Smart Bound Selection for the Verification of UML/OCL Class Diagrams

Robert Clarisó, Carlos A. González and Jordi Cabot

Abstract—Correctness of UML class diagrams annotated with OCL constraints can be checked using bounded verification techniques, *e.g.*, SAT or constraint programming (CP) solvers. Bounded verification detects faults efficiently but, on the other hand, the absence of faults does not guarantee a correct behavior outside the bounded domain. Hence, choosing suitable bounds is a non-trivial process as there is a trade-off between the verification time (faster for smaller domains) and the confidence in the result (better for larger domains). Unfortunately, bounded verification tools provide little support in the bound selection process.
In this paper, we present a technique that can be used to (i) automatically infer verification bounds whenever possible, (ii) tighten a set of bounds proposed by the user and (iii) guide the user in the bound selection process. This approach may increase the usability of UML/OCL bounded verification tools and improve the efficiency of the verification process.

Index Terms—Formal Verification, UML, Class Diagram, OCL, Constraint Propagation, SAT

1 Introduction

Ensuring software correctness is a challenging problem in software engineering. Techniques such as *testing* or *formal verification* can be used to identify and diagnose software defects. These analyses can be performed at a low level of abstraction, *i.e.*, to detect faults in an implementation, but they can also be used earlier in the development process to inspect *software models*, facilitating error detection and correction.

There are several available notations for modeling software systems, such as UML [1] or Alloy [2]. Among them, UML class diagrams are arguably the most commonly used models for describing the specification of a software system [3]. In order to increase their precision and expressiveness, class diagrams can be annotated with constraints written in the Object Constraint Language (OCL). Checking the correctness of a UML/OCL model is a complex problem and, in general, undecidable [4]. This has forced existing tools for UML/OCL analysis [5] to accept a series of trade-offs: reducing the expressiveness of the modeling language [6]; performing an incomplete search [7]; requiring user guidance to conduct the verification [8]; or, finally, limiting the search space [9].

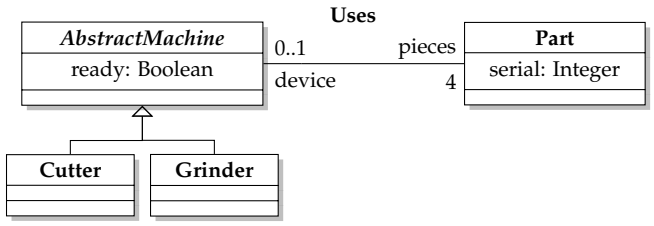
The latter strategy, called *bounded verification*, allows an efficient and automatic analysis of expressive models. Hence, it is popular among existing tools [9], [10], [11], [12], [13], [14]. Nevertheless, bounded verification only proves the presence or absence of faults *within* the bounded search space, with no guarantees about what happens beyond these bounds. Therefore, bounded verification is useful in two scenarios: (a) when a fault is found within the search space; or (b) when no fault is found but the search space is large enough to provide sufficient confidence about the model correctness to practitioners. In both scenarios, the choice of suitable verification bounds is critical. Unfortunately, setting search space boundaries is a limiting factor, since current tools provide little support, either setting inadequate default values or forcing users to manually define these bounds for each model element, which is impractical for large models.

Using an unnecessarily large search space makes bounded verification less efficient. For instance, if a model is analyzed with a SAT solver, large domains require more boolean variables for the encoding and produce larger formulas. Still, selecting optimal boundaries for a search problem, *i.e.*, as narrow as possible, is also a computationally complex problem [15]. Thus, currently users may only rely on heuristics such as the *small scope hypothesis* [16], [17] (use small domains assuming that they will suffice to detect most faults) or *incremental scoping* (invoke the solver repeatedly using progressively larger domains until a fault is detected).

In this paper, we present a technique that can assist users of any UML/OCL bounded verification tool to effectively set the boundaries of the search space, regardless of the specific solver employed by such tool. This approach starts from a set of initial bounds, which may be infinite. Then, the constraints in the model are abstracted as *size constraints* [18]: rather than defining the set of valid instances of the model, they restrict the size (min and max values) and population (number of objects) of those instances. In this system of size constraints, we use a technique called *interval constraint propagation* to discard unproductive values from domain bounds. This process is efficient because (a) abstraction makes the system of constraints more amenable to analysis than the original verification problem and (b) pruning values does not require solving the system of constraints. Lastly, we provide the tightened bounds to the UML/OCL verification tool, which has a better performance on the

- *R. Clarisó is with the IT, Multimedia and Telecommunication Department, Universitat Oberta de Catalunya, Barcelona, Spain. E-mail: rclariso@uoc.edu*
- *Carlos A. González is with the SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg. E-mail: carlos.gonzalez@uni.lu*
- *J. Cabot is with ICREA, Pg. Lluís Companys 23, 08010 Barcelona, Spain. E-mail: jordi.cabot@icrea.cat*

Manuscript received Month DD, YYYY; revised Month DD, YYYY.




```
context Part inv UniqueSerials :
    Part.allInstances() → isUnique(p | p.serial)

context AbstractMachine inv MachineAvailability :
    Cutter.allInstances() → exists(c | c.ready) and
    Grinder.allInstances() → exists(g | g.ready)
```

Fig. 1. UML/OCL class diagram used as example

reduced bounds. We report our experiments using the USE model validator plug-in [13] as the underlying bounded verification tool, and describe the performance gains.

Example 1. Let us consider the class diagram from Figure 1 describing the relationship between *machines* and *parts*. Graphical constraints such as association end multiplicities and inheritance hierarchies define constraints on the valid populations for classes and associations, *e.g.*, there are four parts for each machine.

OCL invariants define additional restrictions on these populations and the domains of attributes. For instance, the invariants in the example require serial numbers to be unique (`UniqueSerial`) and at least one machine of each type to be ready (`MachineAvailability`). From the point of view of class populations, invariant `UniqueSerial` implicitly sets an upper bound on the population of class `Part`: the number of values in the domain of attribute `serial`. As another example, `MachineAvailability` implicitly sets a lower bound of 1 for the population of class `Cutter` and `Grinder`.

These constraints can be used to automatically infer bounds without any user intervention, *e.g.*, for the `MachineAvailability` invariant this results in a lower bound of 1 for classes `Cutter` and `Grinder`, of 8 for class `Part` and 8 for association `Uses`. However, this inference is most effective when used to refine partial bound information provided by a designer. For instance, just by assuming a limit of 10 serial numbers, we can infer that there is exactly 1 `Cutter` and 1 `Grinder`, between 8 and 10 parts and at most 8 links among machines and parts.

Even if the constraints in Example 1 seem trivial in hindsight, a UML/OCL model may contain many constraints like these that will typically interact, making it impossible for users to consider all of them when choosing a proper set of verification bounds. Moreover, trivial approaches such as using a default bound for all model elements make verification inefficient. Hence, providing automatic support to guide bound selection can be helpful.

There are three usage scenarios for bound selection/tightening, depending on the information provided by the user. With no user inputs (*i.e.*, all bounds are assumed to be infinite), the method operates as an automatic "one-off" pre-processing step before the verification takes place, attempting to infer suitable verification bounds from the constraints in the model. If the user provides some candidate verification bounds, the method automatically improves those bounds, thus reducing the search space and making verification more efficient. It may also be able to detect that the initial bounds are too tight without needing to start the verification, *i.e.*, by returning empty bounds as the result of bound tightening. Finally, the method could operate interactively, with users providing partial bounds and using the method to tighten them as much information as possible and suggest what to bound next.

This manuscript is an extended version of a short paper published in the 13th International Conference on Software Engineering and Formal Methods (SEFM'2015) [19]. The contributions of this extended version are: (1) a comprehensive description of bound propagation, its computational complexity and limitations; (2) a more complete description of the bound tightening process; (3) additional experimental results; (4) a more extensive description of the state-of-the-art and (5) a discussion of the application scenarios of this method.

The rest of the paper is organized as follows: Section 2 gives an overview of the method. Section 3 gets into detail on how constraint propagation works and what information is inferred from the different modeling constructs. Section 4 presents the experimental results and the performance gains obtained. Section 5 discusses several verification problems where this approach can be applied. Section 6 covers the related work. Finally, conclusions and future work are presented in Section 7.

2 Overview

The designer of UML/OCL class diagrams has several expectations about its applicability and quality [9], [20]. For instance, it should be possible to create an instance that satisfies all the constraints in the model simultaneously. Also, it is desirable to be aware of redundancies among constraints [21], [22], [23]. These correctness properties can be formally checked using bounded verification.

Bounded verification tools explore a finite search space looking for an instance of the model that acts as a witness of the correctness property under analysis (see Fig. 2, non-dashed elements). For existential properties, the witness is an *example* (proving the property) while for universal properties this witness is a *counterexample* (disproving it). If no witness is found, we cannot conclude anything about the property: a witness may lie outside this search space.

Current approaches work by directly translating the model into a finite domain constraint formalism where efficient off-the-shelf solvers are available, *e.g.*, SAT or constraint satisfaction problems (CSP). However, they either force default bounds to the search space or let the user in charge of manually deciding which bounds to use. This is by no means an easy task since it requires to specify upper and lower bounds for the population of each class and association in the model, as well as finite domains for each attribute. When facing large models, this translates into setting boundary values for a set of modeling elements ranging above the hundreds, a tedious and potentially error prone activity.

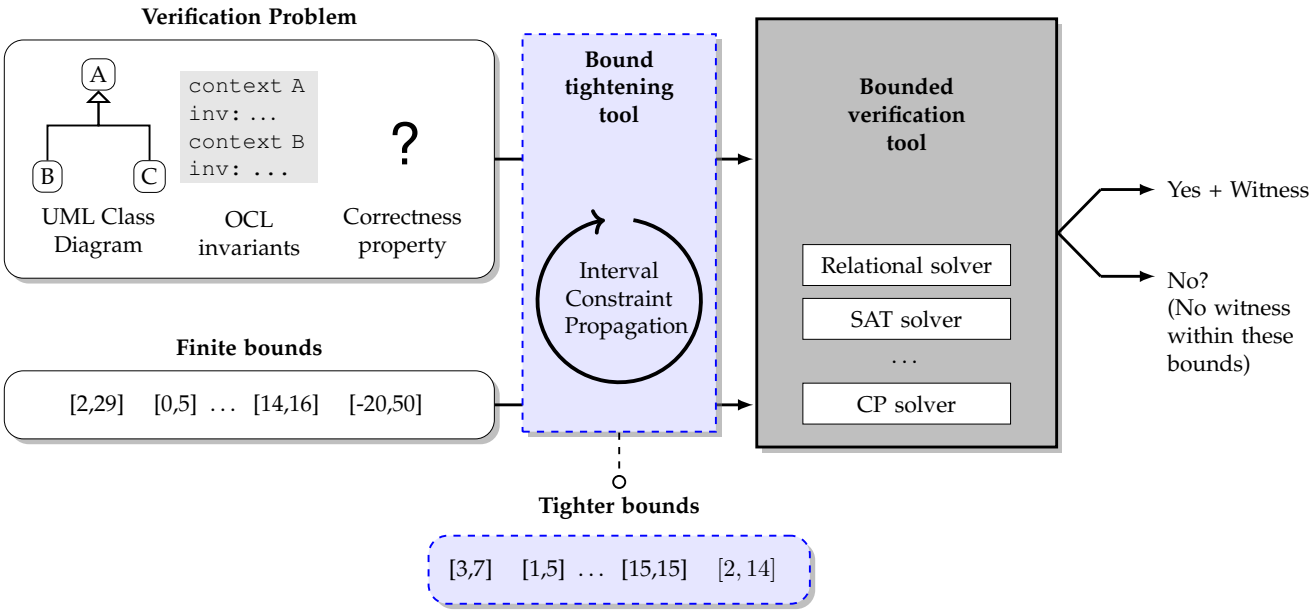


Fig. 2. Typical flow with a bounded verification tool and the role of bound tightening

In this paper, we propose an alternative approach that refines a set of bounds (either from scratch or by tightening an initial set of bounds proposed by the designer) and then relays these improved bounds to the solver (see the dashed elements in Fig. 2). The computation of the bounds relies on interval constraint propagation techniques. As we explain in the following Section, our approach collects all implicit and explicit constraints from the UML/OCL model and formalizes them as a CSP over a set of variables representing the search space boundaries. As we will see in the experimental results, tightening bounds does not add a lot of overhead to the whole verification process and has the potential to drastically reduce the time needed to verify the model. This process is not optimal but it is *safe*, *i.e.*, it may fail to compute the tightest bounds, but it will preserve any witnesses within the original bounds.

3 Bound Tightening Procedure

This section describes the derivation of a CSP from the initial UML/OCL model. The analysis of this CSP returns an appropriate set of bounds to be used in the subsequent verification of that model. These improved bounds can be used by any UML/OCL bounded verification tool independently of the approach used by its underlying solver (SAT, constraint programming, ...), as it is shown in Figure 2.

We first review some basic concepts about CSPs and the constraint propagation capabilities of constraint solvers (Sec. 3.1). Then, we describe the structure of the CSP (Sec. 3.2) and how the UML and OCL constraints in the input model are formalized (Sec. 3.3). Finally, we discuss how the generated constraints are used to tighten bounds (Sec. 3.4).

3.1 CSPs and Propagation

A Constraint Satisfaction Problem (CSP) is characterized by three elements:

- A finite set of *variables* V.
- The set of *domains* D of potential values for each variable.
- The set of *constraints* C over the variables in V.

Solving a CSP consists in choosing, for each variable, one value from its domain such that all constraints are satisfied.

Example 2. Let us consider a CSP with three variables X, Y and Z, each taking values in the integer interval $[-2, 10]$, and the constraints: $X = Y + Z$, $Y = \max(X, Z)$ and $X + 2Z \leq 2$. Two potential solutions to this CSP are $X = 0, Y = 0, Z = 0$ and $X = -2, Y = -1, Z = -1$.

A typical approach for computing a satisfying assignment consists in searching it by assigning values to variables one at a time in a certain order and backtracking when a partial assignment cannot be extended any further. This search process is aided by *early evaluation* (*i.e.*, detecting when a partial assignment is unfeasible and can be discarded) and *propagation* (*i.e.*, removing values from the domain of unassigned variables using information about the constraints and the values of previously assigned variables).

While solving a CSP is computationally expensive, propagation is much faster: practical implementations attempt to tighten domains in a pragmatic cost-effective way, instead of computing the optimal bounds with potentially slow computations. As the goal of this paper is tightening domain bounds rather than finding a specific instance within those bounds, propagation suits our needs better than CSP-solving. Thus, the CSP we define from our UML/OCL model will only be used to apply propagation. To this end, we will use the *hybrid integer-real Interval arithmetic Constraint solver* (IC) from the ECL^{i}PS^e Constraint Programming System [24]. The IC solver can handle both integral and real variables and it provides powerful interval constraint propagation capabilities.

Example 3. Given the CSP from Example 2, the propagation implemented by the IC solver can tighten the original

bounds to the following: $X \in [-2,6], Y \in [-2,6], Z \in [-2,2]$. Notice that propagation reduces the search space but in general it is unable to discard all unfeasible value assignments. For instance, in this example it cannot detect that there is no solution with $Z = 2$.

During the search process, it is desirable to detect the unfeasibility of partial assignments as early as possible. This helps to avoid backtracking so, unless the computational overhead is too high, it will reduce the total execution time of the search. To this end, one potential strategy is to consider a subset of the entire problem and check the feasibility of the partial assignment in this subset. This concept is called *local consistency*, with different levels of precision depending on how the subset of interest is selected, *e.g.*, *node consistency*, *arc consistency* or *path consistency*, among others.

Local consistency is defined in an abstract way in terms of a *constraint network*: an undirected graph with one node per variable and one node per constraint, with edges between a constraint and the variables that participate in it. In this network, the following consistency notions can be defined:

- **Node consistency**: Let $C(X)$ be a unary constraint C over a variable X with domain D. Then, node X is node consistent if, for each value $d \in D$, $C(d)$ is satisfied. The constraint network is node consistent if all nodes are node consistent.
- **Arc consistency**: Let $C(X,Y)$ be a binary constraint C over variables X and Y with domains D_x and D_y respectively. The arc defined by variable X and constraint C is arc consistent if, for each value $x \in D_x$, there is a value in $y \in D_y$ such that $C(x,y)$ is satisfied. The constraint network is arc consistent if all arcs are arc consistent.
- **Path consistency**: Let X, Y and Z be three variables with domains D_x, D_y and D_z and let C_{xy}, C_{xz} and C_{yz} be the binary constraints among them. The pair $\langle X,Y\rangle$ is path consistent with respect to Z if, for any pair of values $d_x \in D_x$ and $d_y \in D_y$ such that $C_{xy}(d_x,d_y)$ is satisfied, then there is a value $d_z \in D_z$ such that both $C_{xz}(d_x,d_z)$ and $C_{yz}(d_y,d_z)$ are satisfied. The constraint network is path consistent if all triples of variables are path consistent.

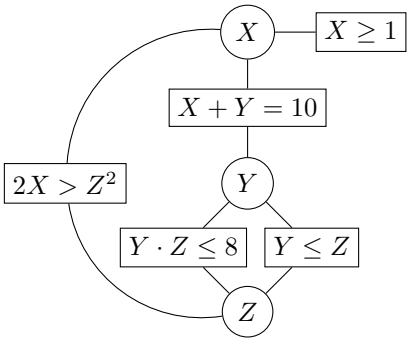
Notice that achieving local consistency in a CSP with non-empty domains is *necessary* for having a satisfying assignment, but in the general case it is not *sufficient*: a constraint problem may be locally consistent but still unfeasible.

Example 4. Figure 3(a) shows an example of a constraint satisfaction problem with 3 integer variables (X, Y and Z) and 5 constraints (1 unary and 4 binary). This CSP can be represented as the constraint network in Figure 3(b) in order to study its local consistency. Considering the domains proposed in Figure 3(c), we can see that:

- The CSP is not node consistent with the domains α: values $[-10,0]$ in the domain of X do not satisfy the unary constraint $(X \geq 1)$.
- Using the domains β, the CSP is now node consistent, but not arc consistent. For instance, consider the arc defined by variable Z and the binary constraint $(2X > Z^2)$. Given that X is smaller or equal to 10, values for Z outside [-4,4] do not satisfy the constraint.
- The domains γ make the CSP arc consistent, but not path consistent: considering $Z = 4$ and $Y = 4$, there is no value for X that can simultaneously satisfy $(X + Y = 10)$ and $(2X > Z^2)$. The former requires $X = 6$ while the latter requires $X > 8$.
- Finally, the domains δ make the CSP path consistent. In fact, they encode a potential solution to the CSP.

$$\begin{aligned} X + Y &= 10 \\ X &\geq 1 \\ 2X &> Z^2 \\ Y \cdot Z &\leq 8 \\ Y &\leq Z \end{aligned}$$

(a) Constraint system



(b) Constraint network

	Domains			Consistent?		
	X	Y	Z	**Node**	**Arc**	**Path**
α	[-10,10]	[-10,10]	[-10,10]	No	No	No
β	[1,10]	[-10,10]	[-10,10]	Yes	No	No
γ	[6,10]	[0,4]	[0,4]	Yes	Yes	No
δ	[9,9]	[1,1]	[1,1]	Yes	Yes	Yes

(c) Consistency analysis of sample domains α, β, γ and δ

	Domain	Constraint	Propagation
1	X:$[-10,10]$	$X \geq 1$	X:$[-10,10]\rightarrow[1,10]$
2	X:$[1,10]$	$2X > Z^2$	Z:$[-10,10]\rightarrow[-4,4]$
3	Z:$[-4,4]$	$Y \leq Z$	Y:$[-10,10]\rightarrow[-10,4]$
4	Y:$[-4,4]$	$X + Y = 10$	X: $[1,10]\rightarrow[\mathbf{6},\mathbf{10}]$
5	X:$[6,10]$	$X + Y = 10$	Y: $[-10,4]\rightarrow[\mathbf{0},\mathbf{4}]$
6	Y:$[0,4]$	$Y \leq Z$	Z: $[-4,4]\rightarrow[\mathbf{0},\mathbf{4}]$

(d) Propagation steps from domains α to arc-consistent domains γ

Fig. 3. Example of local consistency in a CSP.

As stated before, constraint propagation is the process of checking and enforcing local consistency in a CSP. The computational effort required by propagation depends on the level of consistency being enforced. For instance, Figure 3(d) describes how propagation can convert the initial domains α into the arc-consistent domains γ. In this case, 6 propagation steps are required.

The IC library uses two different notions of local consistency depending on the relational operator used in the constraints. This design decision aims to achieve a reasonable trade-off between the richness of the supported constraints and the efficiency of constraint propagation. For inequality constraints, *arc consistency* is enforced. Arc consistency requires that, for each binary constraint, each value in the domain of the first variable has a corresponding value in the domain of the second one that satisfies the constraint. However, for equality constraints a weaker notion called *bound consistency*: only the extreme values of the domain (the

minimum and the maximum of the interval) are checked for consistency. This means that bound consistency is unable to propagate *holes*, *e.g.*, for $X = Y$ the domains $X : [0, 4]$ and $Y : \{0, 2, 4\}$ would be bound consistent, ignoring the fact that X cannot take the values 1 and 3.

The algorithms employed to enforce arc consistency in ECL^{i}PS^e are variants of two well-known algorithms, AC-3 and AC-5 [25]. AC-3 has a worst-case complexity of $O(ed^3)$, where e is the number of binary constraints and d is the size of the largest domain. For special cases it can operate faster, *i.e.*, if the constraint graph is a tree its worst-case complexity drops to $O(ed)$. Meanwhile, AC-5 is a generic algorithm, that can achieve a worst-case bound of $O(ed)$ for restricted constraint types.

3.2 Structure of the CSP

In order to apply CSP techniques, the first step is characterizing the problem in terms of a set of variables, domains and constraints. In our case, the problem is establishing bounds for the verification of UML/OCL models. Hence, instead of a CSP whose solutions are witnesses to the verification problem, we will define a CSP that constrains the *size* of these witnesses. A suitable CSP formalization is described in Table 1:

- The *variables* of this CSP will not characterize a complete instance, but rather the search space boundaries: how many objects and links and which attribute values should be considered when instantiating the model. Additional auxiliary variables are used for convenience to encode complex constraints associated with rich OCL expressions.
- The *domains* of this CSP will be the output of our approach, as we are addressing a bound tightening problem. The analysis may start without providing any information about the domains, *e.g.*, from 0 to ∞ objects per class. In this way, we can attempt to automatically infer finite bounds for each of the variables in our problem. As this is usually not possible, the designer may also define the set of bounds that he intended to use and let the constraint solver propagate the restrictions in order to tighten these bounds.
- The *constraints* of the CSP include graphical restrictions from the UML class diagram and the textual OCL invariants. In the case of OCL, the constraints in the CSP are not a direct translation of the invariants (*e.g.*, as done in [9]), but rather an abstraction of the invariants that only considers size information. Moreover, the correctness property under analysis is another constraint of the CSP.

Example 5. Let us revisit the UML/OCL model from Example 1. The model has four classes (`AbstractMachine`, `Cutter`, `Grinder` and `Part`), one association (`Uses`) and one attribute (`serial`), so the CSP will have six variables. The domain of each class (association) variable will be [0,N], where N is the maximum number of objects (links) allowed for the class (association) and its subclasses. As abstract classes cannot be instantiated, N is simply an upper bound for the number of objects in all of its subclasses. For the attribute variable, the domain is $[-M, M]$ where M is the minimum/maximum value for this attribute.

Using these variables, the graphical elements in the model define the following constraints:

$$\texttt{AbstractMachine} = \texttt{Cutter} + \texttt{Grinder} \quad (1)$$

$$\texttt{Uses} \leq \texttt{Part} * \texttt{AbstractMachine} \quad (2)$$

$$\texttt{Uses} = 4 * \texttt{AbstractMachine} \quad (3)$$

$$\texttt{Uses} \leq \texttt{Part} \quad (4)$$

where (1) is the inheritance hierarchy, (2) is the definition of association "Uses", (3) is the multiplicity 4 of association end "pieces" and (4) is the multiplicity 0..1 of association end "device".

3.3 OCL Constraint analysis

There are several proposals in the literature for formalizing UML class diagrams as a CSP [4], [9], [26]. Regarding OCL, CSPs have been used for verification [9], but that formalization is unsuitable for bound tightening. In this section, we detail how OCL invariants can be encoded as CSP constraints in order to perform bound tightening. Notice that this analysis can also be applied to any graphical constraint in the UML model since, as stated in [27], they can also be expressed as a combination of OCL expressions.

Our method builds upon the work of Yu et al. [18], which addresses the verification of size properties of collection types in OCL. Abstracting away the contents of collections but preserving the constraints on their sizes, it computes an abstract system of constraints that is sufficient to detect size-related errors such as buffer overflows. Rather than solving this abstract system of constraints, our proposal uses it to tighten bounds. As a result, we are able to accelerate the verification of complex properties that may require knowledge beyond the size of collections. Furthermore, the abstraction process is extended to cover more operations involving other data types (*e.g.*, `isUnique`), thus supporting the majority of operations in OCL.

Table 2 summarizes how OCL invariants are abstracted into a size constraint. This table only considers the subset of the OCL language required to analyze the invariants from Example 1. Due to space constraints, the complete analysis of the remaining operations available in the OCL specification can be found in Appendix A.

The first column of this Table represents the OCL expression e being abstracted. The second column identifies the different combinations of data types that can occur in the context of e, where $t(e)$ denotes the type of the value resulting from the evaluation of e (r, i, n, b, s, st, os, bg and sq are shorthand notations for the OCL real, integer, unlimited natural, boolean, string, set, ordered set, bag and sequence types, respectively). Finally, the third column shows the size constraint $e.c$ derived from the analysis of e. This size constraint is expressed with the help of an auxiliary variable $e.v$, which is of integer type when dealing with operations over collections or strings, and shares the operation data type otherwise. When $e.c$ holds, $e.v$ represents the size of the collection or the length of the string, for the case of

TABLE 1
Definition of the CSP used to tighten verification bounds

Vars (V)	**Domains (D)**	**Constraints (C)**
A variable cl for each class	Potential number of objects in class cl, either $[0, \infty)$ or a user-provided domain	– UML: generalizations, association end multiplicities, class multiplicities – OCL: all invariants – Correctness property under analysis, *e.g.*, no redundant invariants
A variable as for each association	Potential number of links in association as, either $[0, \infty)$ or a user-provided domain	– UML: association end multiplicities – OCL: invariants containing navigations through association as
A variable at for each attribute	Potential values of attribute at, depending on its data type, *e.g.*, $[0, 1]$ for boolean, $(-\infty, \infty)$ for integers or a user-provided domain	– OCL: invariants accessing the value of attribute at
An auxiliary variable aux_e for each subexpression e in each OCL constraint	Potential values of the expression e depending on its data type. Non-basic types are abstracted, *e.g.*, collections are abstracted as integers encoding their size.	– A constraint establishing the value of e in terms of the values of its subexpressions. – Correctness property under analysis, *e.g.*, the root expression of each invariant must evaluate to 1 (all invariants must be true)

TABLE 2
Analysis of OCL operations from Example 1

	OCL Expression e	**Type** $t(e) : [t(e_1)][t(c_1)]$	**Size Constraint e.c**
1	$e_1.attr$	{r,i,n,b,s}: {r,i,n,b,s}	$domain(e.v) \subseteq domain(attr)$
2	$c_1 \rightarrow exists(e_1)$	b: {st,os,sq,bg},b	$(0 \leq e.v \leq 1) \wedge$ $((c_1.v = 0 \vee e_1.v = 0) \rightarrow (e.v = 0)) \wedge$ $((e_1.v = 1) \rightarrow (e.v = (c_1.v \geq 1))) \wedge$ $c_1.c \wedge e_1.c$
3	$c_1 \rightarrow isUnique(e_1)$	b: {st,os,sq,bg}, {r,i,n}	$(0 \leq e.v \leq 1) \wedge$ $((e.v = 0) \rightarrow (c_1.v \geq 2)) \wedge$ $((e.v = 1) \rightarrow (domain\_size(e_1.v) \geq c_1.v)) \wedge$ $c_1.c \wedge e_1.c$
		b: {st,os,sq,bg}, b	$(0 \leq e.v \leq 1) \wedge$ $((e.v = 1) \rightarrow (c_1.v \leq 2)) \wedge$ $((e.v = 0) \rightarrow (c_1.v \geq 2)) \wedge$ $((e.v = 1) \rightarrow (domain\_size(e_1.v) \geq c_1.v)) \wedge$ $c_1.c \wedge e_1.c$
		b: {st,os,sq,bg}, s	$(0 \leq e.v \leq 1) \wedge$ $((e.v = 0) \rightarrow (c_1.v \geq 2)) \wedge c_1.c$
4	$Type.allInstances()$		$e.v = num\_obj(Type)$
5	e_1 and e_2	b: b,b	$(e.v = \min(e_1.v, e_2.v)) \wedge$ $e_1.c \wedge e_2.c$

operations on these data types; for the rest of data types, $e.v$ represents the result of evaluating e.

Each table entry describes the translation of a specific type of OCL expressions. A special notation is used to refer to concepts in the CSP: $domain(v)$ is the set of potential values of variable v, $domain\_size(v)$ is the number of values in a domain, $num\_obj(T)$ is the CSP variable storing the number of objects of type T and $num\_links(A)$ is the CSP variable storing the number of links in association A.

The CSP size constraint resulting from the abstraction of a given OCL invariant depends on the OCL constructs present in that invariant. The construction of the size constraint proceeds inductively over the structure of the OCL invariant: each subexpression of the invariant is matched with the appropriate table entry and produces a size constraint, whose value may depend on the size constraints of its subexpressions. The size constraint for the entire invariant is the one that corresponds to the root expression, which will be of the form `Type.allInstances()` → `forAll(condition)` (or the root subexpression of the invariant, if the keyword `self` does not appear within the invariant).

In what follows, we describe the size constraint $e.c$ for the second entry in Table 2. Therefore, and following the notation described before, we will denote the OCL expression there ($c_1 \rightarrow exists(e_1)$) as e.

Since e returns a boolean value (*exists* is a boolean expression), the auxiliary variable $e.v$ representing the value of evaluating e must be *false* or *true*. This is expressed in the first condition of $e.c$ as $(0 \leq e.v \leq 1)$. The second condition

in $e.c$ indicates that if the collection is empty $(c_1.v = 0)$ or the condition within the quantifier is *false* $(e_1.v = 0)$, then the result of the existential quantifier will always be *false* $(e.v = 0)$. The third condition in $e.c$ indicates that if the condition within the quantifier is *true* $(e_1.v = 1)$, then the OCL expression will hold $(e.v = (c_1.v \geq 1))$ if the source collection contains at least one element. Finally, $c_1.c$ and $e_1.c$ correspond to the size constraints resulting from the abstraction of the source collection expression, and the condition serving as argument for the *exists* quantifier.

Example 6. Going back to the UML/OCL model from Example 1 and the variables defined in Example 5, let us consider the derivation of CSP constraints from the OCL invariants. This process is illustrated in Figure 4. For each invariant, we consider its abstract syntax tree and identify all its subexpressions (10 in this case). For each OCL subexpression i, a CSP variable $e_i.v$ will be defined, together with a CSP constraint $e_i.c$ on the value of $e_i.v$. This constraint $e_i.c$ is constructed by matching each subexpression against the patterns defined in Table 2. In our example, expressions e_5, e_7 and e_{10} correspond to pattern 1; e_2 and e_3 to pattern 2; e_8 to pattern 3; e_4, e_6 and e_9 correspond to pattern 4; and e_1 to pattern 5.

The CSP constraint for the whole invariant corresponds to the constraint of the root subexpression, in this case e_1 and e_8. Depending on the correctness property being checked, additional constraints need to be added to the CSP. A typical requirement is the *satisfiability* of the UML/OCL model, *i.e.*, requiring the invariant to hold. This is enforced by adding a constraint $(e_i.v = 1)$, where $e_i.v$ is the boolean variable for the root node. In our running example, this means adding the constraints $(e_1.v = 1)$ and $(e_8.v = 1)$.

The CSP constraints that are generated automatically may be very verbose, but it is possible to simplify them for readability using information such as the correctness property. For example, Figure 5 depicts how the CSP constraints in the running example can be simplified. The following CSP constraints would be derived from the OCL invariants:

$$\texttt{Part} \leq \mathit{domain\_size}(\texttt{serial}) \quad (5)$$

$$\texttt{Cutter} \geq 1 \quad (6)$$

$$\texttt{Grinder} \geq 1 \quad (7)$$

where (5) is generated from invariant "UniqueSerials" and (6-7) from "MachineAvailability". Notice that this last simplification step is not required by the bound tightening procedure and it is simply used to present the resulting CSP constraints in a more readable way.

3.4 Tightening bounds

Once a CSP for the UML/OCL model has been constructed, its constraints are used to remove unproductive values from the domains of variables. This process, known as integer bound propagation, is based on the notion of *propagators* [28]. For each constraint, its propagator is a procedure that can tighten the bounds of participating variables. For instance, for an equality $x = y$, the domains of x and y can be tightened to $\text{domain}(x) \cap \text{domain}(y)$ as only the values that can make x and y equal should be considered. Propagators are applied to the set of domains until a fixpoint is reached. This computation is already built-in in all integer interval constraint solvers, which provide optimizations to speed up convergence and threshold parameters to control the amount of effort spent in the process.

Computing an *optimal* solution to this fixpoint is, in general, an NP-complete problem [15]: there are algorithms that exhibit a pseudo-polynomial behavior but may have exponential runtimes in some inputs. However, slow convergence is typically caused by domains with many values, *e.g.*, intervals holding 2^{32} values to encode all values of a 32-bit integer. This is an unlikely scenario because bounded verifiers would be unable to deal with bounds of this size. Again, our work does not target the optimal solution and Section 4 illustrates that in practice bound tightening does not impose a significant overhead in the overall verification.

4 Experimental Results

In this section, we evaluate our method to answer the following questions:

- **Q1:** Does the bound tightening procedure reduce the execution time of UML/OCL bounded verification significantly?
- **Q2:** Is the execution time of the bound tightening procedure negligible with respect to the execution time of UML/OCL bounded verification?

To answer these questions, we have implemented our translation procedure for UML/OCL models into a CSP for propagation as an extension of the EMFtoCSP tool [12]. Our proposed bound tightening procedure has been implemented using the interval solver IC from the ECL^{i}PS^e Constraint Programming System [24].

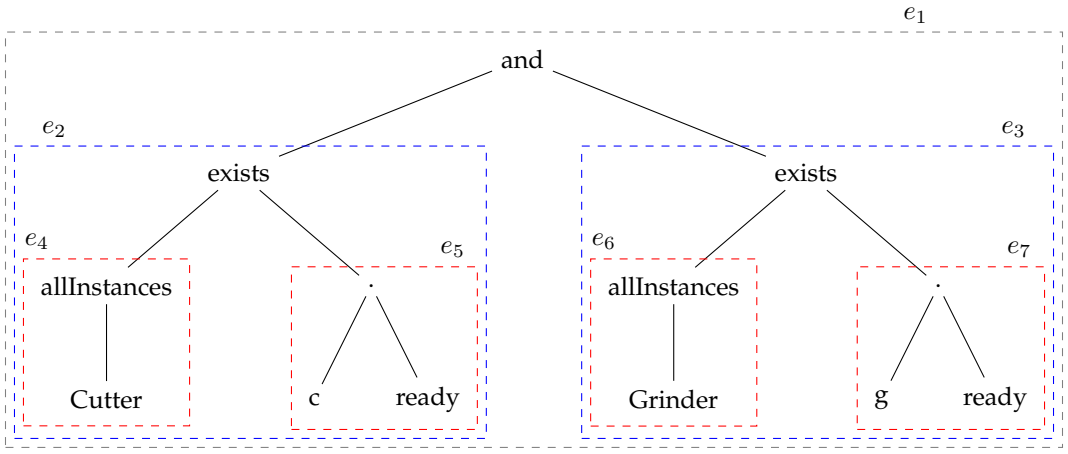
To run the experiments we have selected a third-party tool: the USE model validator plug-in [13]. This tool verifies UML/OCL models by translating them into relational logic formula to be checked using the KodKod relational solver [29], which relies on SAT-solvers like MiniSat [30]. The choice of this particular toolkit has been motivated by its public availability and its competitive execution time results. We also believed it was important to use an external tool to showcase the fact that our bound tightening procedure can be useful to any bounded verification tool for UML/OCL. Therefore our experimental settings consist in the combination of these two tools, the extension of EMFtoCSP for the tightening part and USE for the actual verification tasks.

Using these two tools, we have designed three experiments involving manually created UML/OCL models (Sec. 4.1), a set of randomly generated models (Sec. 4.2) and a fragment of the OCL meta-model (Sec. 4.3). Random models allow us to study the efficiency of bound tightening (Q2), while the other experiments focus on measuring the performance gains in verification when using tightened bounds (Q1).

4.1 Characteristic models

Experimental design. We have considered two UML/OCL models where we attempt to validate whether the model

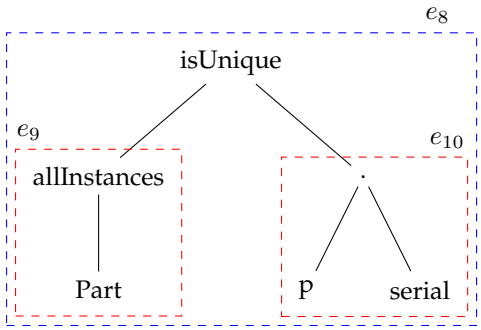
```
context AbstractMachine inv MachineAvailability :
    Cutter.allInstances() → exists(c | c.ready) and
    Grinder.allInstances() → exists(g | g.ready)
```



$e_1.c$: $(e_1.v = min(e_2.v, e_3.v)) \wedge e_2.c \wedge e_3.c$
$e_2.c$: $(0 \leq e_2.v \leq 1) \wedge ((e_4.v = 0 \vee e_5.v = 0) \rightarrow (e_2.v = 0)) \wedge\ ((e_5.v = 1) \rightarrow (e_2.v = (e_4.v \geq 1))) \wedge e_4.c \wedge e_5.c$
$e_3.c$: $(0 \leq e_3.v \leq 1) \wedge ((e_6.v = 0 \vee e_7.v = 0) \rightarrow (e_3.v = 0)) \wedge\ ((e_7.v = 1) \rightarrow (e_3.v = (e_6.v \geq 1))) \wedge e_6.c \wedge e_7.c$
$e_4.c$: $e_4.v = \texttt{Cutter}$
$e_5.c$: $domain(e_5.v) \subseteq domain(\texttt{ready})$
$e_6.c$: $e_6.v = \texttt{Grinder}$
$e_7.c$: $domain(e_7.v) \subseteq domain(\texttt{ready})$

(a)

```
context Part inv UniqueSerials :
    Part.allInstances() → isUnique(p | p.serial)
```



$e_8.c$: $(0 \leq e_8.v \leq 1) \wedge ((e_8.v = 0) \rightarrow (e_9.v \geq 2))$
$\wedge\ ((e_8.v = 1) \rightarrow (domain\_size(e_{10}.v) \geq e_9.v)) \wedge e_9.c \wedge e_{10}.c$
$e_9.c$: $e_9.v = \texttt{Part}$
$e_{10}.c$: $domain(e_{10}.v) \subseteq domain(\texttt{serial})$

(b)

Fig. 4. Abstraction of the OCL invariants `MachineAvailability` (a) and `UniqueSerials` (b). For each invariant, we show the textual OCL constraint, its abstract syntax tree and the corresponding CSP constraints.

$$e_1.v = 1 \xrightarrow{e_1.v = min(e_2.v, e_3.v)} \begin{cases} e_2.v = 1 \xrightarrow{e_4.v=0 \rightarrow e_2.v=0} e_4.v \geq 1 \xrightarrow{e_4.v=\texttt{Cutter}} \texttt{Cutter} \geq 1 \\ e_3.v = 1 \xrightarrow{e_6.v=0 \rightarrow e_3.v=0} e_6.v \geq 1 \xrightarrow{e_6.v=\texttt{Grinder}} \texttt{Grinder} \geq 1 \end{cases}$$

$$e_8.v = 1 \xrightarrow{(e_8.v=1) \rightarrow (domain\_size(e_{10}.v) \geq e_9.v)} domain\_size(e_{10}.v) \geq e_9.v \xrightarrow{e_9.v=\texttt{Part}}$$
$$domain\_size(e_{10}.v) \geq \texttt{Part} \xrightarrow{domain(e_{10}.v) \subseteq domain(\texttt{serial})} domain\_size(\texttt{serial}) \geq \texttt{Part}$$

Fig. 5. Simplifying the CSP constraints in Figure 4 for readability, assuming the goal is checking satisfiability.

TABLE 3
Input UML/OCL models.

Name	Classes	Assocs	Attrs	Inv
Teams	5	4	9(2)	3
Company	6	8	19(2)	16

is *strongly satisfiable*, *i.e.*, whether it is possible to create an instance of each non-abstract class in the model. Table 3 summarizes some features of the models under analysis: the number of classes, associations, attributes (in parenthesis, boolean attributes) and invariants.

The models include a wide variety of UML/OCL features and illustrate two levels of constraint density: "Teams"[1] with few constraints and "Company"[2] with many constraints, so a priori the second one should be harder to verify. Minor changes (*e.g.*, rewriting association classes) were required to adapt the models to the particular syntax requirements of the verification tools. For each UML/OCL model, different sets of input bounds have been considered in order to illustrate the performance of our approach in different scenarios. Three types of bounds are defined: the number of objects in each class, the number of links in each association and the potential values for integer attributes ([0,1] is trivially used for boolean attributes).

For the sake of representativity, we are interested in measuring the performance of verification for both satisfiable and unsatisfiable models. As both examples are satisfiable, we have devised an unsatisfiable version of each one by adding one invariant that cannot be satisfied due to its interaction with the rest of constraints.

We then evaluate the tightened bounds for our UML/OCL models, measuring both the computation time for the bounds and the verification time in USE with (and without) the tightened bounds.

Results. Table 4 summarizes the results obtained in these experiments. Each entry contains the model being analyzed, the initial verification bounds and the execution time (in seconds) of USE with the original bounds (**USE-orig**), of the bound tightening procedure (**Tight**) and of USE with the tightened bounds (**USE-tight**). Regarding the execution time for USE, we further identify the time required by the tool to translate the UML/OCL model into a formula (**Trans**) and the time needed by the solver to check the formula (**Solv**). Finally, we measure the ratio of improvement in the execution time (**Speedup**) as USE-orig divided by Tight+USE-tight (1 if there is no change, higher is better).

As expected, the verification of the "Teams" model is faster than the verification of "Company". Model size (less associations and attributes) and the number of invariants (6 vs 16) are the reasons for this difference.

Therefore, with respect to Q1 (does bound tightening reduce verification time?) the effect of bound tightening is most noticeable in models where verification is most complex. For those examples, significant reductions can be achieved with some examples running 6 times faster. Moreover, bound tightening requires about two seconds in every example. Thus, the overhead generated by bound tightening will only be noticed in those examples where the solver verifies the model in a couple of seconds.

Again, for "easy" models that can be verified quickly, bound tightening may fail to cause any reduction at all or it may be insufficient to compensate the bound tightening overhead. In any case, the performance gains in "hard" instances compensate this small penalty in "easy" instances.

To illustrate what the tightened bounds look like with respect the original ones, we discuss the tightening of the initial bounds [1,5] for classes and [1,10] for associations. For the sake of brevity, we do not discuss attribute bounds:

- **Teams** – 2 of the 5 classes have improved bounds: `Person` to [2,5] and `Team` to [1,2]. With respect to associations, 2 of the 3 associations have improved bounds: `TeamMember` and `MembersInTeam` have bounds [2,5] and `MeetingParticipants` has bounds [2,10].
- **Company** – 5 of the 6 classes have improved bounds: three classes (`Department`, `Project` and the association class `Manages`) have bounds [1,1] while two classes (`Employee` and the association class `WorksOn`) have bounds [4,5]. Regarding associations, there are improvements in 5 out of 8 associations: three of them (`EmployeeManages`, `DepartmentManages` and `DepartmentProject`) have bounds [1,1] and two of them (`EmployeeWorksOn` and `ProjectWorkOn`) have bounds [4,5].

It should be noted that in these experiments the time required for bound tightening is almost the same for all models, regardless of the initial bounds. That is, the amount of reduction in the verification bounds does not have an impact in the performance of the bound tightening procedure. Nevertheless, the performance of bound tightening is analyzed in more detail in the next section with experiments using random UML/OCL models.

4.2 Random models

Experimental design. With respect to Q2 (is the execution time of bound tightening negligible?), we performed a series of experiments in order to measure the efficiency of our bound tightening procedure. In particular, we were interested in assessing the impact of three parameters: (1) the size of the model, *e.g.*, the number of classes; (2) the number of constraints in the model, *e.g.*, the number of association end multiplicity constraints; and (3) the size of the domains used as inputs.

For these experiments, we generated a set of random models using a different number of classes (10, 25, 50, 100 and 200). These models included a UML class diagram with binary associations and its corresponding cardinality constraints (no additional, OCL-based, constraints were added at this point). In these models, any pair of classes has a certain probability of being connected by a binary association (3 %, 6 % and 10 %). The multiplicities of each association end were selected randomly (0..1, 1..1, 0..*, 1..*, 0..N, N..*

1. http://st.inf.tu-dresden.de/files/general/OCLByExampleLecture.pdf
2. http://cs.ulb.ac.be/public/_media/teaching/infoh302/oclnotes.pdf

TABLE 4
Experimental results on the impact of bound tightening in verification time

	Verification Bounds			USE-orig			USE-tight		
Name	**Class**	**Assoc**	**Attrib**	**Trans**	**Solv**	**Tight**	**Trans**	**Solv**	**Speedup**
Teams (sat)	[1, 5]	[1, 10]	[0, 300]	0.7 s	0.8 s	1.9 s	0.7 s	0.4 s	x0.50
	[1, 10]	[1, 20]	[0, 300]	1.5 s	5.5 s	1.9 s	1.5 s	6.7 s	x0.70
	[1, 15]	[1, 30]	[0, 300]	2.4 s	31.2 s	1.9 s	2.3 s	30.4 s	x0.97
Teams (unsat)	[1, 5]	[1, 10]	[0, 300]	0.7 s	0.6 s	1.9 s	0.7 s	0.6 s	x0.41
	[1, 10]	[1, 20]	[0, 300]	1.5 s	3.6 s	1.9 s	1.5 s	6.5 s	x0.50
	[1, 15]	[1, 30]	[0, 300]	2.3 s	39.5 s	1.9 s	2.3 s	31.0 s	x1.18
Company (sat)	[1, 5]	[1, 10]	[0, 300]	2.9 s	5.1 s	2.0 s	1.9 s	1.5 s	x1.46
	[1, 10]	[1, 20]	[0, 300]	6.2 s	10.5 s	2.1 s	4.9 s	26.1 s	x0.51
	[1, 15]	[1, 30]	[0, 300]	14.2 s	275.8 s	2.0 s	14.9 s	311.7 s	x0.88
Company (unsat)	[1, 5]	[1, 10]	[0, 300]	1.8 s	0.5 s	2.1 s	1.5 s	2.7 s	x0.36
	[1, 10]	[1, 20]	[0, 300]	5.7 s	30.8 s	2.1 s	1.4 s	2.1 s	x6.41
	[1, 15]	[1, 30]	[0, 300]	14.1 s	80.4 s	2.1 s	4.8 s	9.3 s	x5.82

Settings

Computer	Intel i5-760 2.8 GHz 4 GB RAM
OS & Java	Windows 10 64 bits, JDK 8u121
USE	v4.2, Solver MiniSat with bitwidth=32
ECL^{i}PS^e	v6.1 64 bits

TABLE 5
Experimental results on the scalability of bound tightening

(a) Execution time of bound tightening (seconds)

		Bounds				
%A	**N**	[1,10]	[1,100]	[1,250]	[1,1000]	[1,5000]
3 %	10	1.7 s	1.7 s	1.8 s	1.5 s	1.8 s
	25	1.7 s	1.8 s	2.0 s	1.6 s	1.7 s
	50	1.9 s	2.1 s	2.2 s	1.9 s	1.9 s
	100	2.6 s	3.1 s	3.7 s	2.8 s	5.1 s
	200	5.9 s	7.9 s	8.9 s	47.9 s	65.1 s
6 %	10	1.5 s	1.7 s	2.4 s	1.7 s	2.4 s
	25	1.7 s	1.9 s	2.6 s	1.8 s	2.3 s
	50	2.0 s	2.8 s	3.6 s	22.4 s	871.6 s
	100	3.4 s	3.6 s	4.0 s	7.8 s	14.3 s
	200	10.0 s	11.7 s	11.7 s	25.9 s	26.5 s
10 %	10	1.6 s	2.1 s	1.9 s	1.6 s	1.6 s
	25	1.7 s	4.7 s	4.8 s	1.7 s	1.9 s
	50	2.1 s	3.1 s	3.9 s	113.0 s	335.4 s
	100	4.2 s	5.5 s	5.3 s	5.2 s	9.3 s
	200	15.6 s	18.2 s	16.3 s	16.8 s	17.8 s

(b) Success rate (% of experiments with improved bounds)

		Bounds				
%A	**N**	[0,10]	[1,100]	[1,250]	[1,1000]	[1,5000]
3%	10	100 %	100 %	100 %	100 %	100 %
	25	40 %	80 %	100 %	100 %	100 %
	50	0 %	60 %	80 %	80 %	100 %
	100	0 %	0 %	0 %	0 %	20 %
	200	0 %	0 %	0 %	0 %	0 %
6%	10	80 %	100 %	100 %	100 %	100 %
	25	0 %	80 %	80 %	100 %	100 %
	50	0 %	0 %	20 %	20 %	40 %
	100	0 %	0 %	0 %	0 %	0 %
	200	0 %	0 %	0 %	0 %	0 %
10%	10	20 %	100 %	100 %	100 %	100 %
	25	0 %	0 %	0 %	60 %	80 %
	50	0 %	0 %	0 %	0 %	0 %
	100	0 %	0 %	0 %	0 %	0 %
	200	0 %	0 %	0 %	0 %	0 %

Settings See Table 4

Parameters

N	Number of classes in the UML model
%A	Percentage of pairs of classes connected by an association
Bounds	Input bounds to the bound tightening procedure

or $N..M$) with N and M between 2 and 10. Finally, five different input bounds for the population of classes and the number of links in associations were used: [1,10], [1,100], [1,250], [1,1000] and [1,5000]. The largest bounds were used to illustrate the limits of this approach, but they are not representative of SAT-based tools for model verification, which typically suggest (or use by default) much smaller bounds, *e.g.*, 3, 5 or 10 values in [31].

For each of these parameters, we measure the execution time of bound propagation. For each combination of model size and probability of associations, 5 random models were created (*i.e.*, 75 models in total). Bound propagation was applied in each model using the five different input bounds (*i.e.*, 375 executions). Then, each execution was performed 5 times, averaging the results, in order to avoid potential interference from other processes (*i.e.*, $1,875$ runs).

Results. The results of the experiments with random models are displayed in Table 5. For each experiment, we measured the execution time of bound tightening in seconds (Subtable 5(a)) and we tracked whether bound tightening managed to reduce the input bounds (Subtable 5(b)).

Subtable 5(a) reports the execution time for models with

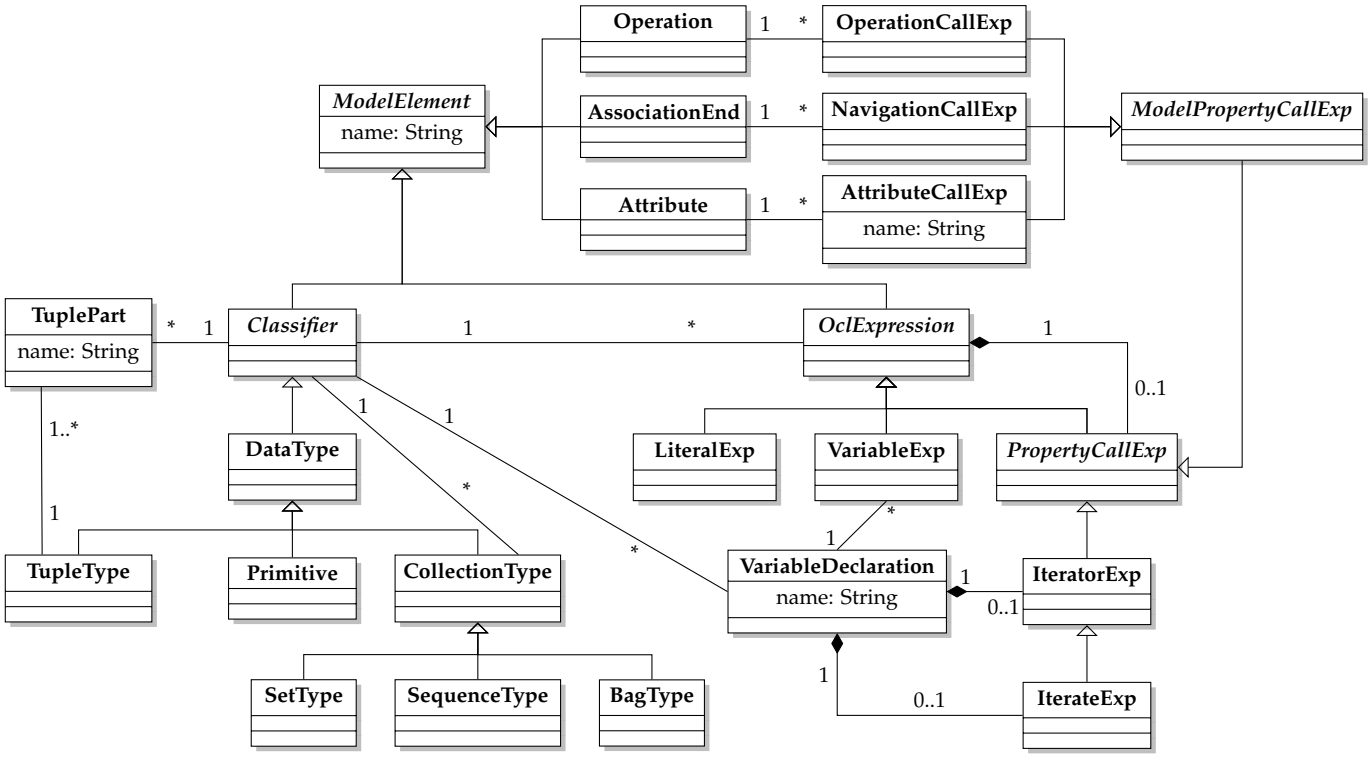


Fig. 6. A fragment of the OCL meta-model (several associations involving `OclExpression` are not depicted to improve readability)

a given size (N) and connectivity (%A), considering a set of input bounds. As we have created five models for each size and connectivity, each entry contains the average execution time (in seconds) for these five models. Results show execution times of around 1–2 seconds for small models (10–25 classes) and 2–5 seconds for medium-sized models (50–100 classes). Furthermore, all the parameters under consideration have an impact on performance: as a general rule, adding more classes, more constraints or wider bounds increases the time required to tighten bounds. However, specific worst-case instances require more resources than larger instances (*e.g.*, the model with 50 classes and 6% connectivity).

In any case, this worst-case behavior can only be observed by selecting input bounds that are beyond the capabilities of state-of-the-art SAT-based tools for model verification. For instance, computing an instance of the running example in Figure 1 (with 4 classes and 1 association) using a SAT-based solver requires less than a second with bounds [1,10], 2.8 seconds with bounds [1,25] and more than one hour with bounds [1,100]. Hence, bound tightening is fast and its worst cases are still orders of magnitude faster than UML/OCL bounded verification in the corresponding instances.

On the other hand, Subtable 5(b) specifies in which experiments bound propagation managed to infer tighter bounds with respect to the input ones. It is unclear how representative random models are of models written by practitioners. However, some interesting observations can be reported. First, propagation is more successful when input bounds are large with respect to multiplicities of associations, *e.g.*, a multiplicity of 1..2 in an association end is more helpful than a multiplicity 1..10 if the input bounds are [0,10]. And second, propagation is more successful when constraints are local, *i.e.*, there are several constraints involving the same model elements, rather than many constraints involving different model elements.

4.3 OCL meta-model case study

Experimental design. To consider a more realistic case study, we have studied a fragment of the OCL meta-model (depicted in Figure 6) which describes features of the OCL language such as classifiers or expressions and model elements like attributes or association ends. This meta-model is provided as one of the examples within the distribution of the tool USE (`examples\MetaModels\OCL2MM`). It includes 24 classes (including several abstract classes) and 15 associations (including some compositions), but is does not include relevant OCL invariants. As our experiment, we will check whether this meta-model is satisfiable and can be instantiated.

Results. This check has been performed using the USE model validator plug-in (configured to use the SAT solver MiniSat with a bit-width of 8) using our hardware set-up (Intel i5-760 2.8 GHz processor with 4 GB RAM). Using bounds [1,5] for classes and associations, the SAT solver states that the model is unsatisfiable. It is necessary to increase the bounds up to [1,10] before the SAT solver can find a solution. However, the solver requires more than half an hour (1,938 seconds) to perform this computation.

With the use of bound tightening, we will explore a more efficient way to find a solution. We will focus on the central class of this meta-model, the abstract class `ModelElement` which is at the root of the inheritance hierarchy. In this

class, we will explore potential bounds for the number of instances of all of its subclasses.

First of all, let us consider what information can be directly inferred from the class diagram. Running bound tightening with initial bounds of $[1, \infty]$ in all non abstract classes, we find out in less than 0.1 seconds that our solution requires at least 14 instances of subclasses of `ModelElement`. This is pretty straightforward as the inheritance hierarchy of `ModelElement` includes 14 non-abstract subclasses.

With this information, a designer can request a solution with at most 15 `ModelElement` in total and at most 10 objects in each non-abstract class. Providing these initial bounds to bound tightening, a more precise set of bounds can be obtained. These tightened bounds make the verification of this model finish in only 67 seconds, 28.9 times faster than using a brute-force strategy for setting bounds. This is a significant result as we are saving more than 30 minutes of verification time on a realistic model.

5 Discussion

In this Section, three issues are discussed: *how* bound tightening should be used, *i.e.*, in which ways it can interact with UML/OCL verification tools (Sec. 5.1); *when* it should be used, *i.e.*, in which types of UML/OCL models it will be more successful (Sec. 5.2); and *where* it should be used, *i.e.*, what software engineering problems are well-suited for the use of this method (Sec. 5.3).

5.1 Primary usage scenarios

We consider three potential usage scenarios of bound tightening in the verification of software models: *inference*, *optimization* and *interactive selection*.

Inference. The first usage scenario is to automatically infer finite bounds for a given input model, without designer intervention. This scenario is the most desirable one, as the output of the bounded verification tool would be valid for any potential bound, *i.e.*, the lack of an example (resp. counterexample) within the finite bounds would constitute a refutation (resp. proof) of the property being verified. However, this theoretical scenario is unlikely to occur in practice as the models would need to be highly constrained in order to bound all domains.

Optimization. Instead, the most promising scenario is using bound tightening to automatically refine the finite bounds provided by the user. In this way, bound tightening acts as an "optimization" step executed before the bounded verification tool. This optimization is performed transparently to both the designer and the bounded verification tool.

Interactive selection. Finally, bound tightening can be used interactively to aid the user in the selection of proper verification bounds. The bound selection process is divided into a sequence of decisions, each consisting of choosing finite bounds for one of the remaining parameters of the verification problem (*e.g.*, population size for a given class). Bound tightening can help designers by tightening bounds after each decision, possibly assigning values to some parameters automatically.

For this purpose, bound tightening should be complemented by an heuristic procedure that selects the order in which decisions must be taken, in order to maximize the amount of information that can be automatically inferred by bound tightening. That is, the designer should start assigning those parameters that have a larger impact on the values of other parameters. Even though an in-depth discussion of those heuristics is out of the scope of this paper, these heuristics are closely related to ones used by constraint solvers in order to decide the order in which variables of a CSP are given a value [32].

5.2 Target models

The performance gains offered by the bound tightening procedure on UML/OCL model verification will depend on a variety of factors. In the following, we discuss under what conditions bound tightening will be most effective:

Original domain bounds. If the initial domains are very small, the verification is typically fast and the speedup provided by this technique might not be noticeable. For larger domains, the speedup can become significant.

Hardness of verification. In order to find a witness (or conclude that there is none), the solver will evaluate partial solutions in order to decide whether they can be extended until a complete solution is reached. For some models and properties, the solver may be able to reach its conclusion by evaluating few partial solutions. In contrast, in other models the solver will need to explore many candidates. Given that bound tightening aims to prune the space of candidate solutions, the speedup it offers can be greater in models where many candidate solutions need to be explored.

Number (and strictness) of constraints in the model. If the model has few or weak constraints (*e.g.*, multiplicities "0..*"), this approach may fail to reduce bounds in a noticeable way. Conversely, tightening can be most effective in highly constrained models.

That is, the benefits of our approach may be most noticeable in the models that take longer to verify and thus those where the user can benefit most from a speedup. Moreover, in models where bound tightening does not produce any speedup, the overhead it incurs is negligible. For these reasons, bound tightening can be a valuable addition to any bounded verification framework for UML/OCL.

5.3 Other Software Engineering activities

Beyond the main usage scenarios described above, bound tightening can also be effective in situations where a partial instance that needs to be extended is available. One scenario is *incremental verification* [33], where the correctness property has already been checked in a submodel or a similar model where some changes have been applied. Another suitable scenario is *integrity repair* [34], where an instance that violates some integrity constraints needs to be repaired with the least amount of modifications.

Furthermore, if our goal is *testing or verifying model transformations* [35], transformation rules can provide hints about the necessary model sizes for some fragments of the model. For instance, in graph transformation rules, the left-hand

side and right-hand side of a rule indicate the minimum number of objects required to apply a rule and the number of objects that are created, modified or deleted as a result.

All these scenarios share a common trait: they can be restated as the search of suitable target instances of a model, e.g. faults, test cases, ... A bounded verification tool can be used as a model finder to compute these target instances. Fortunately, the nature of the problem provides some clues about the size of the target instances. For instance, if we want to restore integrity to an instance by only deleting elements, we already know an upper bound for the target instance. Conversely, if we are testing a rule in a model transformation, test cases should include at least the necessary objects required to apply the rule, so we have a lower bound. This partial information about the target instance can be leveraged to speed up the model finder: tightening can propagate these initial bounds to further reduce the search space.

6 Related Work

The verification of UML class diagrams is an EXPTIME-hard problem, which becomes undecidable with the inclusion of general OCL constraints [4]. This implies that UML/OCL verification tools need to decide whether they will support OCL, and if so, up to what extent [5].

Some approaches focus on decidable verification problems either by excluding OCL [6], [26], [36] or by restricting OCL to a decidable subset [37], [38]. Among the methods supporting general OCL constraints, *e.g.*, [20], [39], bounded verification is a popular strategy [9], [10], [11], [13], [14]. Efficiency, lack of user intervention and expressiveness are its advantages, while the need to define search boundaries and inconclusive answers outside these bounds are its drawbacks.

Given that UML/OCL verification is undecidable, it is not possible to establish verification bounds a priori. However, it becomes feasible if we exclude OCL entirely [36] or we restrict its expressivity, as in OCL-Lite [38]. Similarly, even though first-order logic is undecidable in general, some fragments of first-order logic are known to be decidable, such as "effectively propositional logic". This fragment can also be defined in the context of the many-sorted first-order logic used in Alloy [40]. In specifications within these restricted notations, verification is decidable and it is possible to compute exact verification bounds. Nevertheless, these notations do not support OCL features such as constraints over integers (*e.g.*, $x > y$), the size of collections (operations $size$, $count$ or sum) or arbitrary multiplicities in associations (any upper bound in the case of OCL-Lite or any bound outside 0, 1 and * for effectively propositional logic), among others. Thus, none of the example models described in this paper (not even the running example) falls within the scope of these methods.

Several mechanisms can be used to accelerate bounded verification: *parallelization* (use of several solvers running in parallel over different parts of the formula or the domains), *slicing* (partition the problem into independent components that can be analyzed separately) and *bound tightening/reduction* (reduce the size of the verification bounds). In the context of UML/OCL verification, [41], [42] describe slicing techniques to partition class diagrams, while ParAlloy [43] and Ranger [44] study the parallel verification of Alloy models. Considering UML class diagrams without OCL, [45], [46] study the potential interactions among association multiplicities to detect situations where multiplicities can be strengthened or are unsatisfiable. In [47], preliminary ideas on the automatic determination of verification bounds in a UML model are presented, with the analysis of OCL constraints left as future work. However, to the best of our knowledge, this paper is the first work addressing bound reduction for the verification of UML/OCL models.

In contrast, bound reduction is a well-studied problem in another field: static program analysis. Techniques such as *interval analysis* or *shape analysis* can infer bound information about the program variables which can later be used when verifying the dynamic behavior of programs. For variables of type string, it is also typical to perform size analysis to determine valid string lengths before searching for feasible values [48].

In the context of static analysis, the most related tool in terms of bound reduction is TACO [49], a tool for the verification of JML-annotated Java programs. TACO attempts to eliminate individual values from domains by calling the solver with a specially tailored formula for each value *before* analyzing the entire program. This allows a more fine-grained bound refinement than using intervals but, on the other hand, a time threshold must be set to avoid wasting too much time on each call.

Other types of optimizations have been proposed for the model checking of hybrid systems. *Domain reduction abstraction* [50] partitions the input domains into equivalence classes with the same behavior. Then, only one representative value from each equivalence class needs to be considered during the analysis. Nevertheless, the number of equivalence classes is exponential with respect of the number of atomic properties appearing in the system being analyzed: if there are n properties, each combination of properties (and their negations) defines an equivalence class, *i.e.*, 2^n classes. Another approach is *CounterExample-Guided Abstraction Refinement* (CEGAR) [51], which studies an abstracted version of the system under analysis. Lack of failures in the abstraction proves the correctness of the original system. However, failures in the abstraction may be an artifact of the abstraction process and need to be double-checked. Spurious failures are removed by refining the abstraction and the analysis resumes on the new abstraction. In any case, even though CEGAR shares the goal of improving the efficiency of verification, it does not deal with bound reduction.

7 Conclusions

We have introduced a novel technique to aid in the bounded verification of UML/OCL models. This approach aims to assist users in the selection of verification bounds, a task which currently lacks adequate support. The proposed method operates by translating the UML/OCL model into a constraint satisfaction problem, focusing only on the information relevant to infer domain bounds. Then, interval constraint propagation techniques are used to tighten the

domain bounds. Smaller bounds produced can reduce the verification time significantly.

This approach can be used in different ways: as a preprocessing stage before verification or as part of an interactive process to guide the choice of bounds. As future work, we plan to investigate heuristics regarding the best order for selecting bounds, *i.e.*, one that reduces the number of choices and maximizes the amount of information that can be inferred automatically by bound propagation. We also intend to investigate how to reverse this approach, *e.g.*, by broadening (instead of tightening) user provided bounds which are too strict to find a counterexample.

ACKNOWLEDGMENTS

This work is partially funded by the H2020 ECSEL Joint Undertaking Project "MegaM@Rt2: MegaModelling at Runtime" (737494), the Spanish Ministry of Economy and Competitivity through the project "Open Data for All: an API-based infrastructure for exploiting online data sources" (TIN2016-75944-R) and a research grant from the Internet Interdisciplinary Institute (IN3) at UOC. The authors would like to thank the anonymous reviewers and Dr. Tim Nelson for their valuable comments.

REFERENCES

[1] M. Petre, "UML in practice," in *ICSE '13*, 2013, pp. 722–731.
[2] E. Torlak, M. Taghdiri, G. Dennis, and J. Near, "Applications and extensions of Alloy: Past, present, and future," *Mathematical Structures in Computer Science*, vol. 23, pp. 915–933, 2013.
[3] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in Model-Driven Engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014.
[4] D. Berardi, D. Calvanese, and G. D. Giacomo, "Reasoning on UML class diagrams," *Artificial Intelligence*, vol. 168, no. 1-2, pp. 70–118, 2005.
[5] C. A. González and J. Cabot, "Formal verification of static software models in MDE: A systematic review," *Information and Software Tech.*, vol. 56, no. 8, pp. 821–838, 2014.
[6] M. Cadoli, D. Calvanese, G. D. Giacomo, and T. Mancini, "Finite satisfiability of UML class diagrams by Constraint Programming." in *DL'2004*, ser. CEUR Workshop Proc., vol. 104, 2004.
[7] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, "Generating test data from OCL constraints with search techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1376–1402, 2013.
[8] A. D. Brucker and B. Wolff, "The HOL-OCL book," ETH Zurich, Tech. Rep. 525, 2006.
[9] 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.
[10] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from UML to Alloy," *Software and Systems Modeling*, vol. 9, no. 1, pp. 69–86, 2010.
[11] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using boolean satisfiability," in *DATE'2010*. IEEE, 2010, pp. 1341–1344.
[12] C. A. González, F. Büttner, R. Clarisó, and J. Cabot, "EMFtoCSP: A tool for the lightweight verification of EMF models," in *FormSERA 2012*, 2012, pp. 44–50.
[13] M. Kuhlmann and M. Gogolla, "From UML and OCL to relational logic and back," in *MODELS'2012*, ser. LNCS, vol. 7590. Springer, 2012, pp. 415–431.
[14] S. Maoz, J. O. Ringert, and B. Rumpe, "CD2Alloy: Class diagrams analysis using Alloy revisited," in *MODELS'2011*, 2011, pp. 592–607.
[15] L. Bordeaux, G. Katsirelos, N. Narodytska, and M. Y. Vardi, "The complexity of integer bound propagation," *Journal of Artificial Intelligence Research (JAIR)*, vol. 40, pp. 657–676, 2011.
[16] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, "Evaluating the "small scope hypothesis"," MIT CSAIL, Tech. Rep., 2003.
[17] J. Oetsch, M. Prischink, J. Pührer, M. Schwengerer, and H. Tompits, "On the small-scope hypothesis for testing answer-set programs," in *KR'12*. AAAI Press, 2012, pp. 43–53.
[18] F. Yu, T. Bultan, and E. Peterson, "Automated size analysis for OCL," in *FSE'2007*. ACM, 2007, pp. 331–340.
[19] R. Clarisó, C. A. González, and J. Cabot, "Towards domain refinement for UML/OCL bounded verification," in *SEFM' 2015*, ser. LNCS, vol. 9276. Springer, 2015, pp. 108–114.
[20] A. Queralt and E. Teniente, "Verification and validation of UML conceptual schemas with OCL constraints," *ACM Transactions on Software Engineering Methodology*, vol. 21, no. 2, pp. 13:1–13:41, 2012.
[21] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, independence and consequences in UML and OCL models," in *TAP'2009*. Springer, 2009, pp. 90–104.
[22] M. Gogolla, L. Hamann, and M. Kuhlmann, "Proving and visualizing OCL invariant independence by automatically generated test cases," in *TAP'2010*. Springer, 2010, pp. 38–54.
[23] N. Przigoda, R. Wille, and R. Drechsler, "Leveraging the analysis for invariant independence in formal system models," in *DSD'2015*, 2015, pp. 359–366.
[24] K. R. Apt and M. Wallace, *Constraint Logic Programming using ECLiPSe*. Cambridge University Press, 2007.
[25] P. V. Hentenryck, Y. Deville, and C.-M. Teng, "A generic arc-consistency algorithm and its specializations," *Artificial Intelligence*, vol. 57, no. 2, pp. 291–321, 1992.
[26] M. Balaban and A. Maraee, "Finite satisfiability of UML class diagrams with constrained class hierarchy," *ACM Transactions on Software Engineering Methodology*, vol. 22, no. 3, p. 24, 2013.
[27] M. Gogolla and M. Richters, "Expressing UML class diagrams properties with OCL," in *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, ser. LNCS, vol. 2263. Springer, 2002, pp. 85–114.
[28] C. Choi, W. Harvey, J. H. M. Lee, and P. Stuckey, "Finite domain bounds consistency revisited," in *AI'2006*. Springer, 2006, pp. 49–58.
[29] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *TACAS'2007*, ser. LNCS, vol. 4424. Springer, 2007, pp. 632–647.
[30] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT'2003*. Springer, 2003, pp. 502–518.
[31] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
[32] K. Apt, *Principles of Constraint Programming*. Cambridge University Press, 2003.
[33] H. Bagheri and S. Malek, "Titanium: Efficient analysis of evolving Alloy specifications," in *FSE'2016*. ACM, 2016, pp. 27–38.
[34] N. Macedo, T. Jorge, and A. Cunha, "A Feature-based Classification of Model Repair Approaches," *IEEE Transactions on Software Engineering*, 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7605502/
[35] B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Communications of the ACM*, vol. 53, no. 6, pp. 139–143, 2010.
[36] M. Cadoli, D. Calvanese, G. D. Giacomo, and T. Mancini, "Finite model reasoning on UML class diagrams via Constraint Programming," in *AI*IA 2007*. Springer Berlin Heidelberg, 2007, pp. 36–47.
[37] M. Clavel, M. Egea, and M. A. G. de Dios, "Checking unsatisfiability for OCL constraints," *ECEASST*, vol. 24, 2009.
[38] A. Queralt, A. Artale, D. Calvanese, and E. Teniente, "OCL-Lite: Finite reasoning on UML/OCL conceptual schemas," *Data & Knowledge Engineering*, vol. 73, pp. 1–22, 2012.
[39] A. D. Brucker and B. Wolff, "HOL-OCL: A formal proof environment for UML/OCL," in *FASE 2008*, ser. LNCS, vol. 4961. Springer, 2008, pp. 97–100.
[40] T. Nelson, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Toward a more complete Alloy," in *ABZ'2012*, 2012, pp. 136–149.
[41] A. Shaikh, R. Clarisó, U. K. Wiil, and N. Memon, "Verification-driven slicing of UML/OCL models," in *ASE'2010*. ACM, 2010, pp. 185–194.
[42] J. Seiter, R. Wille, M. Soeken, and R. Drechsler, "Determining relevant model elements for the verification of UML/OCL specifications," in *DATE'2013*. EDA Consortium, 2013, pp. 1189–1192.
[43] N. Rosner, J. P. Galeotti, C. L. Pombo, and M. F. Frias, "ParAlloy: Towards a framework for efficient parallel analysis of Alloy

models," in *ABZ'2010*, ser. LNCS, vol. 5977. Springer, 2010, pp. 396–397.
[44] N. Rosner, J. H. Siddiqui, N. Aguirre, S. Khurshid, and M. F. Frias, "Ranger: Parallel analysis of Alloy models by range partitioning," in *ASE'2013*. IEEE, 2013, pp. 147–157.
[45] I. Feinerer, G. Salzer, and T. Sisel, "Reducing multiplicities in class diagrams," in *MODELS 2011,*, ser. LNCS, vol. 6981. Springer, 2011, pp. 379–393.
[46] M. Balaban and A. Maraee, "Simplification and correctness of UML class diagrams - focusing on multiplicity and aggregation/composition constraints," in *MODELS'2013*, ser. LNCS, vol. 8107. Springer, 2013, pp. 454–470.
[47] M. Soeken, R. Wille, and R. Drechsler, "Towards automatic determination of problem bounds for object instantiation in static model verification," in *MoDeVVa'2011*. ACM, 2011.
[48] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *TACAS'2009*. Springer, 2009, pp. 307–321.
[49] J. P. Galeotti, N. Rosner, C. G. L. Pombo, and M. F. Frias, "Taco: Efficient SAT-based bounded verification using symmetry breaking and tight bounds," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1283–1307, 2013.
[50] Y. Choi and M. Heimdahl, "Model checking software requirement specifications using domain reduction abstraction," in *ASE'2003*. IEEE, 2003, pp. 314–317.
[51] E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald, "Abstraction and counterexample-guided refinement in model checking of hybrid systems," *Int. Journal on Foundations of Computer Science*, vol. 14, no. 04, pp. 583–604, 2003.

Robert Clarisó received his BSc (2000) and PhD (2005) in Computer Science from UPC-Barcelona Tech. Since 2005, he is a lecturer at the IT, Multimedia and Telecommunications Department of Universitat Oberta de Catalunya (UOC). He is also a member of the SOM Research Lab within the Internet Interdisciplinary Institute (IN3-UOC). His research interests include formal methods, model-driven engineering and tools for e-learning.

Carlos A. González received his PhD degree from the École des Mines de Nantes (EMN) in 2014. Before steering his career toward research related positions, Carlos worked in the software industry for almost 10 years. Since October 2016, he works as a research associate in the Software Verification and Validation Lab, at the SnT Centre for Security, Reliability and Trust of the University of Luxembourg. His research interests include, but are not limited to, model-driven engineering and software verification and validation.

Jordi Cabot received his PhD degree in Computer Science from Universitat Politècnica de Catalunya (UPC) in 2006 and his Habilitation (French HdR) from the École Doctorale in Nantes in 2012. He has been a visiting researcher in Milan (Politecnico di Milano) and Toronto (University of Toronto) and an Associate Professor and Inria International Chair at École des Mines de Nantes where he led an Inria Research team in Software Engineering. Since May 2015, he is an ICREA Research Professor at Internet Interdisciplinary Institute (IN3), a research center of the Universitat Oberta de Catalunya (UOC), where he leads the SOM Research Lab. Beyond his core research activities, he tries to book some time for blogging and other dissemination and technology transfer actions.