash \boxed{e.p} \longrightarrow E_{\mathcal{P}}(o, p)} \ \text{(T-PROPACC)} \qquad \frac{E \vdash e_l \longrightarrow \tau_l \quad E \vdash e_r \longrightarrow \tau_r}{E \vdash \boxed{e_l < e_r} \longrightarrow \tau_l < \tau_r} \quad \text{(T-LT)}$$

Fig. 5. Rules of a symbolic OCL evaluator (excerpt). Note that expressions on the right of $\longrightarrow$ are symbolic computations, i.e., terms and formulae constructed by the evaluator. Unlike for Figure 2, symbolic computations are not (immediately) evaluated. Note that the notation is overloaded. In particular, we use '$<$' for OCL's comparison operator, the assumed operation on integer values (in Figure 2), and the symbolic operation on terms as a form of ICL formula (in this figure). For (T-EXISTS), the remarks on (E-EXISTS) apply accordingly. Finally, note that (T-PROPACC) makes a simplifying assumption that the receiver expression of property access evaluates to an object; we lift this restriction in Section VII (and add the case that e evaluates to many objects in Section IX).

makes no effort however to perform actual computations, where possible. Let us now derive a useful (proper) partial evaluator.

A partial evaluator should use result types that subsume those of a regular evaluator and extend them to permit generation of residuals. We note that the regular evaluator maps expressions to either integers l or objects o. These cases are covered by ICL's terms τ. Further, the regular evaluator maps OCL formulae ϕ to Boolean values b; these cases are covered by ICL's formulae Φ.

In this manner, we are ready to match on operands that are values and to compute results that are values, while we fall back to the symbolic cases where necessary. Thus, we submit that all operator symbols used in deduction rules are interpreted to behave as in the case of the regular evaluator whenever possible. For instance:

$$\tau_1 < \tau_2 \ = \ \begin{cases} l_1 < l_2 & \text{if } \tau_1 = l_1 \text{ and } \tau_2 = l_2 \\ \tau_1 <_{ICL} \tau_2 & \text{otherwise} \end{cases}$$

Here, we use the ICL subscript for disambiguation to express that the symbol $<$ is taken from ICL syntax. In the same manner, we also set up interpretations of conjunction, disjunction, and implication. These interpretations also incorporate common *shortcut* rules, thereby facilitating *optimization* of residual constraints. For instance, conjunction is interpreted on ICL's formulae as follows:

$$\Phi_1 \wedge \Phi_2 \ = \ \begin{cases} \mathit{false} & \text{if } \Phi_1 = \mathit{false} \text{ or } \Phi_2 = \mathit{false} \\ \Phi_2 & \text{else, if } \Phi_1 = \mathit{true} \\ \Phi_1 & \text{else, if } \Phi_2 = \mathit{true} \\ \Phi_1 \wedge_{ICL} \Phi_2 & \text{otherwise} \end{cases}$$

As a result, shortcut optimizations are performed along with partial evaluation as opposed to a separate optimization, thereby helping to keep the overall problem tractable with regard to the size of ICL representations, as we discuss in Section IX. Interpretations like those above, with shortcuts included, can be implemented as "smart constructors" in the partial evaluator, i.e., the optimizations are conducted right when the terms are constructed.

To arrive at a useful partial evaluator for OCL, we also need to mix the environments for regular and symbolic evaluation as follows:

- We subdivide $E_{\mathcal{I}}$ further into $E_{\mathcal{I},\sigma}$ and $E_{\mathcal{I},\delta}$, where $E_{\mathcal{I},\sigma}(C)$ maps C to a set of known (static) instances (just like $E_{\mathcal{I}}$ in the regular evaluation of Section IV), and $E_{\mathcal{I},\delta}(C)$ maps C to a dynamic variable (just like $E_{\mathcal{I}}$ in Section V for translation).
- $E_{\mathcal{P}}$ maps each object-property pair $o.p$ to either a value v or a variable ψ, depending on whether $o.p$ is static or dynamic.
- $E_{\mathcal{V}}$ is the same as in Sections IV and V.

In this manner, we can deal with a *partially known model instance* (comprised of static and dynamic objects with static and dynamic properties), thereby addressing, for example, the use case of model completion. Regular evaluation would fail when the environment does not provide values for some relevant classifiers or properties. Symbolic evaluation would not fail in such cases; translation could, in theory, commence with assuming everything to be dynamic, subject to subsequent instantiation of the variables with known values (which may however be restrained in practice by memory limits; see Section IX). Partial evaluation uses known values upfront for computing smaller residual formulae.

The deduction rules for partial evaluation are provided in Figure 6. In fact, we only show rules for **inv** and **exists**, as the remaining rules are the same as in the case of symbolic evaluation (Figure 5), except that we assume a partially evaluating interpretation for all ICL constructs, as discussed above. Rule (PE-INV) splits up the conjunction for "all instances" so that there is a part for all "known instances" ($E_{\mathcal{I},\sigma}(C)$) and a part for "potential instances" ($[E_{\mathcal{I},\delta}(C)]$). Only the conjuncts in the second part need to be constrained by an extra membership constraint.

There are two rules for **exists**. Rule (PE-EXISTS1) deals

$$\frac{(\langle E_{\mathcal{I}}, E_{\mathcal{P}}, E_{\mathcal{V}}[\textbf{self} \mapsto o]\rangle \vdash \phi \searrow \Phi_o)_{o \in (E_{\mathcal{I},\sigma}(C) \cup [E_{\mathcal{I},\delta}(C)])}}{E \vdash \boxed{\textbf{context } C \textbf{ inv } I : \phi} \searrow \bigwedge_{o \in E_{\mathcal{I},\sigma}(C)} \Phi_o \wedge \bigwedge_{o \in [E_{\mathcal{I},\delta}(C))]} o \in E_{\mathcal{I},\delta}(C) \Rightarrow \Phi_o} \quad \text{(PE-INV)}$$

$$\frac{E \vdash e \searrow \{o_1, ..., o_n\} \quad (\langle E_{\mathcal{I}}, E_{\mathcal{P}}, E_{\mathcal{V}}[x \mapsto o_i]\rangle \vdash \phi \searrow \Phi_i)_{1 \leq i \leq n}}{E \vdash \boxed{e\text{->}\textbf{exists}(x|\phi)} \searrow \Phi_1 \vee ... \vee \Phi_n} \quad \text{(PE-EXISTS1)}$$

$$\frac{E \vdash e \searrow \psi \quad (\langle E_{\mathcal{I}}, E_{\mathcal{P}}, E_{\mathcal{V}}[x \mapsto o]\rangle \vdash \phi \searrow \Phi_o)_{o \in \lfloor \psi \rfloor}}{E \vdash \boxed{e\text{->}\textbf{exists}(x|\phi)} \searrow \bigvee_{o \in \lfloor \psi \rfloor} o \in \psi \wedge \Phi_o} \quad \text{(PE-EXISTS2)}$$

Fig. 6. Rules of a partial OCL evaluator (excerpt).

with the situation inherited from regular evaluation such that e in e->**exists**$(x|\phi)$ evaluates to a set of objects whereas rule (PE-EXISTS2) deals with the sitution that e evaluates to a variable ψ and thus, the operands of the disjunction have to be conjoined by membership constraints, as in the case of symbolic evaluation. Here, we iterate over the potential members of ψ, $\lfloor \psi \rfloor$.

For instance, in our running example, if all instances of A and B, as well as the properties bs for all instances of A, are fixed to the values given by $\mathcal{I}$ of Figure 1, while the properties c are variable for all instances of B, we get the residual formula

$$([\![b_1.c]\!] < 5 \vee [\![b_2.c]\!] < 5) \wedge ([\![b_3.c]\!] < 5)$$

which may be used for fixing model instances violating invariant (1) (see Section II) by changing the values of c. Alternatively, if the properties bs are variable and the properties c are fixed to the values of $\mathcal{I}$, we get the residual formula

$$(b_1 \in [\![a_1.bs]\!] \vee b_3 \in [\![a_1.bs]\!]) \wedge (b_1 \in [\![a_2.bs]\!] \vee b_3 \in [\![a_2.bs]\!])$$

(b_2 has been dropped from the disjunctions since $b_2.c \not< 5$) which may be used to fix problems by adding instances of B to bs (again, see Section II). Note that both residuals can be further reduced by fixing properties not involved in the problem, for instance $b_1.c$ and $b_3.c$, or $a_2.bs$.

VII. PATH EXPRESSIONS

In the discussion so far, we have glanced over one complication: expressions of the form $e.p$ where e evaluates to a variable. Rule (T-PROPACC) in Figure 5 did not yet admit this case, as we insisted that e evaluates to an object o. The situation with such a variable receiver expression occurs when we use chains of property accesss in so-called *path expressions* with variable properties, even when starting from a receiver that evaluates to an object. Consider, for example, the OCL expression $e.p.p'$ with e evaluating to an object o, p being a variable property, and p' being a known property. When evaluating $e.p$, we return the corresponding variable $E_{\mathcal{P}}(o,p) = \psi$. When evaluating $\psi.p'$, we need to unroll ψ and return a collection of results.

This is the essential change to be applied to the partial evaluator of Figure 6: expression evaluation returns a collection of ICL terms as opposed to a single term; all uses of expression evaluation need to be lifted to process collections. In fact, each ICL term must be accompanied by an ICL formula stating the guarding condition for the term to be considered.

Consider the OCL formula $e.p.p' < 42$. Let us assume that e evaluates to o, $E_{\mathcal{P}}(o,p) = \psi$, and the domain of ψ is the set of objects o_1 and o_2 with $E_{\mathcal{P}}(o_1,p') = 41$, $E_{\mathcal{P}}(o_2,p') = 43$. The OCL expression $e.p.p'$ is thus evaluated to this collection:

$$\{\langle \psi = o_1, 41\rangle, \langle \psi = o_2, 43\rangle\}$$

The formulae constrains use of the term in a formulae context. Let us complete the partial evaluation of the OCL formulae $e.p.p' < 42$. As before, we map OCL's $<$ to ICL's $<$, but we lift the construct to collections; the multiple terms for the left operand are combinatorially combined with the single term (42) for the right operand in a conjunction with the guarding formulae in implications. Thus:

$$((\psi = o_1) \Rightarrow 41 < 42) \wedge ((\psi = o_2) \Rightarrow 43 < 42)$$

The shortcut interpretations for ICL symbols imply this simplification: $\psi \neq o_2$. Formally, the partial evaluator's deduction rule for property access changes as follows:

$$\frac{E \vdash e \searrow ts}{E \vdash \boxed{e.p} \searrow dot(E, ts, p)} \quad \text{(PE-PROPACC)}$$

The operation dot on terms is defined as follows:

$$dot(E, ts, p) = \bigcup_{\langle \Phi, \tau \rangle \in ts} ts_{\Phi,\tau}$$

where

$$ts_{\Phi,\tau} = \begin{cases} \{\langle \Phi, E_{\mathcal{P}}(o,p)\rangle\}, \\ \quad \text{if } \tau \text{ is of the form } o \\ \{\langle \Phi \wedge (\psi = o), E_{\mathcal{P}}(o,p)\rangle \mid o \in [\psi]\}, \\ \quad \text{if } \tau \text{ is of the form } \psi \end{cases}$$

All rules defining or using expression evaluation must be systematically lifted to collections. For instance:

$$E \vdash \boxed{\textbf{self}} \searrow \{true, E_{\mathcal{V}}(\textbf{self})\} \text{ (PE-SELF)}$$

$$\frac{E \vdash e_l \searrow ts_l, e_r \searrow ts_r}{E \vdash \boxed{e_l < e_r} \searrow lt(ts_l, ts_r)} \quad \text{(PE-LT)}$$

The operation lt on terms is defined as follows:

$$lt(ts_l, ts_r) = \bigwedge_{\langle \Phi_l, \tau_l \rangle \in ts_l} \bigwedge_{\langle \Phi_r, \tau_r \rangle \in ts_r} (\Phi_l \wedge \Phi_r) \Rightarrow (\tau_l < \tau_r)$$

The treatment of path expressions, as described above, only covers single-valued expressions. Many-valued expressions necessitate some additional provisions in interpretation. In particular, OCL's flattening semantics must be incorporated, as shown, e.g., in [15].

VIII. Implementation

Both the big-step semantics of Figure 2 and the translation rules of Figure 5 provide blueprints for straightforward implementations of OCL interpreters and translators, resp. In fact, with the modifications presented in Section VI, it is not difficult to implement a partial evaluator for OCL. As a proof of concept, we implemented the partial evaluator of Section VI (translating to ICL) in Haskell (see Footnote 4).

For a more complete coverage of OCL (including **forAll**, **select**, **reject**, **collect**, **includes**, **includesAll**, **isEmpty**, **let**, and access of many-valued properties; **select**, **reject**, and **collect** may also occur in path expressions), and also to be able to evaluate our partial evaluator for OCL on a large scale, we have also implemented it in Java. Our implementation builds on the Eclipse OCL implementation and uses the Eclipse Modelling Framework (EMF) for the internal representation of models and their instances. As described before (and like our Haskell implementation), the partial evaluator translates to ICL; to make use of its translations, we have also implemented a translator from ICL to Choco (http://www.choco-solver.org/), an off-the-shelf constraint solver. This implementation is the basis of our large-scale evaluation presented in Section IX.

IX. Experimental Results

While the small scope hypothesis [39] suggests that for model finding, small instances are sufficient, for literally all other uses of OCL specifications, model instances are typically large. Therefore, we evaluate our partial evaluator in the context of large instances which, due to the notorious absence of "real" models suitable for experimentation in the modelling domain, we find in the form of open-source Java programs.

The model instances we are using are Java programs using the *Java Persistence API* (JPA) [40]. The JPA is comprised of a set of methods to be invoked for storing and retrieving object graphs in a relational database, and a set of Java annotations to be used for directing the mapping between objects and rows of the database. Use of the JPA is subject to complex conditions, the statically checkable part of which we have extracted from the JPA standard to 76 OCL invariants and derivation rules [41].

Two of the identified OCL invariants are shown in Figure 7; note that both invariants do not only refer to JPA annotations (expressed as `jpa`-prefixed properties of type and field declarations), but also to properties of the annotated Java program (such as a class's constructors, their accessibility or a field's declared type). The UML (Ecore) model $\mathcal{M}$ in whose context the invariants and derivation rules are defined is shown in Figure 8. Each of our subject JPA programs is an instance $\mathcal{I}$ of this model, so that we can statically check these programs for the correct use of JPA annotations (which is otherwise only done at run-time) using our regular evaluator.

Totally evaluating our identified 76 OCL expressions on the subject programs of Table I as described in Section IV detects a total of 404 JPA specification violations. The time our implementation requires for this is plotted in Figure 9, which shows the number of expressions evaluated (in thousands, up to 1.3 million) on the abscissa, and which contrasts this time with that required by the original Eclipse OCL evaluator. As can be seen, our implementation (which serves translation and partial evaluation as well) is competitive throughout.

Totally translating them, or symbolic evaluation with the extensions of all classifiers in $E_{\mathcal{I}}$ and all properties in $E_{\mathcal{P}}$ dynamic as described in Section V, is of little interest, since we already know that there are programs satisfying the JPA specification (although in our sample, we have only a single instance, below we will show how we created many more by means of repair). Also, given that our OCL invariants do not cover the well-formedness conditions of Java [42], we cannot expect that the found instances are valid Java programs.

Partial evaluation as in Section VI, on the other hand, with static $E_{\mathcal{I}}$, and all Java properties in $E_{\mathcal{P}}$ static, but JPA properties dynamic (giving us a mixed environment as in Section VI), is useful, for instance to compute all legal JPA annotations of a naked Java program, which serves model completion. The results of the ensuing partial evaluations are also summarized in Table I, which provides the number of syntax nodes in the generated ICL expressions (under "Model Completion"). The column labelled "without" shows the number of nodes that would have been generated without the shortcut optimization (smart constructors) of the ICL operators of Section VI. Note that partial evaluation with all properties in $E_{\mathcal{P}}$ dynamic and simplifying the so-obtained ICL expressions after substituting variables with values where known (as considered as an alternative in Section VI), turned out to be unfit for a comparison — we were unable to produce numbers here, as the translation exceeded the capabilities of our hardware.

While the partial evaluation leading us to the exclusive coverage of JPA annotations exploited the fixation of properties applied to all instances of given classifiers (e.g., all types of fields or all names of classes), partial evaluation shows its full potential only when changeability of properties varies individually. This is the case, for instance, when we attempt to repair the found errors in the use of JPA annotations. However, for repairs of this kind, we must not only consider the conflicting properties as dynamic (so that constraint solving can compute new values for them), but also the properties directly or indirectly constraining them (cf. the example in Section II). We have therefore applied partial evaluation to each of the 404 errors in our subjects, increasing the number of dynamic

```
context ClassDeclaration inv EntityOrEmbeddableHasNoArgConstructor:
(self.jpaIsEntity or self.jpaIsEmbeddable) implies
  self.constructors->isEmpty()
  or self.constructors->exists(c | c.parameters->isEmpty() and
     (c.accessibility = Public or c.accessibility = Protected))

context FieldDeclaration inv PersistentFieldMapping:
self.jpaIsPersistent implies (
  (self.referredType.isCommon and NativeTypes->includes(self.referredType.referredCommon))
  or self.jpaIsOneToOne or self.jpaIsManyToOne
  or (self.referredType.isCommon and self.referredType.referredCommon.jpaIsEmbeddable)
  or ((self.referredType.isCommon or self.referredType.isGeneric or self.referredType.isRaw)
   and self.referredType.referredCommon.implementedInterfaces->
     includes(java_io_Serializable))
```

Fig. 7. Two sample OCL invariants for checking JPA annotations

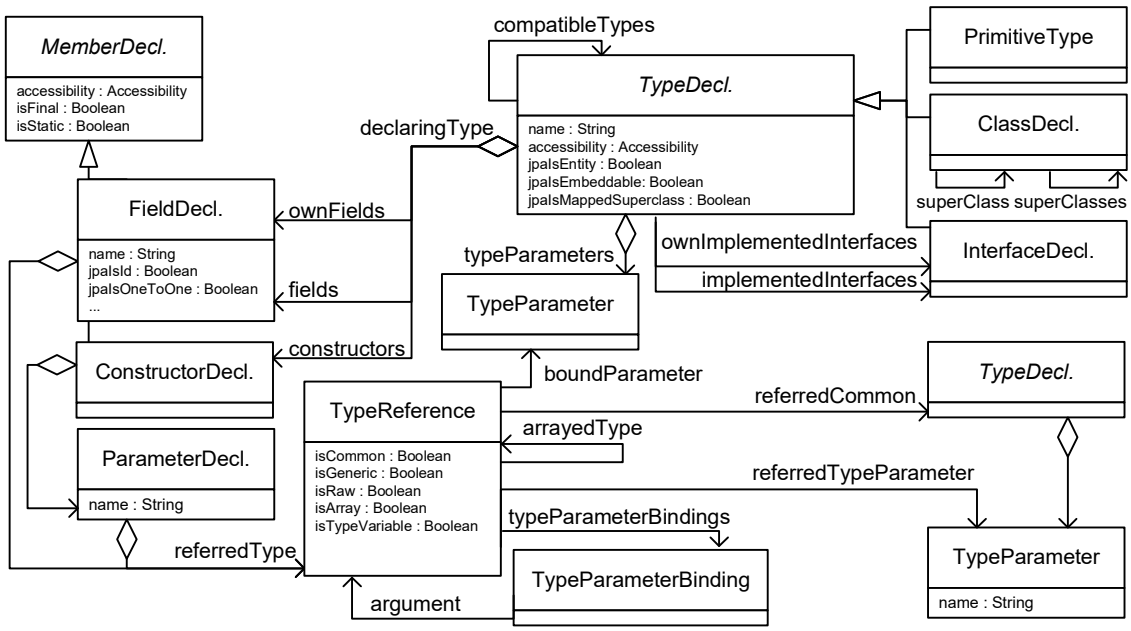

Fig. 8. Model of Java programs using the JPA (excerpt)

properties until the error could be fixed by constraint solving.[6] Table I, right, shows the average complexity of the constraints generated per error depending on d, where d is the depth of the variability, that is, the allowed maximum distance between a dynamic property and the properties of the violated constraint, counted as the number of linking constraints. As can be seen, all resulting CSPs are small initially, yet increase strongly with increasing d; this is indicative of the combinatorial explosion introduced by the unrolling required by variable property access (Section VII). However, all CSPs are small compared to those obtained by full translation, and certainly manageable in size. Figure 10 plots complexity of the so-obtained 581 constraints against the number of dynamic properties resulting from the increasing d; note that the complexity correlates strongly with the number of dynamic properties.

[6]For a deeper discussion of what constraint-based repair can achieve, see [36].

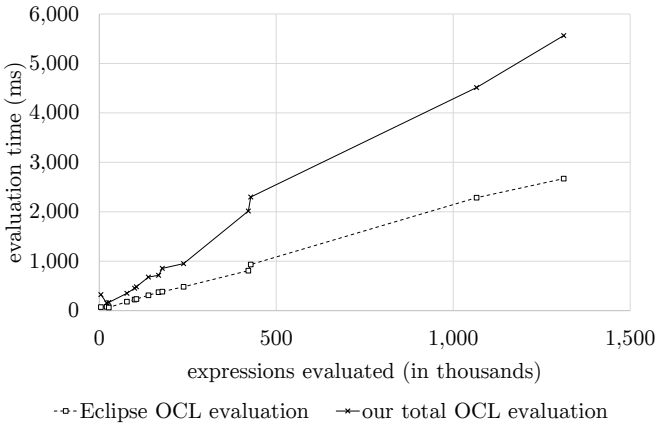

Fig. 9. Evaluation time

X. Lessons Learnt and Future Work

As was to be expected, we found during our experiments that the handling of variable property access (i.e., property

TABLE I
SUBJECT JPA PROGRAMS, THEIR SIZES, AND RESULTS OF TOTAL AND PARTIAL EVALUATION

#	PROJECT	MODEL ELEMENTS (Objects)			TOTAL EVALUATION		PARTIAL EVALUATION Number of Syntax Nodes in ICL Expressions					
		Types	*Fields*	*All*[*]	*Expressions Evaluated*[*]	*Errors*	*Model Completion* shortcut optim.[*]	without[*]	*Model Repair (avg. per error)* $d=0$	$d=1$	$d=2$	$d=3$
1	AgileExpress	103	385	4.25	20.9	1	208	276	6	3,806	—	—
2	Apache CloudStack	4,429	20,048	215	1,066	19	10,220	13,654	56	580	—	—
3	Apache Syncope	591	1,381	18.8	78.2	36	744	991	4	—	—	—
4	Candlepin	681	1,874	27.3	105	8	986	1,327	97	2,234	7,960	17,800
5	IQSS Dataverse	544	3,184	32.2	168	31	1,683	2,234	123	7,936	118,550	—
6	JawaBot	270	449	7.36	26.9	10	236	322	193	113	—	—
7	Kuali Student	1,831	7,993	116	428	48	4,034	5,445	18	1,779	—	—
8	LMCO EurekaStreams	4,121	7,369	58.9	421	91	3,890	5,233	9	246	2,869	74,255
9	Offene-Pflege.de	1,470	4,323	34.6	238	15	2,074	2,861	12	200	—	—
10	OpenMeetings	343	1,883	24.8	99.7	12	955	1,280	9	1,418	—	—
11	OpenOLAT	5,667	24,558	264	1,312	89	13,085	17,256	12	3,455	33,227	50,361
12	OPF Labs Planet Suite	826	3,292	38.9	178	33	1,666	2,251	14	1,366	23,184	62,393
13	QCRI AIDR	693	2,552	32.8	138	11	1,298	1,758	69	517	—	—
14	Tudu	35	78	1.19	4.52	0	43.1	57.4	—	—	—	—

[*] in thousands

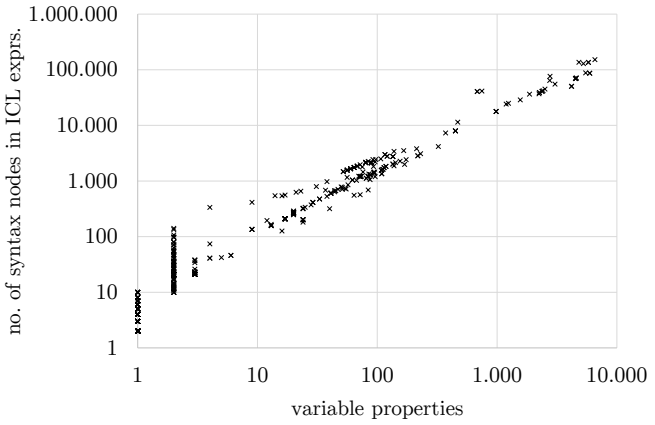

Fig. 10. Complexity of constraints for fix-specific CSPs obtained by using partial evaluation

access via expressions whose value is not known statically; Section VII) is critical for translation and partial evaluation: With no restrictions on the domain of the receiver expression e of a property access $e.p$, all instances of e's type must be considered which, for large models like those of our case study, are in the tens of thousands (see Table I). Complexity tends to explode when variable receivers are chained (as in $e.p.p'$, when the values of neither e nor p are known statically) or expressions contain several variable property accesses (as in $e_l.p_l < e_r.p_r$, when the values of neither e_l nor e_r are known statically). In these cases, translation or partial evaluation may fail due to space restrictions.

Some of these problems can be avoided by eliminating variables via careful invariant design. For instance, if we have a derived property p of classifier C whose value is defined by the derivation rule

context $C::p$ **derive** : **if self**.x **then self**.y **else self**.z **endif**

and an invariant

context C **inv** : **self**.$p.c < 5$

using p, then partial evaluation of the invariant would unroll over the domain of $C::p$ as long as p is dynamic, regardless of whether x, y, or z are. This can be avoided by inlining the derivation rule in the invariant, as in

context C **inv** : **if self**.x **then self**.$y.c$ **else self**.$z.c$ **endif** < 5

which does not require unrolling if y and z are static. However, optimizations of this kind should be left to the partial evaluator.

Generally, if the dynamic variables involved in a receiver expression are additionally constrained by other expressions, these constraints may reduce the domain of the variable receiver, and therefore the complexity induced by unrolling. For instance, for the model $\mathcal{M}$ and instance $\mathcal{I}$ of our running example (Figure 1), partially evaluating the invariant I of (1) in Section II and the invariant

context A **inv** J : **self**.bs->**forAll**$(b|b.c > 1)$

for static extensions of classifiers and static properties c, but dynamic properties bs, would produce the ICL expression

$$(b_1 \in [\![a_1.bs]\!] \vee b_3 \in [\![a_1.bs]\!]) \wedge (b_1 \in [\![a_2.bs]\!] \vee b_3 \in [\![a_2.bs]\!])$$

(repeated here from Section VI) for invariant I, and

$$b_1 \notin [\![a_1.bs]\!] \wedge b_1 \notin [\![a_2.bs]\!] \tag{3}$$

for invariant J. With hindsight, however, partial evaluation of invariant I on $\mathcal{I}$ needlessly unrolls over all instances b_i of classifier B (the potential members of $a_1.bs$ and $a_2.bs$, from which it rules out b_2, since it can see that $b_2.c \not< 5$), even though partial evaluation of invariant J clearly yields that b_1 cannot be a member of $a_1.bs$ or $a_2.bs$.

A possible solution to this problem is to update the domains of variables in the environments $E_{\mathcal{I}}$ and $E_{\mathcal{P}}$ with constraints such as (3), which would then be used in the unrolling of constraints yet to be evaluated. However, as can be seen from the above example, the effect of this critically depends on the order in which the OCL expressions are evaluated, and choosing the best order depends on what is static and what is dynamic. We leave this problem to future work.

XI. Conclusion

OCL is arguably *the* specification language in the modelling domain. However, despite its demonstrated suitability for many different purposes, the technical support for the language itself still lags behind the state-of-the-art. In this paper, we have systematically explored partial evaluation of OCL expressions in the context of a wide array of purposes, ranging from total evaluation (for validation) to total translation (for model finding and verification). Our target formalism is not, as in many other works, that of a specific tool, but rather an intermediate language that can be seen as a reduced version of OCL apt for symbolic evaluation. By mapping this language to a standard constraint solver, we have demonstrated that our partial evaluation of OCL is practical even for large models.

Acknowledgements

This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant STE 906/5-1.

References

[1] Object Management Group, *Object Constraint Language Version 2.4*. Object Management Group, 2014.

[2] ——, *OMG Meta Object Facility (MOF) Core Specification Version 2.5*. Object Management Group, 2015.

[3] ——, *OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.4.1*. Object Management Group, 2011.

[4] M. Gogolla and M. Richters, *Development of UML Descriptions with USE*, ser. EurAsia-ICT 2002. Springer, 2002, pp. 228–238.

[5] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 27–34, 2007.

[6] M. Clavel, M. Egea, and M. A. G. de Dios, "Checking unsatisfiability for OCL constraints," *ECEASST*, vol. 24, 2009. [Online]. Available: http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/334

[7] J. Cabot, R. Clarisó, and D. Riera, "On the verification of UML/OCL class diagrams using constraint programming," *Journal of Systems and Software*, vol. 93, pp. 1–23, 2014.

[8] C. Dania and M. Clavel, "OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints," in *Proc. MODELS 2016*. ACM, 2016, pp. 65–75.

[9] N. Przigoda, R. Wille, and R. Drechsler, "Ground setting properties for an efficient translation of OCL in smt-based model finding," in *Proc. MODELS 2016*. ACM, 2016, pp. 261–271.

[10] A. D. Brucker and B. Wolff, "Semantics, calculi, and analysis for object-oriented specifications," *Acta Inf.*, vol. 46, no. 4, pp. 255–284, 2009.

[11] F. Hilken, P. Niemann, M. Gogolla, and R. Wille, "Towards a catalog of structural and behavioral verification tasks for UML/OCL models," in *Modellierung 2016, 2.-4. März 2016, Karlsruhe*, ser. LNI, A. Oberweis and R. H. Reussner, Eds., vol. 254. GI, 2016, pp. 117–124.

[12] F. Steimann and B. Ulke, "Generic model assist," in *Proc. MODELS 2013*, ser. LNCS, vol. 8107. Springer, 2013, pp. 18–34.

[13] S. Ali, M. Z. Z. Iqbal, A. Arcuri, and L. C. Briand, "Generating test data from OCL constraints with search techniques," *IEEE Trans. Software Eng.*, vol. 39, no. 10, pp. 1376–1402, 2013.

[14] B. K. Aichernig and P. A. P. Salas, "Test case generation by OCL mutation and constraint solving," in *Fifth International Conference on Quality Software (QSIC 2005), 19-20 September 2005, Melbourne, Australia*. IEEE Computer Society, 2005, pp. 64–71. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=10545

[15] F. Steimann, "From well-formedness to meaning preservation: model refactoring for almost free," *Software and System Modeling*, vol. 14, no. 1, pp. 307–320, 2015.

[16] N. D. Jones, "Transformation by interpreter specialisation," *Sci. Comput. Program.*, vol. 52, pp. 307–339, 2004.

[17] W. R. Cook and R. Lämmel, "Tutorial on Online Partial Evaluation," in *Proc. DSL 2011*, ser. EPTCS, vol. 66, 2011, pp. 168–180.

[18] C. A. González and J. Cabot, "Formal verification of static software models in MDE: A systematic review," *Information & Software Technology*, vol. 56, no. 8, pp. 821–838, 2014.

[19] ——, "Test Data Generation for Model Transformations Combining Partition and Constraint Analysis," in *Proc. ICMT 2014*, ser. LNCS, vol. 8568. Springer, 2014, pp. 25–41.

[20] J. Cabot and M. Gogolla, "Object constraint language (OCL): A definitive guide," in *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, ser. LNCS, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds., vol. 7320. Springer, 2012, pp. 58–90.

[21] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using boolean satisfiability," in *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*, G. D. Micheli, B. M. Al-Hashimi, W. Müller, and E. Macii, Eds. IEEE Computer Society, 2010, pp. 1341–1344. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5450668

[22] A. Queralt, A. Artale, D. Calvanese, and E. Teniente, "Ocl-lite: Finite reasoning on UML/OCL conceptual schemas," *Data Knowl. Eng.*, vol. 73, pp. 1–22, 2012.

[23] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from UML to alloy," *Software and System Modeling*, vol. 9, no. 1, pp. 69–86, 2010.

[24] B. Beckert, U. Keller, and P. H. Schmitt, "Translating the Object Constraint Language into first-order predicate logic," in *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002, available at i12www.ira.uka.de/~key/doc/2002/BeckertKellerSchmitt02.ps.gz.

[25] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler, "Formalizing UML models and OCL constraints in PVS," *Electr. Notes Theor. Comput. Sci.*, vol. 115, pp. 39–47, 2005.

[26] M. Kuhlmann and M. Gogolla, "From UML and OCL to relational logic and back," in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, ser. LNCS, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., vol. 7590. Springer, 2012, pp. 415–431.

[27] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[28] N. D. Jones, "An introduction to partial evaluation," *ACM Computing Surveys (CSUR)*, vol. 28, no. 3, pp. 480–503, 1996.

[29] J. Hatcliff, "An Introduction to Online and Offline Partial Evaluation using a Simple Flowchart Language," in *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, ser. LNCS, vol. 1706. Springer, 1999, pp. 20–82.

[30] H. Song, X. Zhang, N. Ferry, F. Chauvel, A. Solberg, and G. Huang, "Modelling Adaptation Policies as Domain-Specific Constraints," in *Proc. MODELS 2014*, ser. LNCS, vol. 8767. Springer, 2014, pp. 269–285.

[31] M. Giese and D. Larsson, "Simplifying Transformations of OCL Constraints," in *Proc. MoDELS 2005*, ser. LNCS, vol. 3713. Springer, 2005, pp. 309–323.

[32] B. Beckert, R. Hähnle, and P. H. Schmitt, Eds., *Verification of Object-Oriented Software. The KeY Approach - Foreword by K. Rustan M. Leino*, ser. LNCS. Springer, 2007, vol. 4334.

[33] F. Steimann, C. Kollee, and J. von Pilgrim, "A Refactoring Constraint Language and Its Application to Eiffel," in *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, ser. LNCS, M. Mezini, Ed., vol. 6813. Springer, 2011, pp. 255–280.

[34] F. Tip, A. Kiezun, and D. Bäumer, "Refactoring for generalization using type constraints," in *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, R. Crocker and G. L. S. Jr., Eds. ACM, 2003, pp. 13–26.

[35] B. Ulke and F. Steimann, "OCL as a constraint generation language," in *Proceedings of the MODELS 2013 OCL Workshop*, ser. CEUR Workshop Proceedings, J. Cabot, M. Gogolla, I. Ráth, and E. D. Willink, Eds., vol. 1092. CEUR-WS.org, 2013, pp. 93–102. [Online]. Available: http://ceur-ws.org/Vol-1092/ulke.pdf

[36] F. Steimann, J. Hagemann, and B. Ulke, "Computing repair alternatives for malformed programs using constraint attribute grammars," in *Proc. OOPSLA 2016, part of SPLASH 2016*. ACM, 2016, pp. 711–730.

[37] M. V. Cengarle and A. Knapp, "A formal semantics for OCL 1.4," in *UML 2001*, M. Gogolla and C. Kobryn, Eds. Springer, 2001, pp. 118–133.

[38] E. P. K. Tsang, *Foundations of Constraint Satisfaction*, ser. Computation in cognitive science. Academic Press, 1993.

[39] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Books24x7.com, 2012.

[40] L. DeMichiel, *JSR 317: Java Persistence API 2.0*. Sun Microsytems, 2009.

[41] B. Ulke, "Reparatur von Programmen mithilfe der Object Constraint Language," Ph.D. dissertation, Lehrgebiet Programmiersysteme, Fernuniversität in Hagen, 2017.

[42] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 7 Edition*, 1st ed. Addison-Wesley Professional, 2013.