

Practical Constraint Solving for Generating System Test Data

GHANEM SOLTANA, University of Luxembourg

MEHRDAD SABETZADEH, School of EECS, University of Ottawa & SnT, University of Luxembourg

LIONEL C. BRIAND, School of EECS, University of Ottawa & SnT, University of Luxembourg

The ability to generate test data is often a necessary prerequisite for automated software testing. For the generated data to be fit for its intended purpose, the data usually has to satisfy various logical constraints. When testing is performed at a system level, these constraints tend to be complex and are typically captured in expressive formalisms based on first-order logic. Motivated by improving the feasibility and scalability of data generation for system testing, we present a novel approach, whereby we employ a combination of metaheuristic search and Satisfiability Modulo Theories (SMT) for constraint solving. Our approach delegates constraint solving tasks to metaheuristic search and SMT in such a way as to take advantage of the complementary strengths of the two techniques. We ground our work on test data models specified in UML, with OCL used as the constraint language. We present tool support and an evaluation of our approach over three industrial case studies. The results indicate that, for complex system test data generation problems, our approach presents substantial benefits over the state of the art in terms of applicability and scalability.

Additional Key Words and Phrases: System Testing, Test Data Generation, Model-driven Engineering, UML, OCL, Metaheuristic Search, SMT.

ACM Reference Format:

Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. 2020. Practical Constraint Solving for Generating System Test Data. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2020), 47 pages. <https://doi.org/10.1145/3381032>

1 INTRODUCTION

Test data generation is one of the most important and yet most challenging aspects of software test automation [1, 48, 52, 54]. To be useful, test data often needs to satisfy logical constraints. This makes constraint solving an integral part of test data generation. For example, in white-box testing, one may be interested in achieving path or state coverage. Doing so requires solving the path constraints for the different control paths in a program [52]. Similarly, in black box testing, one might require solving complex constraints to generate test data that comply with the system specifications. Otherwise, trivially invalid data inputs would be frequently and easily discarded by the system under test.

Test data tends to get progressively more complex as one moves up the testing ladder, from unit to integration to system testing. Particularly, system testing, whose goal is ensuring that a system

This work is supported by the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement number 694277).

Authors' addresses: Ghanem Soltana, University of Luxembourg; Mehrdad Sabetzadeh, School of EECS, University of Ottawa & SnT, University of Luxembourg; Lionel C. Briand, School of EECS, University of Ottawa & SnT, University of Luxembourg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

1049-331X/2020/1-ART1 \$15.00

<https://doi.org/10.1145/3381032>

is in compliance with its requirements [4], involves exercising end-to-end system behaviors. This, in turn, typically requires numerous interdependent data structures, and thus the ability to account for the well-formedness and semantic correctness constraints of these data structures.

For example, system testing of an application that calculates citizens' taxes would require meaningful assemblies of many inter-related entities, including taxpayers, incomes, dependent family members, and so on. When available, real data, e.g., real tax records in the above example, may be utilized for system testing. In many practical settings, however, real data is (1) incomplete, meaning that the data is insufficient for exercising every system behavior that needs to be tested, (2) incompatible, meaning that, due to changes in the system, the real data no longer matches the system's data structures, or (3) inaccessible, meaning that the real data cannot be used for testing due to reasons such as data protection and privacy.

Because of these factors, system testing is done, by and large, using synthetic data. Generating synthetic data for system testing necessitates constraint solving over system concepts, including concepts related to a system's environment [42]. The languages used for expressing system-level constraints are feature-rich. Languages based on first-order logic are particularly common, noting that quantification, predicates and functions are usually inevitable when one attempts to specify constraints at a system level. This high degree of expressiveness often comes at the cost of making constraint solving undecidable in general, and computationally expensive even when the size of the data to generate is bounded [44, 51].

Our work in this article is prompted by the need to devise a *practical* solution for generating system test data based on a set of complex constraints. To this end, we propose a novel constraint solving approach. We ground our approach on UML [56] and its constraint language, OCL [55]. This choice is motivated by UML's widespread use in industry. OCL supports, among other features, quantification and a variety of arithmetic, string and collection operations.

Our approach leverages earlier strands of work where (metaheuristic) search is applied for solving OCL constraints [2, 3, 77]. In the context of testing, these earlier strands were motivated by the limitations – in terms of scalability and the UML/OCL constructs covered – of commonly used exhaustive approaches based on SAT solving, e.g., Alloy [44], and traditional constraint programming, e.g., UMLtoCSP [16].

The core idea behind our approach is to further enhance the performance of search-based OCL solving by combining it with Satisfiability Modulo Theories (SMT) [10]. Specifically, we observe that SMT solvers, e.g., Z3 [24], have efficient decision procedures for several background theories, e.g., linear arithmetic [27]. While these background theories per se are not sufficient for solving complex OCL constraints, the following prospect exists: if a subformula f in a constraint is solvable by a combination of background theories, then the associated decision procedures are likely to handle f more efficiently than search. What we set out to do in this article is to provide, based on the above prospect, a fully fledged combination of search and SMT for OCL solving.

Contributions. The contributions of this article are three-fold:

- 1) We develop a *hybrid* approach for solving OCL constraints using search and SMT. The approach first transforms OCL constraints into a normal form, known as the Negation Normal Form (NNF) [65]. NNF is key to our approach by enabling an effective distribution of constraint solving tasks across search and SMT. Although our focus in this article remains on OCL, the principles and practical considerations underlying our solving strategy are general, and can be applied to other constraint languages.

- 2) We evaluate our approach on *three industrial case studies* from distinct application domains. The case studies are all concerned with the generation of data for system testing. In each case, we

automatically produce synthetic test data of varying sizes. Our results indicate that our approach is able to solve complex OCL constraints while generating, in practical time, fairly large data samples. We further compare against alternative solvers, and demonstrate that our approach leads to significant improvements in applicability and scalability. It is important to note that our evaluation is not aimed at assessing the effectiveness of different test strategies, but rather at assessing the feasibility and efficiency of automatically generating system test data regardless of test strategy.

3) In support of our approach, we develop a tool named PLEDGE (Practical and Efficient Data Generator for UML). The tool is publicly available at <https://sites.google.com/view/hybridoclsolver/>. To facilitate the replication of our empirical results and also to support future research on test data generation, we provide on the above website our (sanitized) case study material as well.

Structure. Section 2 presents a running example. Section 3 provides background. Section 4 outlines our approach. Sections 5 and 6 elaborate the technical aspects of the approach. Section 7 describes our evaluation. Section 8 discusses limitations and threats to validity. Section 9 compares with related work. Section 10 concludes the article.

2 RUNNING EXAMPLE

In this section, we present an example which we use throughout the article for illustration. For several practical reasons, system test generation is usually black-box and requires an explicit test data model [81]. In UML, test data models can be expressed with relative ease using Class Diagrams (CD) augmented with OCL constraints [84]. The example, shown in Fig. 1, is a small and simplified excerpt of the system test data model for the Government of Luxembourg's personal income tax management application. The full data model is the subject of one of our case studies in Section 7.

The graphical notation of a CD already places certain constraints on how the classes and associations in a CD can be instantiated [56]. One basic constraint is that abstract classes, e.g., *Income* in the CD of Fig. 1(a), cannot be instantiated. A second basic constraint is that the objects on the two ends of a link (association instance) must conform to the CD. For example, a *supports* link should have a *Taxpayer* object on one end and a *Child* object on the other. A CD may additionally have multiplicity constraints. For example, the multiplicities attached to the two ends of the *earns* association impose the following constraints: (1) each *Taxpayer* object must be linked to at least one object of kind *Income*; and (2) each object of kind *Income* must be linked to exactly one *Taxpayer* object.

The constraints that cannot be captured graphically are defined in OCL, as we illustrate in Fig. 1(b). Constraint *C1* states that the age range for individuals is within the range [0..100]. *C2* states that the disability rate is within the range (0..1] when the individual is disabled, and zero otherwise. *C3* states that taxpayers who have an address in Luxembourg are considered residents. *C4* states that taxpayers with a local income but no local address are considered non-residents. *C5* states that a tax card applies only to employment and pension incomes.

The constraint solving solution we propose in this article is aimed at generating *valid* instantiations of CDs, meaning that the generated instance models should satisfy both the diagrammatic and OCL constraints.

3 BACKGROUND

In this section, we provide background on OCL, search-based OCL solving, and SMT.

OCL. OCL is the de-facto constraint and query language for UML [55]. One can use OCL for defining: (1) invariants (conditions that must hold over all instances of a given class), e.g., *C1* to *C5* in Fig. 1(b), (2) (user-defined) operations, e.g., our running example in Fig. 1(a) includes only one user-defined operation, named *getAge*, whereas the remaining operations highlighted in blue in

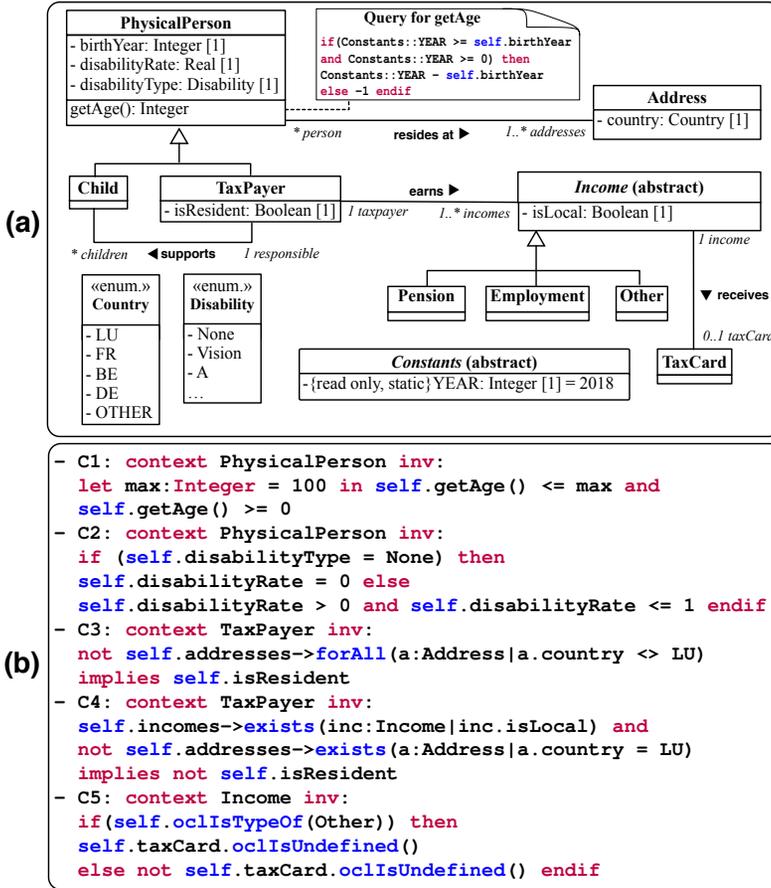


Fig. 1. (a) Example UML Class Diagram, and (b) Example OCL Constraints

Fig. 1(b) are part of OCL, (3) derived attributes (attributes whose values are calculated from other elements), (4) guard conditions for transitions in behavioral diagrams such as UML State Machines, and (5) pre- and post-conditions for user-defined operations.

We concentrate on handling *OCL invariants* and *the user-defined operations* needed by them. In our context, user-defined operations provide a convenient mechanism for factoring out queries which are used multiple times, or which are too complex to spell out within the invariants. Derived attributes are treated as user-defined operations. For example, the *getAge* operation in Fig. 1(b) would be interchangeable with a derived attribute that calculates the age of an individual (not shown).

Guard conditions fall outside the scope of our work, since we deal exclusively with CDs and their instantiations. Pre- and post-conditions are mainly an apparatus for verifying the executability and determinism of system behaviors [17]. Although uncommon, pre- and post-conditions can also be used for constraining the instantiations of CDs. Despite such use of pre- and post-conditions being plausible, we leave pre- and post-conditions out of the scope of our work, and assume that all the relevant data restrictions are provided as invariants.

OCL constraint solving via (metaheuristic) search. Search has been employed for addressing a wide range of software engineering problems [36]. Related to our objectives in this article, Ali

et al. [2, 3] develop a search-based method for solving OCL constraints. In our previous work, we enhanced this method with new heuristics aimed at improving the exploration of the search space [77]. The work we present in this article builds on our enhanced implementation of search-based OCL solving. This enhanced implementation is a contribution of our previous work [76] and not that of this article. Nevertheless, since the implementation is an intrinsic part of our approach, we outline its main technical characteristics below.

Our search-based OCL solver is based on the Alternating Variable Method (AVM) [46, 53]. This choice is motivated by the empirical findings of Ali et al. [2], indicating that, for OCL constraint solving, AVM outperforms popular metaheuristic techniques such as (1+1) evolutionary [29] and genetic [38] algorithms. AVM works in a similar manner to the well-known Hill Climbing algorithm [69]. In particular, it starts with an arbitrary solution; it then attempts to find a better solution by making incremental changes, and keeps evolving the solution until no further improvements can be found.

Concretely speaking and in our context, AVM starts with a random and potentially invalid instance model, I , which the algorithm later evolves so as to get it to satisfy the OCL constraints at hand. Specifically, we use AVM to optimize – based on a fitness function that we describe momentarily – a vector of variables $V = (v_1, \dots, v_n)$, where n is the number of features in I that should be evolved by search. For solving OCL constraints, the features are the objects (class instances), the association ends, and the attributes. For example, the *disabilityRate* attribute of a specific instance of the *PhysicalPerson* class would constitute a variable within V . We note that since AVM evolves the structure of I , e.g., by adding new objects, the length of V is not fixed and may vary as the AVM algorithm unfolds. For example, if I includes no instance from the *PhysicalPerson* class, then there would be no variables representing the attributes and association ends of *PhysicalPerson* instances. However, if I does include some instance of *PhysicalPerson*, each attribute and each association end of that instance would give rise to one variable within V .

Given a non-empty V , AVM optimizes the variables in V as follows. It picks in a sequential manner a single variable v_i from V . The algorithm then subjects v_i to a search process called Iterated Pattern Search (IPS) [46]. For a given v_i , IPS first makes small exploratory moves over v_i . For example, if v_i is an integer, a small exploratory move could be incrementing or decrementing the value of v_i by one unit. These exploratory moves are aimed at guessing what “direction” is more promising to explore, when the notion of direction is applicable. To illustrate, let us assume that v_i is an integer with a current value of 20. IPS’s first exploratory move could be to set v_i to 21. If this increase leads to no improvement in the value of the fitness function, then IPS applies another exploratory move, whereby v_i is decremented, e.g., to 19. Similar exploratory moves are defined for non-nominal variable types other than integers. For example, variables representing sets of objects are explored by adding or deleting an object. If no exploratory move leads to an improvement in the value of the fitness function, then no “direction” is worth further exploration. In this case, IPS terminates and AVM moves on to the next variable in V .

If one of the exploratory moves leads to an improvement in the fitness value, then a positive (increment) or negative (decrement) “direction” is established for making further moves. Based on the selected direction, moves with larger amplitudes will be made. For example, in the case of integers, increases or decreases by more than one unit will be applied. Moves in a certain direction will continue as long as the value of the fitness function keeps improving. When a given move deteriorates the fitness function value, the search has likely overshoot an optimum. When this occurs, IPS goes back to the exploratory phase in order to re-establish a new direction for further optimizing the value of the variable in question.

For nominal variable types, e.g., enumerations, where the notion of direction does not apply, there is no IPS. In such cases, the variable in question is optimized as follows: AVM picks either (1) all the

alternatives, when the space of all alternatives is small, in our case, containing ≤ 10 alternatives, or (2) a random selection of the alternatives (in our case, 10 alternatives), otherwise. The alternative leading to the best fitness value is retained, and AVM moves on to the next variable in V .

Once all the variables in V have been treated, AVM is said to have completed a “search iteration.” After the completion of a search iteration, AVM goes back to the first variable in V . AVM continues in this manner until I becomes valid or a maximum user-specified number of search iterations has been reached. To reduce the chances of entrapment in non-solution regions of the search space, AVM relies on a simple heuristic: If a given search iteration yields no improvement at all in the fitness function value and I is still invalid, then AVM abandons I and continues with a fresh randomly created instance model I' .

Our implementation of search-based OCL solving employs the fitness function defined by Ali et al. [2]. Essentially, this function is a quantitative assessment of how far a given instance model is from satisfying a given set of OCL constraints. For details, we refer the reader to Ali et al. To facilitate comprehension, we briefly illustrate the fitness function using a simple example. Suppose that the set of constraints to satisfy includes only the following constraint, C : `TaxPayer.allInstances()->forAll(x | x.incomes->size())>=1 and x.incomes->size()<4`. Further, suppose that the instance model being evolved has two instances of the *TaxPayer* class, one with a single income and the other with four incomes. The computed distance for the first taxpayer is 0, since that taxpayer already satisfies C ; whereas the distance for the second taxpayer is 1, since this second taxpayer has one more income than what C allows. The distance for `forAll` and thus for C as a whole is computed as the average of the distances for all (here, the two) taxpayers: $(0 + 1)/2 = 0.5$. Now, let C' be the same as C , but with C 's `forAll` operation replaced with `exists`. For the same instance model, the fitness function for C' would be computed as $\min(0, 1) = 0$. An overall distance of 0 indicates that a solution has been found; otherwise, AVM will iteratively attempt to minimize the (overall) distance, as described above.

SMT. SMT is the problem of determining whether a constraint expressed in first-order logic is satisfiable, when certain symbols have specific interpretations in a combination of background theories [25]. State-of-the-art SMT solvers, e.g., Z3 [24] and CVC4 [8], accept as input a standard language, called SMT-LIB [9]. For example, $C2$ of Fig. 1(b) can be expressed in SMT-LIB as follows: `(ite (= x None) (= y 0) (and (> y 0) (<= y 1)))`, where x and y respectively represent the disability type and the disability rate for a given person.

Most SMT solvers employ a SAT solver alongside the decision procedures for the background theories they support [45]. The SAT solver operates on Boolean formulas, and the decision procedures operate on the symbols within the formulas. SMT solvers are highly efficient for solving quantifier-free constraints, when suitable background theories exist. Nevertheless, when constructs such as quantification, collections, or symbols with no background theories are added to the mix, these solvers often become inefficient.

Since SMT solvers alone are not able to efficiently handle all types of OCL constraints that are commonly used for specifying software systems, we propose to exploit SMT only for solving OCL subformulas where SMT is very likely to deliver efficiency gains.

4 APPROACH

Fig. 2 presents an overview of our approach, hereafter referred to as PLEDGE (Practical and Efficient Data Generator for UML), i.e., the name of the tool implementing it as mentioned in the introduction. PLEDGE takes as input a CD and a set of OCL constraints. The output is a valid instance model if one can be found.

PLEDGE works in two phases: First, it constructs a single OCL constraint in Negation Normal Form (NNF) that is logically equivalent to the conjunction of the constraints that need to be solved.

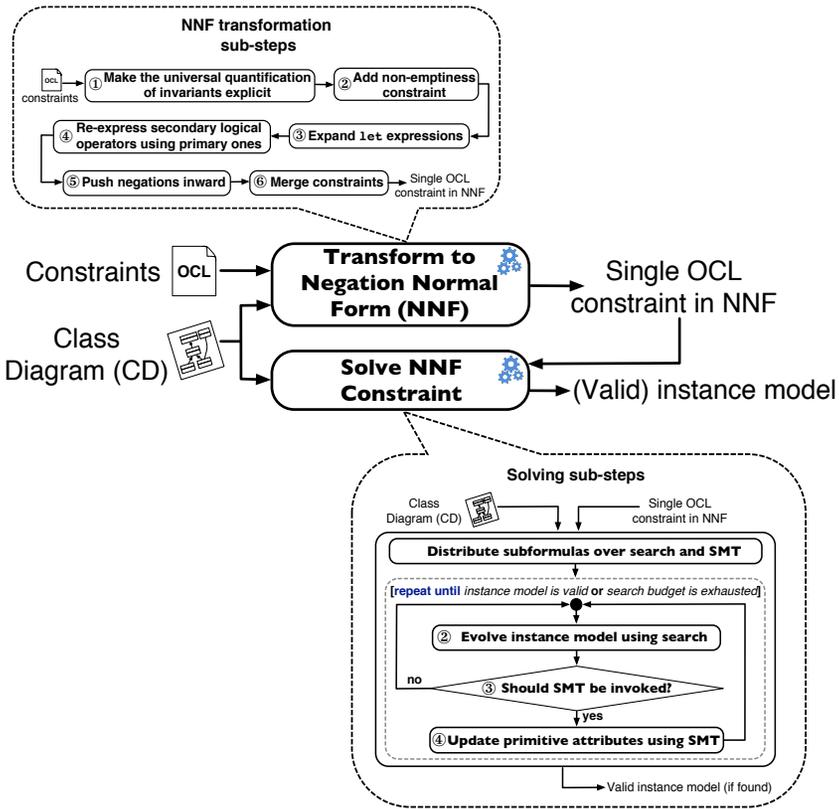


Fig. 2. Approach Overview

A constraint is in NNF if it uses only the primary Boolean operators, i.e., conjunction, disjunction and negation, and further has all the negations pushed inward to the atomic level [65]. The sub-steps underlying the NNF transformation are further discussed in Section 5. We use NNF because it facilitates defining and distributing the solving tasks over search and SMT, while also simplifying the translation from OCL to SMT-LIB. Specifically, the single constraint is a conjunction of the following: (1) the input OCL constraints, and (2) OCL representations of the multiplicity constraints in the input CD. For example, the $1..*$ cardinality attached to the target end of the *earns* association in Fig. 1(a) is represented in OCL as: `self.incomes->size() >= 1`, where `self` refers to any instance of *TaxPayer*. We leave out of this conjunction the basic constraints imposed by the CD’s diagrammatic notation, e.g., type conformance for the association ends and the non-instantiability of abstract classes, as illustrated in Section 2. PLEDGE implicitly enforces these basic constraints when creating or tweaking instance models.

The second phase of PLEDGE is to solve the resulting NNF constraint. To do so, we utilize a combination of search and SMT, with the solving tasks distributed in the following manner: We have search handle subformulas whose satisfaction involves structural tweaks to the instance model, i.e., additions and deletions of objects and links. An example constraint (subformula) handled by search is *C5* of Fig. 1(b). If the instance model happens to violate *C5*, any successful attempt at satisfying the constraint will necessarily involve adding / removing objects and links.

We have SMT handle subformulas which exclusively constrain attributes with primitive types. Many such subformulas, e.g., those whose symbols are within linear arithmetic, can be efficiently handled by background SMT theories. For example, we use SMT to assign a value to the *birthYear* attribute of an instance of *PhysicalPerson* in such a way that *C1* of Fig. 1(b) is satisfied. Finally, we have *both* search and SMT handle subformulas whose satisfaction may require a combination of structural tweaks and value assignments to primitive attributes. For example, satisfying *C3* of Fig. 1(b) may involve both adding instances of *Address* and setting the *country* and *isResident* attributes of *Address* and *TaxPayer* instances.

As mentioned in the introduction, the core idea behind PLEDGE is to enhance the performance of search-based OCL constraint solving by combining it with SMT. Specifically, PLEDGE aims to support search with SMT solving capabilities for the subformulas where efficient and effective decision procedures for certain background theories exist. The decision procedures that PLEDGE taps into are those related to theories for basic Boolean operators, and arithmetic over integer, real, and floating-point numbers [10, 27]. This is ensured by applying SMT over only the subformulas from the NNF constraint that concern attributes with primitive types. All remaining subformulas, which are mostly concerned with constraints on objects and links, are delegated to search. As illustrated earlier, this type of subformulas requires evolving the structure of the instance model to be satisfied. Further, these subformulas often involve, among other things, quantifications, collection operations, and type operations, which are not currently (well) supported by SMT (see Section 3).

Nevertheless, one should *not* view the distribution of the solving tasks as a clear-cut division of the NNF constraint into smaller constraints that can be later solved independently. For example, and as illustrated earlier, SMT and search might have overlapping solving tasks where the involved subformulas require a combination of structural tweaks and value assignments to primitive attributes. Further, in the PLEDGE approach, SMT only operates over the primitive attributes of the objects already created via search (fixed universe). For example, satisfying *C1* and *C2* of Fig. 1(b), which are fully assigned to SMT, is only warranted for the objects of type *PhysicalPerson* that were created via search, e.g., created to satisfy multiplicity or other user constraints. As a result, *C1* and *C2* cannot be solved independently by SMT regardless of how search evolves the instance model. Rather and as discussed in Section 6, search and SMT modify a shared instance model in order to satisfy the NNF constraint, with each technology focusing on the subformulas delegated to it. We further discuss in more detail how we delegate the solving tasks in Section 6.1.

We note that the generated data is stored as a UML instance model and thus can be used directly for model-based testing in a UML context [47, 73]. If a different test input format is required, say, text files, databases or a combination thereof, the generated instance model needs to be adapted. For simple CDs that do not involve inheritance or association classes, the objects and links in an instance model can be converted relatively easily into other representations. In particular, one can employ model-to-text or model-to-model transformation using tools such as Acceleo [30], ATL [31] and Xpand [32] for conversion into the desired test-input format. In an object-oriented context and when CDs have inheritance or association classes, it is more practical to use an object-relational mapping (ORM) tool for adapting the test input [80]. To this end, one may use tools such as Hibernate [64] or Apache OpenJPA [7]. In the remainder of the article, we take an instance model to be equivalent with test data.

We further note that our examples above refer directly to the original constraints in Fig. 1, rather than to the corresponding fragments in the derived NNF constraint. This is only to ease illustration; the whole solving process, including making decisions about which subformulas to delegate to search, SMT or both, is with respect to the NNF constraint resulting from the first phase of PLEDGE. All the technical subs-steps underlying the solving phase are discussed in depth in Section 6.

5 TRANSFORMATION TO NEGATION NORMAL FORM

In this section, we elaborate the first phase of the PLEDGE approach outlined in Section 4. The transformation to NNF is performed using the pipeline of Fig. 3.

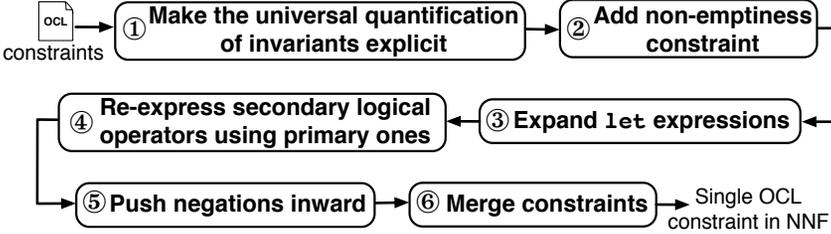


Fig. 3. Pipeline for Deriving the NNF Constraint to Solve

As noted in Section 3, our focus is on constraints expressing OCL invariants. Invariants are implicitly universally quantified. In Step 1 of the pipeline of Fig. 3, we make this implicit quantification explicit. Doing so enables us to later merge different constraints (Step 6), even when the constraints do not share the same context.

In Fig. 4, we show the result of applying Step 1 to $C1$ and $C3$ of Fig. 1(b). For efficiency reasons, Step 1 binds only one universal quantifier to a given OCL context. This is achieved by taking the conjunction of all the invariants that have the same context, before universal quantification is made explicit. For example, $C1$ and $C2$ of Fig. 1(b) are both defined over *PhysicalPerson*. Step 1 would thus consider $C1$ and $C2$ as one constraint. Due to space, we do not illustrate this treatment in Fig. 4.

Step 2 adds a user-defined non-emptiness constraint to the set of constraints resulting from Step 1. The non-emptiness constraint excludes the empty solution and thus avoids vacuous truth, noting that the constraints from Step 1 are invariants and thus always satisfied by the empty instance model. The non-emptiness constraint is, in contrast, not universally quantified. In the simplest case, this constraint states that a certain class of the input CD must have an instance. A natural non-emptiness constraint for our running example would be `TaxPayer.allInstances()->size() >= 1`, stating that the instance model must contain at least one instance of *TaxPayer*. In practice, the non-emptiness constraint can be more elaborate, e.g., bounding the maximum number of instances to generate from each class. Regardless of its complexity, the non-emptiness constraint must always be violated by the empty instance model and its satisfaction has to involve the creation of certain objects within the instance model.

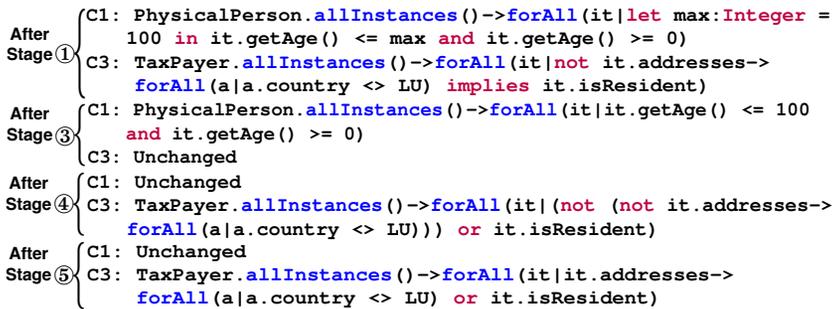


Fig. 4. Illustration of NNF Transformation over $C1$ and $C3$ of Fig. 1(b)

Step 3 of the pipeline performs an inline expansion of any **let** expressions within the constraints. These expressions are commonly used for avoiding repetition and improving readability. For our OCL solving process to be sound, we need to expand all **let** expressions, including nested ones. Fig. 4 illustrates how the **let** expression in *C1* is expanded inline.

Step 4 re-expresses OCL's secondary Boolean operators, i.e., **implies**, **xor** and **if-then-else**, in terms of the primary ones, i.e., **and**, **or** and **not**. Step 5 pushes the negation operators inward by applying De Morgan's law [39]. Both Steps 4 and 5 are illustrated in Fig. 4 over *C3*.

Steps 4 and 5 are the core steps for transforming the initial OCL constraints to NNF. These steps always terminate. In particular, Step 4 terminates when the OCL constraints become free of OCL's secondary Boolean operators; Step 5 terminates when no negation operator can be pushed further inward. The logical rules underlying these steps are equivalence preserving [39], meaning that the rules transform any constraint *C* into an NNF constraint *C'* such as *C* and *C'* share the same truth table. In terms of scalability, the main risk associated with the NNF transformation is that Step 4 can lead to an exponential blowup in the size of the NNF constraint. Despite such a blowup being theoretically possible, the situation is unlikely to pose issues in practice. This is because the transformation is applied to OCL constraints that have been either manually written by analysts or derived from CDs, as explained in Section 4. In the former case, individual constraints would typically include at most only a handful of secondary Boolean operators, considering that a large number of such operators would considerably increase cognitive load and decrease understandability. In the latter case, the derived constraints are free of secondary Boolean operators and thus not affected by Step 4. In all of our case studies reported in Section 7, the NNF transformation took negligible time (in the order of milliseconds).

Finally, Step 6 takes the conjunction of the constraints obtained from Step 5. The resulting NNF constraint for our running example is shown in Fig. 5. For easier reference, we divide this constraint into five subformulas, labeled S1 to S5.

We need to point out two subtleties about the pipeline of Fig. 3: First, Steps 1 and 3 have to be done in such a way that no name clashes arise. To avoid name clashes, we give all variables distinct (automatically generated) names. Since no name clashes occur in our running example, we elected, for readability reasons, not to rename the variables in Figs. 4 and 5. Second, Steps 3 through 5 of the pipeline further apply to the user-defined operations needed by the constraints. In our example,

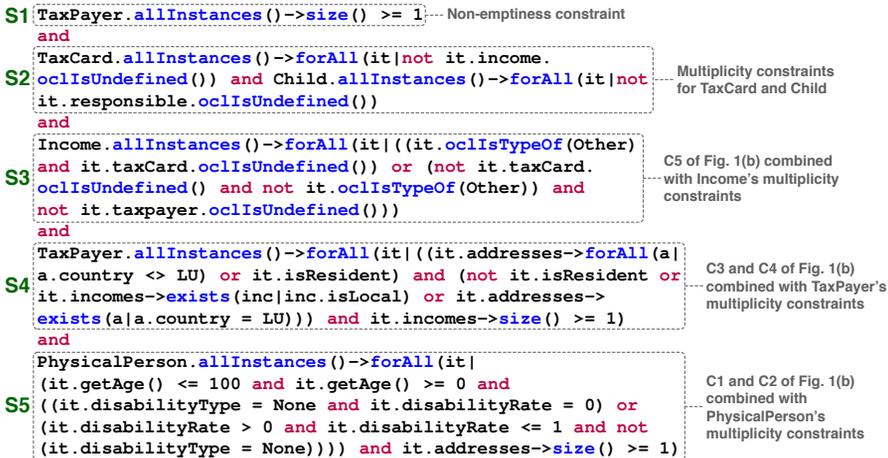


Fig. 5. Example NNF Constraint Produced by the Pipeline of Fig. 3

the only (user-defined) operation needed, *getAge*, remains unaltered since it has no **let** expressions, secondary logical operators, or negations.

The NNF constraint generated by the pipeline of Fig. 3 is solved through the process that we present next.

6 HYBRID OCL SOLVING USING SEARCH AND SMT

In this section, we elaborate our hybrid OCL solving process, i.e., the second phase of the PLEDGE approach in Section 4. The solving process, depicted in Fig. 6, has four steps. The first step is performed once; the remaining three steps are iterative. These three steps are repeated until either an instance model satisfying the NNF constraint is found, or the maximum number of search iterations is reached. We discuss each step of the solving process below.

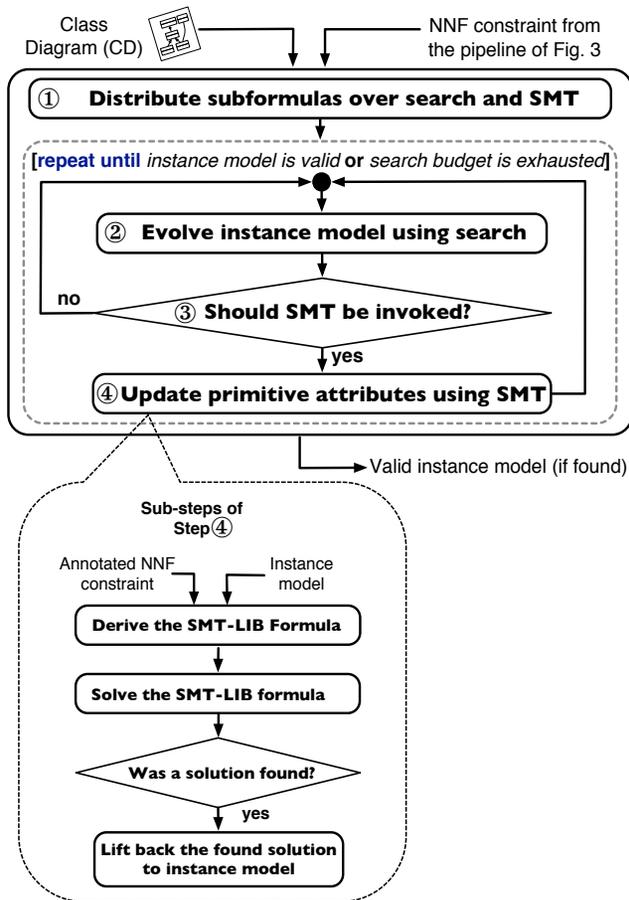


Fig. 6. Hybrid OCL Solving Process

6.1 Delegating Solving Tasks to Search & SMT (Step 1, Fig. 6)

Step 1 of the process of Fig. 6 decides how to delegate to search and SMT the solving of different subformulas of the NNF constraint derived in Section 5. Specifically, Step 1 applies a static and

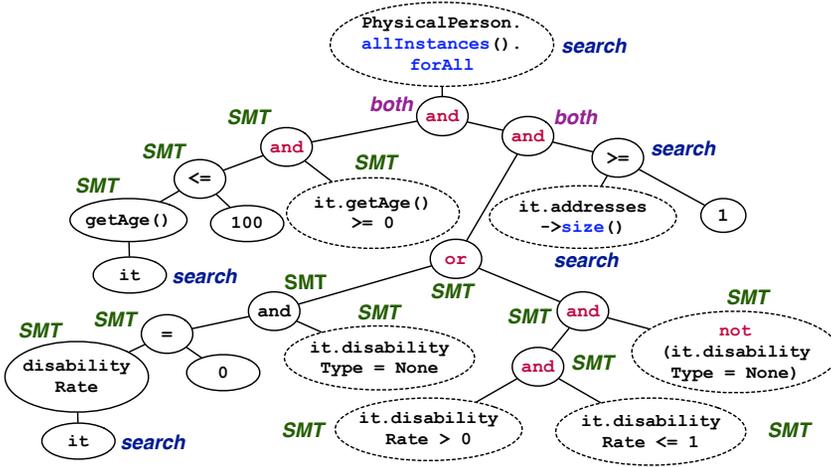


Fig. 7. Abstract Syntax Tree for S5 of the Constraint in Fig. 5

deterministic procedure, whereby the nodes in the Abstract Syntax Tree (AST) of the derived NNF constraint are given one of the following labels: (1) *search*, (2) *SMT*, or (3) *both*.

These labels operationalize our strategy, described in Section 4, for combining search and SMT. Before presenting our labeling procedure in detail, we intuitively describe how it works: The procedure attaches labels to the AST nodes of an NNF constraint based on each node’s type, properties, and position within the AST. For example, S5 in Fig. 5 has two subformulas of the form $x \geq (\text{constant})$, where x is a variable. Although the AST root nodes for both subformulas are the \geq operation, these root nodes are labeled differently. Specifically, the root node of the first subformula, `it.getAge() >= 0`, is labeled *SMT*; this is because the inequality here is indirectly defined over the *birthYear* primitive attribute in Fig. 1(a). In contrast, the root node of the second subformula, `it.addresses->size() >= 1`, is labeled *search*, because the inequality here restricts the minimum number of allowed instantiations of the *resides at* association in Fig. 1(a). We recall from Section 4 that we want to have search handle all subformulas that are exclusively concerned with the structure of the instance model. For the sake of argument, let us hypothetically assume that we have a third subformula, which is the conjunction of the two subformulas mentioned above. In such a case, the AST root node representing the conjunction would be labeled *both*. This because solving this (hypothetical) subformula would involve both search and SMT.

More precisely and technically speaking, the labeling procedure of Step 1 is realized by a depth-first traversal of the AST of a given NNF constraint, with the visited nodes labeled via a set of predefined rules. The rules are heuristics that we have designed to assign, in a deterministic manner, a single label to each visited AST node, except those representing constants (e.g., 100 in Fig. 7, discussed later); constants do not play a role in deciding about the delegation of solving tasks. Our rules cover all the AST node types defined by the OCL 2.4 metamodel [55]. Since the rules are numerous, we do not list them here; our complete rule set and the rationale behind each rule can be found in Appendix A. Below, we provide a general description of the rules and illustrate them over S5 of the NNF constraint in Fig. 5.

The AST for S5 is shown in Fig. 7. To avoid clutter, we have collapsed some nodes. The collapsed nodes, marked by dashed borders, expand similarly to the node with the “=” symbol (bottom left of the figure). The label shown for each collapsed node is that of its root. Our rules can be broadly classified into three categories. The rules in the first category label an AST node based only

on information contained within the node. Notably, (i) nodes representing quantifiers or object references are labeled *search*, e.g., the root node of the AST of Fig. 7; and (ii) nodes representing calls to user-defined operations are labeled *SMT* if the operations are not recursive and have primitive return types (e.g., the `getAge` nodes in Fig. 7); otherwise, these nodes are labeled *search*.

The rules in the second category infer a label for an AST node based on the type of the node itself and those of its ancestors. Notably, a node representing a primitive attribute is labeled *SMT*, unless the node has an ancestor of a certain type, e.g., `exists`, in which case, the attribute is labeled *both*. In Fig. 7, all primitive attributes are labeled *SMT*. To provide an example of a primitive attribute that is labeled *both*, consider `S4` in the constraint of Fig. 5. In the AST for `S4` (not shown), the node representing `isLocal` would be labeled *both*.

The rules in the third category infer a label for an AST node based on the node's children. Notable constructs handled by this category of rules are logical and numerical operators. For example, the `and`, `or`, `not`, `=`, `>=`, `<=`, `>` nodes in Fig. 7 are all labeled based on their (immediate) children. Specifically, if all children have the same label, that label is propagated to the parent node. If the children have different labels, then the parent node is labeled *both*. AST nodes representing constants are left unlabeled; these nodes do not play a role in deciding about the delegation of solving tasks.

Although not illustrated in Fig. 7, when a node representing a call to a user-defined operation is labeled *SMT*, the AST of the operation is subject to the same labeling procedure discussed above. This treatment does not apply when these nodes are labeled *search*; in such cases, all the nodes within the AST of the referenced operation are labeled *search*. This is because, as we elaborate in Section 6.4, the PLEDGE approach hides from *SMT* any user-defined operation called from a node that is labeled *search*. Consequently, we need to handle any such operation call via *search* only.

Noting that OCL has its roots in first-order logic, we believe that our delegation rules are likely to be adaptable to other formal specification languages based on first-order logic. At the same time, we need to stress that our current rules have been defined directly over the OCL metamodel. Assessing whether and how these rules would generalize beyond OCL requires additional investigations, which we leave for future work.

6.2 The Search Step (Step 2, Fig. 6)

Step 2 runs one iteration of search. This step utilizes the search-based OCL solver we developed previously [77] and outlined in Section 3. We refer to this solver as the *baseline* hereafter. We modify the baseline by limiting what search is allowed to do. Precisely, we allow search to manipulate only element types that are referenced by AST nodes labeled *search* or *both*. For example, if we were to satisfy `S5` of the constraint in Fig. 5, search would be allowed to add / remove instances of only the following types: the *PhysicalPerson* and *Address* classes and the *resides at* association. Suppose that search creates an instance *P* of *PhysicalPerson*. In doing so, *P*'s primitive attributes necessarily receive initial values (e.g., default or random values). However, once *P* is added to the instance model, search is prohibited from tweaking *P*'s primitive attributes. This is because the primitive attributes of *PhysicalPerson* are referenced only within nodes that are labeled *SMT* in the AST of Fig. 7.

6.3 Avoiding Futile SMT Invocations (Step 3, Fig. 6)

Step 3 is aimed at avoiding *SMT* invocations that, given the current structure of the instance model, have no chance of solving the NNF constraint. We recall from Section 4 that the only means we make available to *SMT* for altering an instance model is by manipulating primitive attribute values. If a violation is unfixable this way, invoking *SMT* will be futile. For example, if the current instance model violates some multiplicity constraint, the violation cannot be fixed by *SMT* in our proposed solving process.

To decide whether to invoke SMT, we do as follows: First, we clone the (labeled) AST of the underlying NNF constraint. Next, in this (cloned) AST, we replace with `true` any node labeled `SMT` and representing a Boolean expression. Naturally, the branches below the replaced nodes are pruned. In Fig. 8, we show the result of this treatment applied to the AST of Fig. 7. The rationale behind the treatment is simple: We assume that SMT, *if* invoked, will be able to solve all subformulas delegated to it. If the constraint represented by the reduced AST evaluates to `false` on the current instance model, SMT cannot conceivably satisfy the whole constraint. Otherwise, we give SMT a chance to solve the subformulas delegated to it.

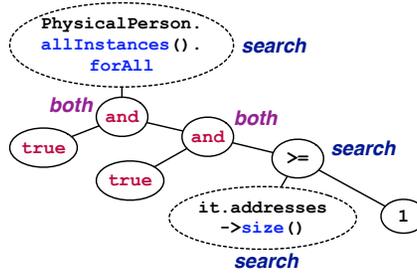


Fig. 8. Reduced AST obtained from the AST in Fig. 7

We make two remarks about Step 3. First, for the step to be sound, the original AST must represent an OCL expression in NNF, where negations have been pushed all the way inward, as discussed in Section 5. Second, the construction of the reduced AST is a one-off activity. It is only the evaluation of this AST that needs to be repeated in each iteration of Step 3.

6.4 The SMT Step (Step 4, Fig. 6)

For a given instance model, Step 4 does the following: It first encodes into an SMT-LIB formula the solving task that needs to be handled by SMT. Next, it invokes an SMT solver on the resulting formula. The formula is solved if the SMT solver succeeds in finding a satisfying assignment to the SMT-LIB variables, where each variable corresponds to a specific primitive attribute in the instance model. If the formula is solved, the solution will be “lifted back” to the instance model. By lifting back, we mean that the value of each SMT-LIB variable in the solution is assigned to its corresponding primitive attribute in the instance model. For example, suppose that an instance model passes the check in Step 3 (Section 6.3) but still violates S5 of the NNF constraint in Fig. 5. This means that the attributes in some instances of *PhysicalPerson* (individuals) have to be re-assigned so that the instance model would satisfy S5. In this case, S5 is violated because the instance model includes individuals who: (1) have an age outside the range [0..100], thus violating *C1* in Fig. 1(b), (2) are disabled but have a disability rate outside the range (0..1], thus violating *C2* in Fig. 1(b), or (3) are not disabled but have a non-zero disability rate, thus also violating *C2* in Fig. 1(b). Here, the SMT solver would be tasked with finding appropriate values for the *birthYear*, *disabilityType* and *disabilityRate* attributes of the *PhysicalPerson* instances already in the instance model such that *C1* and *C2* in Fig. 1(b) would hold.

6.4.1 Deriving the SMT-LIB Formula. Alg. 1, named **deriveFormula**, presents our algorithm for constructing an SMT-LIB formula. The algorithm takes the following as input: (1) the AST of an NNF constraint already processed by the labeling procedure of Section 6.1, and (2) an instance model passing the check discussed in Section 6.3. Initially, **deriveFormula** expands the primitive-attributes-carrying quantifiers of the NNF constraint over the instance model (L. 2). The expansion

Alg. 1: Derive SMT-LIB Formula from NNF Constraint (*deriveFormula*)

Inputs : (1) The (labeled) AST, T , of an NNF constraint; (2) an instance model, $inst$.

Output : An SMT-LIB formula over the primitive attributes of $inst$ referenced in T .

Fun. calls: **expand**: expands over an instance model the primitive-attribute-carrying quantifiers of an OCL expression represented by an AST (Alg. 2); **substitute**: substitutes all subformulas delegated exclusively to search by their concrete evaluation over an instance model (Alg. 3); **processOperations**: performs special processing over the ASTs of user-defined operations so that these operations can later be translated to SMT-LIB (Alg. 4); **fromOCL2SMT-LIB**: generates an SMT-LIB formula for a (processed) NNF constraint (Alg. 5).

```

1 Let  $root$  be the root node of  $T$ 
2  $root_{expanded} \leftarrow \mathbf{expand}(root, inst)$ 
3  $root_{post-substitution} \leftarrow \mathbf{substitute}(root_{expanded}, inst)$ 
4 Let  $\mathcal{U}$  be the set of the ASTs of all the user-defined operations referenced
   (directly or indirectly) within the constraint represented by  $T$ 
5  $\mathcal{U}_{processed} \leftarrow \mathbf{processOperations}(\mathcal{U}, root_{post-substitution}, inst)$ 
6 return  $\mathbf{OCL2SMT-LIB}(root_{post-substitution}, \mathcal{U}_{processed}, inst)$ 

```

hides quantification, thus allowing SMT to be invoked on a quantifier-free formula – this is where, as we noted in Section 3, existing SMT solvers are most efficient. After expansion, **deriveFormula** substitutes any subformula that is exclusive to search with the concrete evaluation of that subformula over the instance model (L. 3). The substitution hides from SMT those subformulas within the expansion whose evaluation will not be affected by SMT. Subsequently, the algorithm applies expansion and substitution to user-defined operations, taking into account the peculiarities posed by these operations (L. 4-5). Finally, **deriveFormula** translates the now processed NNF constraint and user-defined operations into an SMT-LIB formula (L. 6).

In the remainder of this section, we elaborate the four algorithms used by Alg. 1 for (1) expansion, (2) substitution, (3) user-operation handling, and (4) SMT-LIB translation, respectively.

(1) Expanding the NNF constraint: Alg. 2, named **expand**, re-expresses quantifiers in terms of their quantified objects. Expansion is warranted only for nodes whose body contains primitive attributes (L. 1-3). This is because SMT, the way we use it in the PLEDGE approach, cannot affect the evaluation of subformulas that have no primitive attributes. For example, consider S2 and S3 in the NNF constraint of Fig. 5. The quantifiers in these subformulas do not need to be expanded. A universal quantifier (**forAll**) is expanded by creating a copy of its body for each quantified object (L. 1-11), and then taking the conjunction of the copies (L. 12-13). As an example, Fig. 9(b) shows S5 of the NNF constraint in Fig. 5 expanded over the instance model of Fig. 9(a). An existential quantifier (**exists**) is expanded similarly, but with a disjunction applied to the copies (L. 14-15). If the collection of quantified objects is empty, the quantifier in question is replaced with **true** in case of **forAll** (L. 19-20), and with **false** in case of **exists** (L. 21).

In addition to the explicit universal and existential quantifiers, expansion applies to certain other (built-in) OCL operations which, logically speaking, are shortcuts for expressions with quantification (L. 24-26 in Alg. 2). To illustrate, the operation $c \rightarrow \mathbf{includes}(e)$, where c is a collection and e is an object, is equivalent to: $c \rightarrow \mathbf{exists}(i \mid i = e)$. We expand such shortcuts by making quantification explicit (L. 25-26 in Alg. 2). In Appendix B, we provide a complete list of these

Alg. 2: Expand Quantifiers Involving Primitive Attributes (expand)

Inputs : (1) An AST node, *original*, from the labeled AST of an (NNF) OCL expression; (2) An instance model, *inst*.

Output : An AST (root node) resulting from the expansion over *inst* of the primitive-attribute-carrying quantifiers of *original*.

Fun. calls: **rename**: replaces within an AST all occurrences of an iterator name by a new name.

```

1 if (original is a quantifier node) then
2   Let expr be the root AST node for the body of original
3   if (expr contains some primitive-attribute node) then
4     Let S be the collection of objects from inst quantified by original
5     if (S ≠ ∅) then
6       C ← ∅ /* C will store one copy of expr per object in S */
7       Let itrName be the name of the iterator variable of original
8       foreach (s ∈ S) do
9         Let objId be the unique identifier for s
10        Let exprcopy be a (cloned) copy of expr
11        C ← C ∪ {rename(exprcopy, itrName, objId)}
12      if (original is a forall) then
13        expansion ←  $\bigwedge_{j=1}^{j=|C|} C_j$  /* Take the conjunction of the expressions in C */
14      else
15        expansion ←  $\bigvee_{j=1}^{j=|C|} C_j$  /* Take the disjunction of the expressions in C */
16      Let root be the root node of expansion's AST
17      return expand(root, inst)
18    else
19      if (original is a forall) then
20        return an AST node representing true
21      else return an AST node representing false
22  else return original
23 else
24  if (original contains some primitive-attribute node and is further of one of the
25  following types: an equality/inequality operator over collections, excludes,
26  excludesAll, includes, includesAll, isUnique, or one) then
27    /* original is an OCL operation with implicit quantification over primitive attributes */
28    Reformulate original in terms of explicit quantifiers (see Appendix B), and let
29    root' be the root node of the AST of the reformulation
30    return expand(root', inst)
31  else
32    /* original involves neither explicit nor implicit quantification. expand is called recursively over
33    original's children */
34    Let N be the set of the direct child nodes of original
35    foreach (n ∈ N) do
36      n' ← expand(n, inst)
37    Replace n with n' in the children of original
38    return original

```

shortcuts alongside their equivalent expressions with explicit quantification. In the final segment of the algorithm (L. 28-31), expansion is applied recursively to the children of a visited node.

(2) Substituting subformulas that are exclusive to search: After expansion, Alg. 3, named **substitute**, hides from SMT the subformulas that are exclusive to search. Specifically, Alg. 3 substitutes by its concrete evaluation over the instance model any subformula whose root AST node

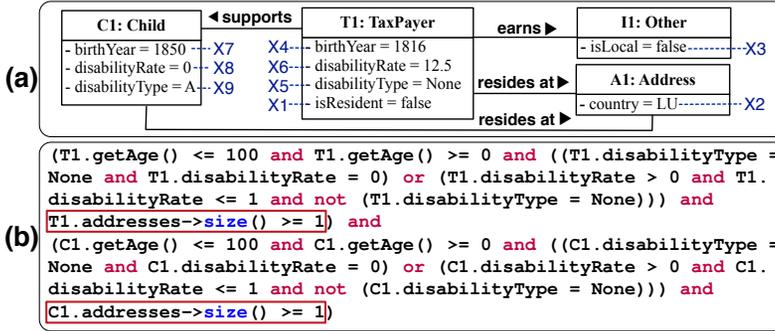


Fig. 9. (a) Example Instance Model, and (b) Expansion of S5 of the NNF Constraint in Fig. 5 over the Example Instance Model

has been labeled *search* by the procedure of Section 6.1 (L. 1-3). For example, consider the already expanded formula in Fig. 9(b). The AST nodes representing the two subformulas delimited by boxes have their root AST node labeled *search* (not shown in the figure). The **substitute** algorithm replaces these two subformulas with **true**, noting that *T1* and *C1* in Fig 9(a) are each associated to one instance of *Address*. Similarly to the **expand** algorithm (Alg. 2) discussed earlier, the final segment of the **substitute** algorithm (L. 5-8) recursively applies substitution to the children of the visited node.

Going back to the NNF constraint of Fig. 5, the expansion and substitution processes would unwind over the NNF constraint of Fig. 5 as follows: S1 to S3 are substituted by their concrete evaluation over the instance model without being expanded. The primitive-attribute-carrying quantifiers in S4 and S5 are first expanded over the instance model of Fig 9(a) in the manner illustrated for S5 in Fig 9(b). Subformulas whose root AST nodes are labeled *search* are then substituted by their concrete evaluation over the same instance model.

(3) Handling user-defined operations: User-defined operations may too contain quantification as well as subformulas that should be hidden from SMT. Since user-defined operations are expressed in OCL, we process them in the same way as the NNF constraint, i.e., by invoking **expand** and **substitute** (Algs. 2 and 3). Nevertheless, before expansion and substitution can be applied to user-defined operations, some pre-processing is required. Alg. 4, named **processOperations** and discussed next, tailors expansion and substitution to user-defined operations.

Alg. 3: Substitute Subformulas Delegated Exclusively to Search (substitute**)**

Inputs : (1) An AST node, *original*, from the labeled and expanded AST of an OCL expression; (2) An instance model, *inst*.

Output : An AST (root node) resulting from the substitution by their concrete evaluation over *inst* of all subformulas that need to be hidden from SMT.

```

1 if (original is labeled search) then
2   | Let eval be the result of evaluating over inst the expression represented by original
3   | return the root node of the AST for eval
4 else
5   | Let N be the set of the direct child nodes of original
6   | foreach (n ∈ N) do
7     | n' ← substitute(n, inst)
8     | Replace n with n' in the children of original
9   | return original
    
```

Alg. 4: Prepare User-defined Operations for Translation to SMT-LIB
(processOperations)

Inputs : (1) A set \mathcal{U} of the ASTs of the user-defined operations to process; (2) The root AST node, $root$, of an (NNF) constraint after expansion and substitution; (3) An instance model, $inst$.

Output : A set of ASTs obtained by applying substitution and expansion to the ASTs in \mathcal{U} .

Fun. calls: **expand**: see Alg. 2; **substitute**: see Alg. 3.

```

1 Let  $T$  be the AST rooted at  $root$ 
2  $\mathcal{U}_{to\ process} \leftarrow \emptyset$  /* Will store the ASTs of those (user-defined) operations that should be translated to
   SMT-LIB and thus need processing */
3  $\mathcal{U}_{processed} \leftarrow \emptyset$  /* Will store the processed ASTs of the operations */
4 foreach ( $u \in \mathcal{U}$ ) do
5   if (some node  $n$  in  $\{T\} \cup \mathcal{U}$  calls the operation for which  $u$  is the AST and  $n$  is
   labeled SMT) then
6     Let  $u_{externalized}$  be the AST resulting from transforming the primitive attributes
   within  $u$  into additional parameters for the operation corresponding to  $u$ 
7     In  $\{T\} \cup \mathcal{U}$ , replace references to  $u$  with references to  $u_{externalized}$ 
8      $\mathcal{U}_{to\ process} \leftarrow \mathcal{U}_{to\ process} \cup \{u_{externalized}\}$ 
9 Let  $\mathcal{L}$  be an (ordered) list resulting from sorting  $\mathcal{U}_{to\ process}$  such that no
    $u_i \in \mathcal{U}_{to\ process}$  depends on  $u_j \in \mathcal{U}_{to\ process}$  where  $j > i$ 
10 foreach ( $l \in \mathcal{L}$ ) do
11   if ( $l$  contains some node that needs to be either expanded or substituted) then
12     Let  $\mathcal{S}$  be the set of objects in  $inst$  that call the operation for which  $l$  is the AST
13     foreach ( $s \in \mathcal{S}$ ) do
14       Let  $op_s$  be an  $s$ -specific (renamed) copy of the operation corresponding to  $l$ 
15       Associate to  $op_s$  as its AST a (cloned) copy of  $l$ 
16       In  $\{T\} \cup \mathcal{L}$ , replace references to  $l$  with references to  $op_s$ 
17        $op_{expanded} \leftarrow \mathbf{expand}(\text{root AST node of } op_s, inst)$ 
18        $op_{substituted} \leftarrow \mathbf{substitute}(op_{expanded}, inst)$ 
19        $\mathcal{U}_{processed} \leftarrow \mathcal{U}_{processed} \cup \{\text{AST rooted at } op_{substituted}\}$ 
20   else  $\mathcal{U}_{processed} \leftarrow \mathcal{U}_{processed} \cup \{l\}$ 
21 return  $\mathcal{U}_{processed}$ 

```

Initially, Alg. 4 determines which user-defined operations require expansion and substitution (L. 1-5). Specifically, these are the operations that are either directly or indirectly used by the (processed) NNF constraint via an AST node labeled *SMT* (L. 4-5). Indirect usage means that some operation op is not called directly within the NNF constraint; however, op appears in some chain of user-defined operation calls originating from the NNF constraint.

Due to OCL's object-oriented nature, user-defined operations have access to properties that are encapsulated within the operations' calling objects. For example, the *getAge* operation in Fig. 1(a) refers to the *birthYear* attribute of its calling object. In SMT-LIB, however, only the input parameters and the built-in SMT-LIB functions are accessible within the body of a given operation (function in SMT-LIB). To get around this difference between OCL and SMT-LIB, **processOperations** externalizes as a parameter any mutable primitive attribute within the user-defined operations that need to be processed (L. 6-8 in Alg. 4). For example, the operation call **T1.getAge()** in the constraint

New header	{ getAge (extPar: Integer) : Integer
New operation query	{ if (Constants::YEAR >= extPar and Constants::YEAR >= 0) then Constants::YEAR - extPar else -1 endif

Fig. 10. The *getAge* Operation after externalizing Primitive Attributes as Parameters

of Fig. 9(b) becomes `T1.getAge(T1.birthYear)`; the OCL query that defines the operation is also updated accordingly. The *getAge* operation after externalization is provided in Fig. 10.

The **expand** and **substitute** algorithms need to be able to dynamically evaluate OCL subformulas. In the case of **expand**, this ability is required for extracting the quantified objects, and in the case of **substitute** – for calculating concrete results to replace the subformulas that should be hidden from SMT. To enable dynamic (on the fly) evaluations, we need knowledge of the calling object and parameter values of a given operation at the time the operation is processed. This knowledge is available *a priori* for the operations that are called directly by the (processed) NNF constraint. However, when an operation is called only indirectly, the knowledge needs to be obtained from the underlying chain of operation calls. To ensure that we have the required knowledge at the time we need it (during expansion or substitution), the dependencies of each user-defined operation need to be processed before the operation itself. In **processOperations**, this is achieved by ordering the user-defined operations to process according to their dependencies. Specifically, an operation in position i should not depend on an operation in position $j > i$ (L. 9 in Alg. 4).

To illustrate, let us hypothetically assume that the AST in Fig. 5 is that of a user-defined operation *op*, instead of that of an NNF constraint. In this case, *op* must be processed first to enable the identification of the objects and parameters of calls to *getAge*. The reverse order, i.e., processing *getAge* before *op*, would not be an option because we neither know the calling object, nor the parameter value to pass to the (externalized version of) *getAge* shown in Fig. 10.

The final additional consideration regarding user-defined operations has to do with the fact that the same user-defined operation may be called by different objects. This means that (the AST of) a user-defined operation may expand differently for different calling objects. Further and during the substitution process, the concrete evaluation of the same subformula might yield different results for different calling objects. To handle this, **processOperations** replaces, for each calling object, the original operation with a new (cloned) operation that is specific to the calling object (L. 10-16). This treatment does not apply to the (externalized) *getAge* operation in Fig. 10, since this operation has no quantification and has no subformulas that are exclusive to search (L. 11). For the sake of argument, let us hypothetically assume that the query defining *getAge* includes some quantification. Given the instance model of Fig. 9(a), the calls to *getAge* in the constraint of Fig. 9(b) would be replaced by **getAge_for_T1** for *T1* and by **getAge_for_C1** for *C1*, with *getAge_for_T1* and *getAge_for_C1* being clones of the *getAge* operation (Fig. 10). After this final treatment, the user-defined operations that need processing are expanded and substituted in exactly the same manner as the NNF constraint (L. 17-19).

(4) **Translation to SMT-LIB**: Alg. 5, named **OCL2SMT-LIB**, translates the now processed NNF constraint and user-defined operations to SMT-LIB. Fig. 11 presents the SMT-LIB formula obtained for the NNF constraint of Fig. 5 and the instance model of Fig. 9(a). L. 1 of the formula of Fig. 11, produced by L. 1 of Alg. 5, indicates to the SMT solver that we would like to obtain a satisfying assignment if the formula is satisfiable.

L. 2-4 of the SMT-LIB formula in Fig. 11, produced by L. 2-10 of Alg. 5, declare the inv fML enumeration types in SMT-LIB. Specifically, L. 2-4 of Alg. 5 extract all UML enumeration *types* that

Alg. 5: Translate to SMT-LIB (OCL2SMT-LIB)

Inputs : (1) The root AST node, $root$, of a processed NNF constraint; (2) A set \mathcal{U} of user-defined-operation ASTs produced by Alg. 4.

Output : An SMT-LIB translation of the constraint represented by $root$.

Fun. calls: **nameOf**: returns the name of a UML element; **SMT-LIBTypeOf**: returns the SMT-LIB counterpart type for a primitive UML type. For example, **SMT-LIBTypeOf**("Boolean") returns "Bool"; **sortByDependency**: sorts a set of operations such that the operation in position i does not depend on any operation in position $j > i$; **declareOperation**: defines an OCL operation in SMT-LIB (signature and body). For user-defined operations, this function uses **toSMT-LIB** (defined below) for generating the SMT-LIB operation body; **toSMT-LIB**: translates OCL to SMT-LIB through the rules in Table 6 of Appendix C. This function further dynamically creates the variable bindings required for lifting any satisfying assignment found by SMT back to the instance model.

```

1  res ← "(set-option: produce-models true)"
2  Let  $T$  be the AST rooted at  $root$ 
3  Let  $\mathcal{V}$  be the set of AST nodes representing the primitive attributes (variables)
   that appear in  $\{T\} \cup \mathcal{U}$ 
4  Let  $\mathcal{E}$  be the set of enumerations used as types for the variables in  $\mathcal{V}$ 
   /* Declare the required enumerations in SMT-LIB */
5  foreach ( $e \in \mathcal{E}$ ) do
6  |   res ← res + "(declare-datatypes ()((" + nameOf( $e$ )
7  |   Let  $\mathcal{L}$  be the set of literals of  $e$ 
8  |   foreach ( $\ell \in \mathcal{L}$ ) do
9  |   |   res ← res + "(" + nameOf( $\ell$ ) + ")"
10 |   res ← res + "))"
   /* Declare the SMT-LIB variables */
11 foreach ( $v \in \mathcal{V}$ ) do
12 |   res ← res + "(declare-const " + nameOf( $v$ ) + " " + SMT-LIBTypeOf( $v$ ) + ")"
   /* Define: (1) all user-defined operations, and (2) the OCL built-in operations without counterparts in
   SMT-LIB */
13 Let  $\mathcal{B}$  be the set of OCL built-in operations that (1) are used within  $\{T\} \cup \mathcal{U}$ , and
   (2) have no matching function in SMT-LIB (see Appendix C)
14 Let  $\mathcal{O}$  be the union of  $\mathcal{B}$  and the set of user-defined operations whose ASTs are in  $\mathcal{U}$ 
15  $O_{\text{sorted}} \leftarrow \text{sortByDependency}(\mathcal{O})$  /* Determine the order in which the operations should be declared
   */
16 foreach ( $o \in O_{\text{sorted}}$ ) do
17 |   res ← res + declareOperation( $o$ ,  $\mathcal{U}$ ) /*  $\mathcal{U}$  is needed for producing the SMT-LIB body of
   user-defined operations */
18 return res + "(assert " + toSMT-LIB( $root$ ) + ")(check-sat)"

```

are referenced in the processed NNF constraint and user-defined operations. Our running example in Fig. 1 includes two enumerations, namely, *country* and *Disability*. Since both enumerations are relevant to the processed NNF constraint, L. 5-10 of Alg. 5 convert them, alongside their enumeration items, to SMT-LIB enumerations as shown in L. 2-4 of the formula.

L. 5-8 of the formula, produced by L. 11-12 of Alg. 5, declare the primitive SMT-LIB variables that can be assigned by the SMT solver. These variables come from converting to SMT-LIB each

```

1. (set-option:produce-models true)
2. ; Required enumeration types
3. (declare-datatypes 1)((Country (LU)(FR)(BE)(DE)(Other)))
4. (declare-datatypes 2)((Disability (None)(Vision)(A)(B)(C)(D)(E)))
5. ; Variables to assign
6. (declare-const X1 Bool)(declare-const X2 Country)(declare-const X3 Bool)
7. (declare-const X4 Int)(declare-const X5 Disability)(declare-const X6 Real)
8. (declare-const X7 Int)(declare-const X8 Real)(declare-const X9 Disability)
9. ; Operations requiring SMT-LIB definitions
10. (define-fun getAge ((P1 Int) Int (ite (and (>= 2018 P1) (>= 2018 0))(- 2018 P1) -1))
11. ; Formula to solve
12. (assert (and true (and true (and true (and true (and (and (and (or X1 (not (= X2 ZZ))))
13. (or (not X1) (or X3 (= X2 ZZ)))) true) (and (and (and (and (<= (getAge X4) 100) (>=
14. (getAge X4) 0)))(or (and (= X5 None) (= X6 0)) (and (and (> X6 0) (<= X6 1))(not (= X5
15. None)))))) true) (and (and (and (<= (getAge X7) 100) (>= (getAge X7) 0))(or (and (= X9
16. None) (= X8 0))(and (and (> X8 0) (<= X8 1)) (not (= X9 None)))))) true))))(check-sat)

```

Fig. 11. SMT-LIB Formula for the NNF Constraint of Fig. 5 and Instance Model of Fig. 9(a)

primitive attribute of the instance model of Fig. 9(a) that is referenced (directly or indirectly) by the constraint at hand. For example, all attributes in the instance model in Fig. 9(a) are concerned by the translation since they are of primitive types and are accessible from the processed NNF constraint. The rules for declaring SMT-LIB variables are simple and can be found in Table 6 of Appendix C. For example, and as shown by the SMT-LIB formula in Fig. 11, the attributes *isResident* and *country* of the instance model in Fig. 9(a) translate to **(declare-const X1 Bool)** and **(declare-const X2 Country)**, respectively. We note that, as shown by the annotations in Fig. 9(a), each declared SMT-LIB variable is bound to its corresponding primitive attributes in the instance model, e.g., X1 and X2 respectively refer to the *isResident* and *country* attributes. Later, if the solver finds a solution, these bindings will be used for lifting back satisfying assignments to the instance model.

L. 9-10 of the formula, produced by L. 13-17 of Alg. 5, define the required built-in and user-defined OCL operations in SMT-LIB. For the NNF constraint in Fig. 5, only *getAge* needs to be defined. The generation of the SMT-LIB header for *getAge* is straightforward (L. 17 of Alg. 5). The body of the corresponding SMT-LIB function is built by conducting a depth-first traversal over the AST of *getAge* (shown in Fig. 10) and applying a set of predefined OCL to SMT-LIB translation rules (L. 17 of Alg. 5). For example, the if-condition (**Constants::YEAR >= extPar and Constants::YEAR >= 0**) in *getAge* of Fig. 10 becomes **(and (>= 2018 P1) (>= 2018 0))** in the SMT-LIB formula of Fig. 11, with P1 being a parameter of the SMT-LIB function for *getAge* and **2018** being the constant value for the static attribute *YEAR* of the class *Constants* in Fig. 1. Our complete list of OCL to SMT-LIB translation rules are provided in Appendix C.

Several built-in OCL operations such as **abs**, **div**, and **mod** are available as default functions in SMT-LIB. Such operations do not require explicit SMT-LIB function definitions. Built-in OCL operations without a counterpart SMT-LIB function, e.g., **max** and **round**, are defined, when needed, using the rules listed in Appendix C (L. 13-17 of Alg. 5). For example, **max** in OCL is translated into the following SMT-LIB function: **(define-fun max ((a Real)(b Real)) Real (ite (>= a b) a b))**.

L. 11-16 of the formula in Fig. 11, produced by L. 18 of Alg. 5, represent the core assertion derived from the processed NNF constraint. This assertion is produced by applying the OCL to SMT-LIB translation rules listed in Appendix C. Note that these rules are the same as those used for translating user-defined operation into SMT-LIB.

6.4.2 Invoking the SMT Solver and Updating the Instance Model. Given the binding established between the variables in the constructed SMT-LIB formula and the primitive attributes in the instance model (see Fig. 9(a)), updating the instance model is straightforward. For example, when

invoked over the SMT-LIB formula of Fig. 11, the solver may return the following satisfying assignment: $X1 \leftarrow \text{false}$, $X2 \leftarrow \text{FR}$, $X3 \leftarrow \text{false}$, $X4 \leftarrow 1918$, $X5 \leftarrow \text{A}$, $X6 \leftarrow 1$, $X7 \leftarrow 1975$, $X8 \leftarrow 1$, and $X9 \leftarrow \text{A}$. Using the binding shown in Fig. 9(a), the instance model is updated by lifting back this satisfying assignment. In our example, the updated instance model is already a valid solution and thus returned to the user.

7 EVALUATION

In this section, we evaluate the PLEDGE approach over three industrial case studies.

7.1 Research Questions (RQs)

RQ1. How does PLEDGE fare against the state of the art in terms of success rate and execution time? We examine in RQ1 whether PLEDGE presents practical improvements over Alloy [19], UMLtoCSP [20], and pure search as implemented by the baseline we build on [76] (see Section 6.2).

RQ2. Can PLEDGE generate large instance models in practical time? The need to test a large variety of system scenarios and system robustness and performance, e.g., load and stress testing [86], entails large test inputs. RQ2 investigates how PLEDGE scales as we have it produce increasingly larger instance models.

RQ3. Does PLEDGE offer any scalability advantage over the alternatives, when PLEDGE is not the obvious best choice according to the results of RQ1? The goal of this RQ, posed as a follow-on to RQ1, is to provide insights about the scalability of PLEDGE in comparison to Alloy, UMLtoCSP, and the baseline. As we explain in our answer to RQ1, these alternatives are equally good or better than PLEDGE in one out of our three case studies. For that particular case study, RQ3 contrasts PLEDGE's scalability trends seen in RQ2 against those of the alternatives. Intuitively, RQ3 investigates whether PLEDGE can provide an advantage over the alternatives for building *large* instance models in the case study where PLEDGE has no benefit over the alternatives otherwise.

7.2 Tool Support

Fig. 12 provides an overview of the PLEDGE tool. *NNF Transformer* turns the input OCL constraints into an NNF constraint (Section 5). *Constraint Labeler* spreads the solving tasks over search and SMT (Section 6.1). *Solver* implements the three iterative steps depicted in Fig. 6 (Sections 6.2 through 6.4). The PLEDGE tool uses Eclipse OCL [33] for building ASTs and evaluating OCL expressions, the OCL solver from our previous work [77] for search-based solving, and Z3 [26] for SMT solving. Excluding comments and third-party libraries, the PLEDGE tool consists of $\approx 16\text{K}$ lines of Java code. The tool is available online at <https://sites.google.com/view/hybridoclsolver/>.

7.3 Description of Case Studies

Our evaluation is based on three industrial case studies from three distinct domains. The case studies are denoted Case A, Case B, and Case C. The source material for each case study is a test data model expressed using UML using Class Diagrams (CDs) and OCL. Our case study material is

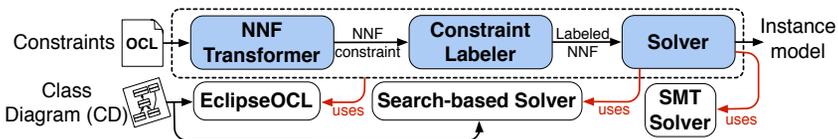


Fig. 12. PLEDGE's Architecture (new modules implemented as part of this article are shaded blue)

borrowed from previous work. Specifically, Case A was built for system testing of an eGovernment application concerned with calculating personal income taxes [77], Case B for system testing of an occupant detection system in cars [82], and Case C for system testing of a satellite communication system [28]. Our case study material (in a sanitized form) is available from PLEDGE’s website. The example described in Section 2 and used throughout the article is a simplified excerpt from Case A.

The existing work strands from which our case study material originates employ different test strategies. In particular, Case A uses statistical testing [68], Case-B uses coverage-driven model-based testing [1], and Case-C uses data mutation testing [85]. Despite the different test strategies, the case studies all share a common technical challenge, which is to find a practical way of producing system test data under OCL constraints. Our evaluation in this article is not meant at assessing the effectiveness of the test strategies in the prior work where our case study material comes from. Instead, we evaluate the applicability and scalability of the PLEDGE approach for system test data generation, irrespectively of the test strategy.

Table 1 provides overall statistics about the CDs and OCL constraints in our case studies. Specifically, rows 1 to 5 in Table 1 provide various size measurements for the CDs. The three CDs have large and comparable sizes, with Case A including the most structural features such as associations and generalizations, and Case C including the highest number of attributes. Nevertheless, we note that the constraint statistics (rows 6 to 10 of Table 1) are for the single NNF constraint derived in each case study (see Section 5), rather than the original constraints. This provides a more convenient basis for comparing the complexity of the solving tasks across the case studies when we discuss the evaluation results next.

Row 6 in Table 1 shows the number of nodes within the ASTs of the NNF constraints built for Case A, Case B, and Case C. To help the reader relate to these numbers, we note that the AST of the NNF constraint in Fig. 5 includes only 253 nodes. This means that the NNF constraint of Case C is ≈ 15 times larger than that of the running example. Rows 7 to 10 in Table 1 intuitively provide information about how search and SMT share the solving tasks for Case A, Case B, and Case C. From these numbers, we observe that the task-split balance between search and SMT differs across the case studies. Specifically, Case A and Case C have more subformulas delegated to SMT than search, with the situation being the opposite for Case B.

Table 1. Statistics for the Case Studies

		Case A	Case B	Case C	
Class Diagram	1	# of classes	64	57	55
	2	# of generalizations	43	35	20
	3	# of associations	53	44	30
	4	# of enumerations (Avg # of literals per enumeration)	17 (6.11)	4 (3.25)	8 (4.87)
	5	# of attributes (primitive + non-primitive)	208 (158+50)	116 (72+44)	255 (219+36)
NNF Constraint	6	# of nodes in the AST of the NNF constraint	2004	1844	4014
	7	# of AST nodes labeled <i>search</i>	574	1128	1814
	8	# of AST nodes labeled <i>SMT</i>	1239	602	2017
	9	# of AST nodes labeled <i>both</i>	63	9	144
	10	# of unlabeled AST nodes (representing constants)	128	55	39

7.4 Results and Discussion

In this section, we answer the RQs of Section 7.1 via experimentation. Our experiments were performed on a computer with a 3GHz dual-core processor and 16GB of memory. The maximum number of iterations for metaheuristic search was set to 1000. The maximum number of iterations for metaheuristic search was set to 1000. This choice was made based on experimentation with the (purely search-based) baseline solver. In particular, we observed that, over our case studies, the fitness function plateaued within 1000 iterations; in other words, improvements in the fitness function value were insignificant past 1000 iterations. Further, and as importantly, all successful runs of the baseline solver required less than 1000 search iterations.

RQ1. We ran PLEDGE, the baseline search-based solver, UMLtoCSP (Version from 2009 with its default ECLiPSe solver V5.0 [22]), and Alloy (V5.0) on the case studies to which they are applicable. Both Alloy and UMLtoCSP have user-specified parameters for bounding the search space. In Alloy, such bound is provided through the “scope” parameter [44]. As an example, if the scope is set to five, the Alloy Analyzer will attempt to instantiate each entity at most five times when searching for a solution. UMLtoCSP works in a similar manner, but requires bounds to be specified for individual classes and associations. These bounds have a significant impact on the performance of Alloy and UMLtoCSP. The general recommendation is to use the smallest bounds within which a solution exists [44]. In RQ1, and specifically for Case B where Alloy and UML2CSP are applicable, we set Alloy’s scope parameter to two; no solution exists when the scope is set to one. Similarly, for UMLtoCSP, we set to two the instantiation bounds for all classes and associations. Further, Alloy uses SAT solving as backend, and comes prepackaged with several alternative SAT solvers. We considered two alternatives: SAT4J [11] – Alloy’s default SAT solver – and miniSat [34] – the SAT solver recommended in Alloy’s documentation for better performance [19]. The results we report in RQ1 and RQ3 on Alloy are the ones obtained using miniSat, noting that Alloy used with miniSat consistently outperformed Alloy used with SAT4J in all our experiments.

PLEDGE and the baseline solver are applicable to all three case studies. Alloy and UMLtoCSP are applicable to Case B and only for a scope higher or equal to two. In RQ1, as mentioned earlier, the scope was set to a minimum of two. Case A and Case C have features which are common in practice, but which Alloy and UMLtoCSP do not support. In the case of Alloy, the main issues for which we could not find a workaround had to do with the absence of real / floating-point arithmetic and string operations. In the case of UMLtoCSP, the main limitation was the lack of support for `null` and `oclInvalid`. This meant that several OCL operations used in our case studies, e.g., `oclIsUndefined` (illustrated in Fig. 1(b)) and `oclIsInvalid`, could not be handled by UMLtoCSP. We note that for Case B, UML2Alloy [5] (V0.52) turned out to be adequate for translating that case study to Alloy’s input language. The results we report here (and for RQ3) for Alloy are based directly on the Alloy model derived by UML2Alloy. The derived model can be found on our tool’s website.

For each solver considered, we measured the execution time. To account for random variation, we ran the solvers 100 times each. We computed the success rate, i.e., how often each solver succeeded in finding a solution, based on the 100 runs. The CDs in our case studies happen to have a root class, i.e., a class from whose instances one can reach all other objects in an instance model. In Case A, this class is *Household* – a unit of taxation (not shown in our example of Fig. 1). In Case B, the class is *BodySenseSystem* – a container for all system components. In Case C, the class is *Vedu* – a container for data packages transmitted over a satellite channel. Through the notion of non-emptiness constraint discussed in Section 5, we instructed each applicable solver to generate exactly one instance of the root class. This is a natural choice for comparison purposes, since a single instance of the root class represents the smallest meaningful (system) test input in each of our case studies. Had the case studies not had a root class, we would have chosen multiple

Table 2. RQ1 Results

		Case A		Case B				Case C	
		Baseline	PLEDGE	Alloy	UMLtoCSP	Baseline	PLEDGE	Baseline	PLEDGE
1	Succeeded?	28 / 100 runs	100 / 100 runs	100 / 100 runs	28 / 100 runs	100 / 100 runs	100 / 100 runs	0 / 100 runs	100 / 100 runs
2	Average execution time	587.2 sec SD = 383.2	27.8 sec SD = 18.25	0.045 sec SD = 0.005	1.56 sec SD = 0.055	4.5 sec SD = 1.7	3.6 sec SD = 0.7	1771 sec SD = 538.9	44.3 sec SD = 7.2
3	Average instance model size (for successful runs)	28.8 obj. SD = 5.8	36.6 obj. SD = 7.1	21 obj. SD = 0	19 obj. SD = 0	19.7 obj. SD = 2.1	23.3 obj. SD = 3.1	–	121.4 obj. SD = 6.1

classes to instantiate in a way that would match system testing needs. We further note that one can seed PLEDGE and the baseline solver with a partial (and potentially invalid) instance model as the starting point. We did not use this feature; both solvers used the default option, which is to start from an *empty* instance model.

Table 2 shows the results of RQ1, alongside the size of the generated instance models. Since each solver was executed 100 times, Table 2 reports averages (Avg) and standard deviations (SD) for the execution time and instance model sizes.

As seen from the table, PLEDGE was applicable to all three case studies and able to produce results in all its runs. In Case B, all four solvers were equally successful and very quick, with Alloy being the quickest. How the execution time trends observed over Case B evolve as one attempts to build larger instance models is assessed in RQ3.

With regard to the performance of PLEDGE compared to the baseline, the results in Table 2 suggest that PLEDGE brings no practical benefit in Case B. This can be explained as follows: The primitive attributes in Case B are less constrained than those in Case A and Case C (rows 8 and 9 of Table 1). We further observed that most operations on the primitive attributes in Case B are simple and of the form $x \langle \text{op} \rangle \text{constant}$, where x is a primitive attribute and $\langle \text{op} \rangle$ is an equality or inequality operator. Consequently, in Case B, search was as effective as SMT in handling primitive attributes.

In Case A and Case C, PLEDGE has better effectiveness (success rate) and efficiency (execution time) compared to the baseline. As shown by the first row of Table 2, PLEDGE was able to produce valid instance models in all its runs. In contrast, the baseline solver produced valid instance models in only 28% of the runs in Case A and in none of the runs in Case C. Specifically, in Case A, PLEDGE outperformed the baseline by a factor of ≈ 3.5 in terms of success rate and a factor of ≈ 21.7 in terms of execution time. In Case C, only PLEDGE was able to generate data. Further, and as indicated by the second row of Table 2, PLEDGE has a smaller execution time than the baseline for Case A and Case C: in Case A, the average execution time of PLEDGE was 27.8 seconds versus 587.2 seconds for the baseline, and, in Case C – 44.3 seconds versus 1771 seconds.

Statistically speaking, in terms of success rate, PLEDGE has significant benefits over the alternatives for Case A and Case C. In particular, and according to the statistical Z-test for the significance of proportions, p-values are $< 10^{-5}$. Recall from Table 2 that for Case B, all the tools have a 100% success rate. As for the execution time, PLEDGE is significantly better than the baseline in Case A and Case C, recalling that Alloy and UMLtoCSP are not applicable to these two case studies. In particular, the p-values from a t-test over the execution time distributions of PLEDGE and the baseline are $< 2.2 \cdot 10^{-16}$ and $< 3.268 \cdot 10^{-15}$ for Case A and Case C, respectively. In Case B, Alloy has a significantly better execution time; p-values are $< 2.2 \cdot 10^{-16}$ against PLEDGE, UMLtoCSP, and the baseline.

Although Case A and Case B are similar with respect to the size of the NNF constraint (row 6 of Table 1) and the size of the generated data (row 3 of Table 2), the performance difference between PLEDGE and the baseline solver is much larger in Case A than in Case B. The better performance of PLEDGE in Case A is due to SMT being more efficient than search at handling linear arithmetic, which is not only more prevalent in Case A, but also more complex. Notably, Case A has numerous inequalities with more than one primitive attribute and using OCL operations such as `max` and `round`.

Case C is inherently more complex than Case A and Case B, as suggested by Tables 1 and 2. The additional complexity has a detrimental effect on the baseline solver, resulting in all runs of the baseline solver on Case C to be unsuccessful. This outcome does *not* imply that the baseline solver is theoretically bound to fail on Case C. Rather, the outcome is a strong indication of the baseline solver being inefficient for Case C.

The answer to RQ1 is that, for one of our case studies, Case B, which has comparatively simple constraints, all the techniques considered were as effective, and produced results within seconds. For the other two case studies, Case A and Case C, Alloy and UMLtoCSP were not applicable, thus reducing the set of alternatives to pure search and our hybrid approach. How PLEDGE fares against pure search depends on the complexity of the constraints over primitive attributes. If such constraints are few and simple, both approaches have similar performance. But, when primitive attributes are subject to many complex restrictions, the PLEDGE approach performs substantially better. In Case A, PLEDGE was ≈ 3.5 times more successful and ≈ 21.7 faster than pure search. In Case C, only PLEDGE yielded results.

RQ2. We measured the execution time of PLEDGE for building progressively larger instance models – a necessity for system testing, as noted earlier. Whereas in RQ1 we set to one the number of instances of the root class, in RQ2, we initially set this number to 20 and increased it up to 100 in increments of 20. Due to the nature of the root class (see RQ1), increasing its number of instances has a knock-on effect, leading to an across-the-board increase in the size of the instance model. This knock-on effect comes primarily from the fact that most NNF-constraint fragments are of the shape `Context.allInstances()->forall(condition)`, potentially with several nested layers of quantification. The more instances of `Context` there are, the more computationally intensive solving becomes, particularly the expansion and fitness function calculations.

Despite the generated data having multiple roots, it is important to note that the data is not necessarily the union of independently constructed single-rooted instance models. This is because the roots are often logically connected. For a simple example from Case A, taxpayers are required to have distinct identifiers; this induces a global relationship between all taxpayers in all households (roots). For another example from the same case, consider child support which plays a role in several of the constraints. Case A allows children to be supported by taxpayers from different households; this is necessary, for example, when the parents of a child are divorced, with the child being in one household and the (financially) supporting parent in another. To build a realistic instance model, we need to cover such situations and thus be able to create links between objects falling under different roots. In short, building an instance model with multiple roots is not always decomposable into simpler solving tasks.

As a final remark, we note that, although this has no bearing on RQ2, we did not attempt to maximize diversity when creating multiple class instances in the instance model. If necessitated by the test strategy, diversity can be effectively enforced by dynamically updating the OCL constraints with new inequalities to guide data generation toward diversity [77].

Fig. 13 shows for each case study the execution time of PLEDGE over instance models containing 20 to 100 instances of the root class. To account for random variation, we calculated average execution times based on ten runs. For example, and as shown in Fig. 13, the average execution time

for generating an instance model with 60 instances of the root class for Case A, Case B, and Case C is 16.79 minutes (SD=29.4), 1.79 minutes (SD=0.84), and 143.64 minutes (SD=79.7), respectively. The average execution time for Case C is the highest, while that of Case B is the lowest. This is because, as we explain in RQ1, the constraints in Case C are inherently more complex than those in Case A and Case B (see rows 6 to 10 in Table 1), while those in Case B are the simplest, i.e., mostly simple inequalities over Boolean attributes.

In Fig. 13, we further report the average number of objects in the generated instance models (n). For example, on average, the largest instance models generated for Case A, Case B, and Case C contain 3.5K, 2.3K, and 10.1K objects, respectively. Although Case C has fewer classes and associations than Case A and Case B (rows 1 and 3 of Table 1), it requires the generation of a significantly higher number of objects to satisfy the multiplicity and user constraints (encoded in the NNF constraint). Subsequently, and as shown in Fig. 13, the execution times measured for Case C are significantly higher than those of Case A and Case B. In general, to various degrees but across all studies, the increase in execution times is exponential.

In addition, Fig. 13 provides a breakdown of the execution times over the iterative steps of our hybrid solving process, i.e., Steps 2 through 4 of Fig. 6. For example, generating an instance model with 60 instances of the root class for Case C took in total 143.63 minutes, with 76.58, 2.11, 64.94 minutes spent on Steps 2, 3, and 4, respectively. The breakdown indicates that the time required for Step 3 (checking whether SMT should be invoked) is negligible. The proportions of the execution times for Steps 2 and 4 reflect how the solving load is spread over search and SMT, taking into account both the characteristics of the NNF constraint to solve and the size of the instance model.

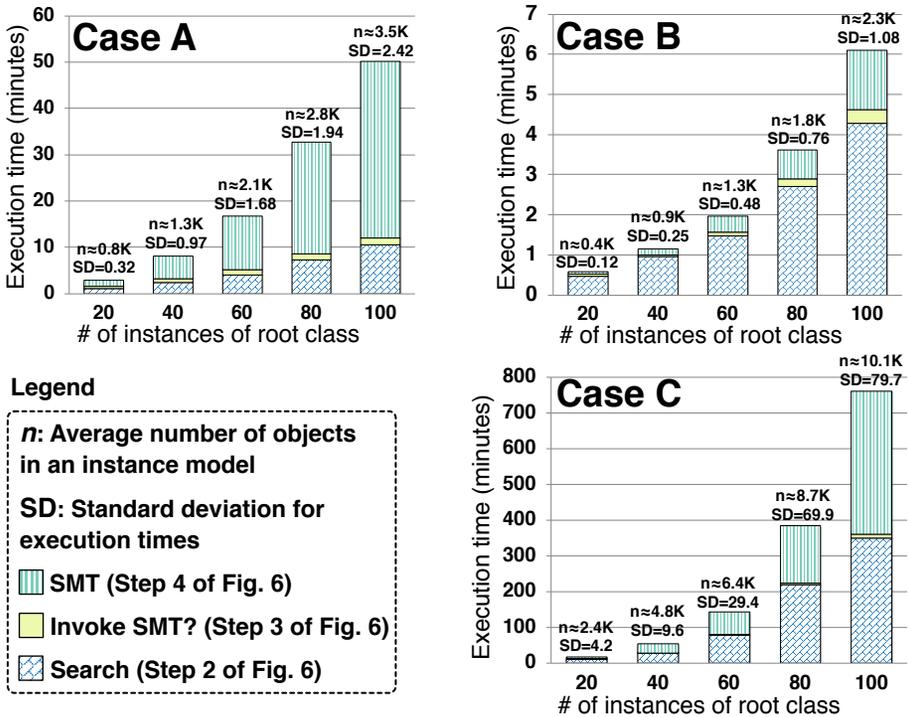


Fig. 13. Execution Times for PLEDGE (RQ2)

Across all the $3 \times 5 \times 10 = 150$ runs underlying the results of Fig. 13, 79.5% of the execution time in Step 4 was spent on expansion and substitution, 0.7% on SMT solving by Z3, and the remaining 19.8% on lifting the output of Z3 back to the instance model. From these percentages, we observe that the time spent by Z3 is proportionally small. This suggests that the subformulas delegated to SMT have been handled by efficient decision procedures.

We note that the proportionally high execution times of the SMT part is due to the expansion and substitution processes (Algs. 2 and 3) being computationally expensive. Further, these processes may have to be executed multiple times because an instance model may pass the check in Step 3 of Fig. 6, meaning that the SMT step will be invoked, and yet the constructed SMT-LIB formula may turn out to be unsatisfiable. As a consequence, the (cumulative) execution time of the SMT part could be significant, and, as we observe in Case A and Case C, it could be even higher than the execution time of the search part.

As expected, Fig. 13 suggests an exponential trend in the execution times with the size of the instance models increasing. Despite this, all runs of our solver were successful, meaning that the solver maintained its 100% success rate (reported in RQ1) over substantially larger instance models. In Case A and Case B, which, as we argued in RQ1, are less complex than Case C, the execution times increase slowly and within reasonable range. In Case C, one needs to take additional measures for scalability, should one need instance models that are much larger than those in our experiments. Particularly, one may utilize existing mechanisms for enhancing the performance of constraint solving, e.g., model slicing [71, 74], parallelization [66, 67], and bound reduction [21].

The answer to RQ2 is that, for our most complex case study, Case C, PLEDGE generated valid data samples with over 10K objects in less than 13 hours. We believe this level of performance is practical for testing, considering the following two factors: First, one does not have to wait until testing time to initiate the generation of data; this can be done well in advance and as soon as one has a stable test data model at hand. Second, given the observed execution times of PLEDGE over three industrial case studies, we anticipate that one would be able to generate large volumes of (logically valid) data overnight, thus minimizing the wait time by test engineers.

RQ3. As shown in Table 2 and discussed in RQ1, all the tools we have experimented with were applicable to Case B and could quickly produce an instance model containing a single instance of the root class in that case study. RQ3 addresses a natural follow-on question: When all the tools happen to be applicable in a particular situation, which one would be the best to use for generating *large* volumes of system test data? Given the results of RQ1, we elaborate RQ3 into two more detailed questions: (RQ3.1) Can one expect Alloy and UMLtoCSP to maintain their execution-time advantage over PLEDGE when they are tasked with generating larger instance models? And similarly, (RQ3.2) can PLEDGE maintain its advantage over the baseline solver as the instance models grow in size?

To answer RQ3.1 and RQ3.2, we subjected Alloy, UMLtoCSP, and the baseline solver to the same experiment as in RQ2, but restricted to Case B only. For each of these three tools, we measured the time it took to generate valid instance models with 20, 40, 60, 80, and 100 instances of the root class (in Case B). Similarly to RQ1, for each selected number of root classes, we set the scope parameters of Alloy and UMLtoCSP to a minimum within which they could find a solution. For example, the minimum scope that would enable Alloy to produce 40 valid instances from the root class of Case B is 80, as all attempts with a smaller scope failed. In other words, the results we report in RQ3 for Alloy and UMLtoCSP represent the best average execution times that one can achieve, noting that in a real-world setting, practitioners would not know what the minimum scope is without experimentation. As for the baseline solver, we set the maximum number of iterations to 1000; this was sufficient for the baseline solver to succeed in all of its runs in the above-described experiment.

Table 3. Average Execution Times for Alloy, UMLtoCSP, the Baseline Solver and PLEDGE over Case B

		Number of Requested Root Class Instances				
		20	40	60	80	100
Average execution time	Alloy	0.98 min SD = 0.05 p-value = $1.9 * 10^{-6}$	5.93 min SD = 0.24 p-value = $1.2 * 10^{-15}$	17.73 min SD = 1.41 p-value = $6.5 * 10^{-11}$	44.8 min SD = 2.03 p-value = $1.7 * 10^{-13}$	124.82 min SD = 2.47 p-value = $2.2 * 10^{-16}$
	UML2CSP	83.69 min SD = 2.48 p-value = $2.9 * 10^{-7}$	Memory Overflow	Memory Overflow	Memory Overflow	Memory Overflow
	Baseline Solver	7.46 min SD = 5.21 p-value = $2.4 * 10^{-5}$	33.65 min SD = 3.82 p-value = $5.1 * 10^{-5}$	79.25 min SD = 9.98 p-value = $2.3 * 10^{-5}$	100.75 min SD = 8.03 p-value = $4 * 10^{-4}$	134.93 min SD = 16.76 p-value = $1.7 * 10^{-5}$
	PLEDGE	0.47 min SD = 0.12	1.04 min SD = 0.25	1.8 min SD = 0.48	3.13 min SD = 0.76	5.58 min SD = 1.08

Table 3 shows the execution times of Alloy, UMLtoCSP and the baseline solver for producing valid instance models containing 20 to 100 instances of the root class in Case B. For easier comparison, the table further shows (in the last row) the results for PLEDGE, repeated from RQ2. To account for random variation and in a similar manner to RQ2, we calculated average execution times and their standard deviation (SD) based on ten runs. For example, Alloy took on average 124.82 min (SD = 2.47) to generate instance models with 100 instances of Case B's root class. We note that UMLtoCSP was able to generate only instance models with up to 20 instances of the root class; all attempts with larger sizes (40, 60, 80, and 100) resulted in memory blow-up. For each selected number of root instances, Table 3 further presents the t-test p-values for the execution times of PLEDGE compared against those of Alloy, UMLtoCSP, and the baseline. For example, the p-value between the execution-time distribution of PLEDGE and that of Alloy, when generating 100 instances of the root class, is $2.2 * 10^{-16}$. The computed p-values indicate that the disparities between the execution times of PLEDGE and those of the considered alternatives are statistically significant.

As seen from the table, on average, UMLtoCSP takes approximately 178 times longer to produce an instance model with 20 instances of the root class when compared to PLEDGE. As for the baseline, the average execution time is approximately 14, 32, 44, 31, and 24 times longer than that of PLEDGE for producing instance models with 20, 40, 60, 80, and 100 instances of the root class, respectively. Finally, and as for Alloy, the average execution times are consistently higher than those of PLEDGE. For instance models with up to 40 instances of the root class, Alloy performs reasonably well in comparison; however, for larger instance models, the performance difference between Alloy and PLEDGE becomes increasingly more pronounced. As an extra step and to examine whether the performance difference seen above could be narrowed, we experimented with an evolutionary-based version of Alloy, named EvoAlloy [83]. We tried various configurations for EvoAlloy over Case B (where EvoAlloy is applicable) by varying the population size and the cross-over and mutation probabilities; however, the tool was outperformed by Alloy in all the configurations that we attempted. Consequently, considering that Alloy is already included in our analysis, we chose to not include EvoAlloy in Table 3, since this would not bring additional insights into the discussion.

The answer to **RQ3** is that Alloy and UMLtoCSP did not maintain the performance edge they had over Case B for larger instance models. UMLtoCSP produced results only for the smallest case experimented with in RQ3. Alloy was consistently outperformed by PLEDGE in all the RQ3 experiments. From a practical standpoint, Alloy scaled well up to a certain threshold, but beyond that, the performance difference between PLEDGE and Alloy became more evident. The baseline solver was consistently outperformed by both PLEDGE and Alloy. In short, the RQ3 results suggest that PLEDGE is the fastest among the alternatives considered for generating large volumes of system testing data.

8 LIMITATIONS AND THREATS TO VALIDITY

Limitations. PLEDGE is non-exhaustive. This means that, given a size bound on the instance model to build (the universe), PLEDGE can neither guarantee that it will find a solution within this bound when there is one, nor guarantee the absence of such a solution when there are none. In other words, PLEDGE forgoes completeness for better applicability and scalability of constraint solving. As a result, and more precisely speaking, PLEDGE is unable to prove bounded (un)satisfiability. This is not a major limitation in our context, since our goal is not to provide proofs of (un)satisfiability, but rather to generate test data. In contrast to completeness, soundness is guaranteed by PLEDGE. More precisely and as we explain below, PLEDGE is as sound as its underlying OCL evaluator, Eclipse OCL (see Section 7.2). Eclipse OCL is the de-facto OCL evaluation tool, leaving little to no room for instance models that are logically invalid to be deemed valid or vice versa. As discussed in Section 4, PLEDGE enforces by construction the basic constraints related to class diagrams, e.g., type conformance for association ends. Further, the derived NNF constraint is logically (SAT- and UNSAT-) equivalent to the conjunction of the user and multiplicity constraints, recalling from Section 5 that the theoretical basis for deriving NNF constraints are De Morgan's transformation rules. Consequently, when PLEDGE deems an instance model I to be valid because it satisfies the NNF constraint, I is indeed a valid solution for the original constraints as per the evaluation performed by Eclipse OCL.

Generally speaking, in any situation where both search and SMT are inefficient, PLEDGE will be inefficient too. One such situation (not encountered in our case studies) is when the constraints contain recursive user-defined operations. We inherit some limitations from the search-based solver we build on. The solver, despite covering the entire OCL, provides only coarse-grained fitness functions for certain constructs, e.g., `subOrderedSet` [77]. This does not pose a practical problem, since the constructs in question are rarely used.

Threats to validity. Internal and external validity are the validity aspects most pertinent to our empirical evaluation.

With regard to internal validity, we need to note the following: Alloy requires upfront bounds on signature (class) instantiations; and UMLtoCSP requires bounds on both class and association instantiations. Selecting these bounds is critical and yet non-trivial, noting that large bounds often lead to time and / or memory blow-up. In our experiments, we were as conservative as possible when comparing to Alloy and UMLtoCSP. Specifically, we minimized the bounds with hindsight from the hybrid and baseline solvers' generated data. By giving Alloy and UMLtoCSP this advantage, we mitigate as much as possible the confounding effects posed by the tuning of the bounds.

As for external validity, we note that we applied PLEDGE to three industrial case studies from different domains. The evaluation results are easy to interpret, and clearly suggest that PLEDGE is widely applicable and beneficial, particularly for complex data generation problems that are common in system testing practice. We also note that our three case studies were developed by

different modeling teams in different domains. The team members all have background in model-driven engineering and testing, with levels of experience ranging from four to more than twenty years. The material for each case study was validated by experts from the underlying domain (legal experts for Case A, automotive experts for Case B, and satellite communications experts for Case C). Our case study material was developed and finalized long before we started working on PLEDGE. This material was used as-is, i.e., without any changes, in our evaluation. Note that, for confidentiality reasons, the case study material we provide online has been anonymized, but from a logical standpoint, this online material is equivalent to the original material; PLEDGE yields the same results on the anonymized case studies as on the original ones. We believe that our case studies provide realistic, diverse, and unbiased examples of what PLEDGE needs to be able to handle in practice. This said, as with any type of case study research, additional case studies remain necessary for gaining further confidence in the generalizability of our approach.

9 RELATED WORK

Numerous approaches exist for automated test data generation. Several of these approaches work by deriving test data from source code [43]. For example, CAUT [78] uses dynamic symbolic execution and linear programming to generate unit test data for C programs. Given a small size bound, Korat [12] generates all nonisomorphic test inputs for Java programs annotated using the Java Modeling Language (JML) [15]. SUSHI [13, 14] combines symbolic execution and metaheuristic search to built input data that can exercise a given path in Java code.

White-box approaches like the above are usually inapplicable to system testing, which, as we noted earlier, is a primarily black-box activity [81]. In particular, white-box approaches cannot be easily used when the source code is composed of several heterogeneous languages or when the code is inaccessible / unavailable due to reasons such as confidentiality and the presence of third-party components. These situations are common when testing at the system level. In contrast to white-box test data generation approaches, PLEDGE requires no knowledge about the source code of the system under test, and can be applied as soon as one has a data model characterizing the system inputs.

PLEDGE relates more closely to those that work based on a conceptual specification of data. In the context of UML, most such approaches utilize either only Class Diagrams [2, 3, 23], or a combination of Class Diagrams and behavioral models such as State Machines [41] and Sequence Diagrams [50]. Regardless of the exact formulation of UML-based data generation, a core challenge in this line of work is the satisfaction of OCL constraints [35]. The most commonly used strategy for OCL solving is through translation to SAT, SMT, and other formalisms equipped with solvers. Alloy [44] and UMLtoCSP [16] are two notable approaches in this class. Alloy is a general-purpose SAT-based analyzer for first-order logic. Alloy can act as an OCL solver when complemented with a tool such as UML2Alloy [6], which adapts UML class diagrams and OCL constraints to Alloy's input language. UMLtoCSP, which is specifically geared toward OCL solving, has as translation target a Prolog-based constraint programming language. Since Alloy and UMLtoCSP are currently the main technologies used for OCL solving, we empirically compared PLEDGE against them in our evaluation (Section 7). In addition to Alloy and UMLtoCSP, there are several threads of work where OCL constraints are solved through translation, with the main translation targets being SAT [49, 61, 75] and SMT [18, 23, 62].

The above approaches have two main limitations: First, they all have to translate class diagrams and OCL constraints into other formalisms. Since the formalisms considered are not sufficiently expressive, compromises have had to be made in terms of supported features. For example, we faced applicability problems with Alloy and UMLtoCSP in two out of our three case studies. Another

limitation is that these approaches do not scale well for test data generation; we empirically demonstrated this for Alloy and UMLtoCSP.

To mitigate challenges posed by undecidability and the lack of scalability, there has been work on identifying decidable fragments in OCL [57, 58, 63]. In practice, narrowing OCL to decidable fragments is problematic, since doing so often makes certain constraints inexpressible, and further poses a usability challenge to users, even when the constraints can be written in a decidable fragment.

An alternative strategy for OCL solving is metaheuristic search [2, 3, 70, 77]. Due to their non-exhaustive nature, search-based approaches cannot provide guarantees about either satisfiability or unsatisfiability. Nevertheless, for activities such as simulation and testing, such guarantees are not necessary, noting that the objective of OCL solving here is not to assess the quality of a model, but rather generate synthetic data from a model that has been validated a priori. Search-based OCL solving approaches are parallelizable, and have been shown to scale well for large and complex data generation problems. Further, these approaches cover nearly the entire OCL language – a direct benefit of not requiring a translation to another formalism – and thus present a major applicability advantage. Despite these advantages, search is a randomized process and not as effective as a decision procedure for handling the decidable fragments of OCL.

Metaheuristic search has also been considered for solving Alloy specifications. Notably, using Alloy’s intermediate relational models as seeds, EvoAlloy provides a population-based evolutionary algorithm aimed at improving the success rate of solving large Alloy specifications [83]. EvoAlloy, by virtue of being based on Alloy’s specification language, inherits the expressiveness restrictions and applicability issues of Alloy. In the context of our work and as discussed in Section 7.4, EvoAlloy is applicable to only one of our case studies (Case B).

More broadly, the Search-Based Software Testing (SBST) community has long studied metaheuristic search for generating test data. Nevertheless, aside from the OCL- and Alloy-related work discussed above, the existing literature on search-based test data generation is oriented primarily around software code [1, 36, 52]. For example, Harman and McMinn [37] combine evolutionary (global) and Hill Climbing (local) search techniques for generating test data from Java programs while maximizing branch coverage. Using control-dependency graphs built from C code, Pargas et al. [60] devise a genetic algorithm for test case generation with the goal of maximizing statement and branch coverage. Panichella et al. [59] investigate multi- and many-objective optimization for building Java test suites that simultaneously maximize several coverage criteria, including statement, branch, and strong mutation coverage. All the above-cited approaches have some form of constraint solving embedded in them. The constraints targeted by these approaches are nevertheless at the source-code level, thus making them either inapplicable or unsuitable for system-level constraints, which are typically written in high-level languages based on first-order logic such as OCL.

While PLEDGE cannot be compared directly with approaches for code-level constraint solving, the hybridization idea pursued by PLEDGE is analogous to some existing work in that area. For example, Inkumsah et al. [40] combine search and Dynamic Symbolic Execution (DSE) for improving scalability in the structural testing of object-oriented systems. Here, search and DSE are combined in a sequential manner (first search followed by DSE), the assumption being that the constraints handled by each technique are independent from one another. In our context, and as we discussed in Section 6.1, one cannot always separate OCL subformulas in a way that search and SMT can be applied to them sequentially and yet effectively. This is why PLEDGE invokes search and SMT in an alternating and iterative manner, thus allowing the two techniques to work in tandem rather than sequentially.

There is work outside UML on specification-based data generation. A recent strand exploits a combination of exhaustive search and graph transformation for generating well-formed graph structures [72]. Here, the underlying constraints are defined through graph-logic expressions. While

this approach has the advantage of providing guarantees in terms of satisfiability, the underlying constraint language has limited expressiveness and covers only a small fragment of the OCL metamodel. Furthermore, the approach does not provide adequate support for integer, real, and string variables beyond treating them as enumerations. PLEDGE covers the entire OCL and does not suffer from the above limitations.

Another data generation approach from outside the UML community is TestBlox [79]. TestBlox aims to produce large, valid, and statistically representative data sets from multidimensional models to test the performance of big-data computing platforms. This approach admits a mix of hard (dimension or integrity) and soft (statistical) constraints. The approach relies on a customized solver that is able to generate, for certain types of constraints, test data in polynomial time with respect to the size of data instances [79]. This low computational complexity nonetheless comes at the expense of some strong restrictions. First, TestBlox requires that the generated data should be a DAG (Directed Acyclic Graph). Second, TestBlox is meant at solving only certain constraints, all of which have to do with the structural aspects of the generated data, e.g., cardinalities and the direction of links between objects. The result is that, in terms of expressiveness, TestBlox's declarative modeling language covers only a fragment of OCL.

Although PLEDGE cannot offer the same level of scalability as TestBlox, it is more practical to use in the context of system testing. PLEDGE does not restrict the structure of the instance models. In addition, it supports the entire OCL and thus a much larger spectrum of constraints, including constraints over primitive attributes. One can of course foresee hybrid approaches where TestBlox is employed as an alternative or complement to the search component of PLEDGE. However, for the reasons discussed above, TestBlox per se is not an alternative to PLEDGE for system-level test data generation purposes. For example, TestBlox cannot be applied to any of the three case studies used in Section 7 for evaluating PLEDGE. This is because these case studies all involve constraints, e.g., over attributes, that are not expressible in TestBlox.

In relation to soft (statistical) constraints supported by TestBlox, we note the following: We have developed in our previous work an approach for guiding OCL solving toward generating statistically representative data [77]. That approach works by continuously monitoring the data produced by an OCL solver and dynamically adding "corrective" soft constraints to align the properties of the data with a given set of statistical distributions. PLEDGE can be readily applied for generating statistically representative data. Evaluating PLEDGE in such a usage scenario is nevertheless outside the scope of this article. This said, we show in this article that PLEDGE outperforms the baseline solver from our previous work which was used for generating statistically representative data. We thus anticipate PLEDGE to also outperform the baseline when used in a similar manner.

10 CONCLUSION

We proposed a tool-supported approach based on UML for generating system test data under constraints. Our approach hybridizes metaheuristic search and SMT for solving complex constraints expressed in OCL. To our knowledge, we are the first to develop such a hybrid approach for constraint solving. We evaluated our approach on three industrial case studies, showing that the approach brings about important benefits in terms of applicability and scalability, when used on complex real-world problems.

Our current approach uses SMT only for solving constraints over primitive attributes. In the future, we would like to look into ways to exploit SMT more broadly, and thus further tap into the potential of background SMT theories. Another topic for future work has to do with how we delegate solving tasks to search and SMT. Currently, our delegation process is static. We plan to investigate whether a dynamic delegation strategy, e.g., using a machine-learning-based feedback loop, can provide advantages.

REFERENCES

- [1] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. 2010. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *IEEE Transactions on Software Engineering* 36, 6 (2010), 742–762.
- [2] Shaukat Ali, Muhammad Zohaib Z. Iqbal, Andrea Arcuri, and Lionel C. Briand. 2013. Generating Test Data from OCL with Search Techniques. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1376–1402.
- [3] Shaukat Ali, Muhammad Zohaib Z. Iqbal, Maham Khalid, and Andrea Arcuri. 2016. Improving the Performance of OCL Constraint Solving with Novel Heuristics for Logical Operations: A Search-based Approach. *Empirical Software Engineering* 21, 6 (2016), 2459–2502.
- [4] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing* (2nd ed.). Cambridge University Press.
- [5] Kyriakos Anastasakis. 2009. UML2Alloy (Version 0.52). <http://www.cs.bham.ac.uk/~bxb/UML2Alloy/>, last accessed: January 2019.
- [6] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. 2007. UML2Alloy: A Challenging Model Transformation. In *Proceedings of the 10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*. 436–450.
- [7] Apache Foundation. [n. d.]. Apache OpenJPA. <http://openjpa.apache.org/>. Last accessed: September 2019.
- [8] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. 171–177.
- [9] Clark W. Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. The University of Iowa. <http://smtlib.cs.uiowa.edu/language.shtml>, last accessed: January 2019.
- [10] Clark W. Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*. Springer, 305–343.
- [11] Daniel Le Berre and Anne Parrain. 2010. The SAT4J library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 2-3 (2010), 59–6.
- [12] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: automated testing based on Java predicates. In *Proceedings of the 11th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02)*. 123–133.
- [13] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17)*. 90–101.
- [14] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2018. SUSHI: a test generator for programs with complex structured inputs. In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE'18)*. 21–24.
- [15] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer* 7, 3 (2005), 212–232.
- [16] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2007. UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 547–548.
- [17] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2009. Verifying UML/OCL Operation Contracts. In *Proceedings of the 7th International Conference on Integrated Formal Methods (IFM'09)*. 40–55.
- [18] Jérôme Cantenot, Fabrice Ambert, and Fabrice Bouquet. 2014. Test generation with Satisfiability Modulo Theories solvers in model-based testing. *Software Testing, Verification and Reliability* 24, 7 (2014), 499–531.
- [19] Felix Chang, Emina Torlak, Julie Pelaez, and Daniel Jackson. 2018. Alloy Analyzer (Version 5). <http://alloytools.org/>, last accessed: January 2019.
- [20] Robert Clarisó, Carlos A. González, and Jordi Cabot. 2009. UMLtoCSP. <http://gres.uoc.edu/UMLtoCSP/>, last accessed: January 2019.
- [21] Robert Clarisó, Carlos A. González, and Jordi Cabot. In press. Smart Bound Selection for the Verification of UML/OCL Class Diagrams. *ACM Transactions on Software Engineering* (In press).
- [22] Coninfor Ltd. 2006. The ECLiPSe Constraint Programming System (Version 5.0). <http://eclipseclp.org/>, last accessed: January 2019.
- [23] Carolina Dania and Manuel Clavel. 2016. OCL2MSFOL: A mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints. In *Proceedings of the 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'16)*. 65–75.
- [24] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. Springer, 337–340.

- [25] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.
- [26] Leonardo Mendonça de Moura and Nikolaj Bjørner (Microsoft Research). 2012. Z3 Solver. <https://github.com/Z3Prover/z3>, last accessed: January 2019.
- [27] Rina Dechter and David Cohen. 2003. *Constraint processing*. Morgan Kaufmann.
- [28] Daniel Di Nardo, Fabrizio Pastore, and Lionel C. Briand. 2017. Augmenting Field Data for Testing Systems Subject to Incremental Requirements Changes. *ACM Transactions on Software Engineering and Methodology* 26, 1 (2017), 1–40.
- [29] Stefan Droste, Thomas Jansen, and Ingo Wegener. 2002. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science* 276, 1-2 (2002), 51–81.
- [30] Eclipse Foundation. [n. d.]. Aceleo - Transforming Models Into Code. <http://www.eclipse.org/aceleo/>. Last accessed: September 2019.
- [31] Eclipse Foundation. [n. d.]. ATL - A model transformation technology. <https://www.eclipse.org/atl/>. Last accessed: September 2019.
- [32] Eclipse Foundation. [n. d.]. Xpand. <https://www.eclipse.org/modeling/m2t/?project=xpand>. Last accessed: September 2019.
- [33] Eclipse Foundation. 2018. Eclipse OCL (Version 6.4.0). <http://www.eclipse.org/ocl/>, last accessed: January 2019.
- [34] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*. 502–518.
- [35] Carlos A. González and Jordi Cabot. 2014. Formal verification of static software models in MDE: A systematic review. *Information & Software Technology* 56, 8 (2014), 821–838.
- [36] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *Comput. Surveys* 45, 1 (2012), 11:1–11:61.
- [37] Mark Harman and Phil McMinn. 2010. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering* 36, 2 (2010), 226–247.
- [38] John Henry Holland et al. 1992. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.
- [39] Patrick Hurley. 2014. *A concise introduction to logic*. Nelson Education.
- [40] Kobi Inkumsah and Tao Xie. 2008. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. 297–306.
- [41] Muhammad Zohaib Z. Iqbal, Andrea Arcuri, and Lionel C. Briand. 2012. Combining Search-Based and Adaptive Random Testing Strategies for Environment Model-Based Testing of Real-Time Embedded Systems. In *Proceedings of the 4th International Symposium on Search Based Software Engineering (SBSSE'02)*. 136–151.
- [42] Muhammad Zohaib Z. Iqbal, Andrea Arcuri, and Lionel C. Briand. 2015. Environment modeling and simulation for automated testing of soft real-time embedded software. *Software & System Modeling* 14, 1 (2015), 483–524.
- [43] Stefan J. Galler and Bernhard K. Aichernig. 2014. Survey on test data generation tools. *International Journal on Software Tools for Technology Transfer* 16 (2014), 727–751.
- [44] Daniel Jackson. 2012. *Software Abstractions: Logic, language, and analysis*. Massachusetts Institute of Technology Press.
- [45] Guy Katz, Clark W. Barrett, Cesare Tinelli, Andrew Reynolds, and Liana Hadarean. 2016. Lazy proofs for DPLL(T)-based SMT solvers. In *Proceedings of the 2016 International Conference on Formal Methods in Computer-Aided Design (FMCAD'16)*. 93–100.
- [46] Joseph Kempka, Phil McMinn, and Dirk Sudholt. 2015. Design and analysis of different alternating variable searches for search-based software testing. *Theoretical Computer Science* 605 (2015), 1–20.
- [47] Marouane Kessentini, Houari A. Sahraoui, and Mounir Boukadoum. 2011. Example-based model-transformation testing. *Automated Software Engineering* 18, 2 (2011), 199–224.
- [48] Bogdan Korel. 1990. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering* 16, 8 (1990), 870–879.
- [49] Mirco Kuhlmann and Martin Gogolla. 2012. Strengthening SAT-Based Validation of UML/OCL Models by Representing Collections as Relations. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA'12)*. 32–48.
- [50] Bao-Lin Li, Zhi-Shu Li, Li Qing, and Yan-Hong Chen. 2007. Test Case Automate Generation from UML Sequence Diagram and OCL Expression. In *Proceedings of the 2007 International conference on Computational Intelligence and Security (CIS'07)*. 1048–1052.
- [51] Leonid Libkin. 2004. *Elements Of Finite Model Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer Verlag.

- [52] Phil McMinn. 2004. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.
- [53] Phil McMinn and Gregory M. Kapfhammer. 2016. AVMf: An Open-Source Framework and Implementation of the Alternating Variable Method. In *Proceedings of the 8th International Symposium on Search Based Software Engineering (SSBSE'16)*. 259–266.
- [54] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). Wiley Publishing.
- [55] Object Management Group. 2004. Object Constraint Language 2.4 Specification. <http://www.omg.org/spec/OCL/2.4/>, last accessed: January 2019.
- [56] Object Management Group. 2015. OMG Unified Modeling Language (UML). <http://www.omg.org/spec/UML/2.5>, last accessed: January 2019.
- [57] Xavier Oriol and Ernest Teniente. 2017. OCL_{UNIV}: Expressive UML/OCL Conceptual Schemas for Finite Reasoning. In *Proceedings of the 36th International Conference on Conceptual Modeling (ER'17)*. 354–369.
- [58] Xavier Oriol and Ernest Teniente. 2017. Simplification of UML/OCL schemas for efficient reasoning. *Journal of Systems and Software* 128 (2017), 130–149.
- [59] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158.
- [60] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. 1999. Test-data generation using genetic algorithms. *Software testing, verification and reliability* 9, 4 (1999), 263–282.
- [61] Nils Przigoda, Mathias Soeken, Robert Wille, and Rolf Drechsler. 2016. Verifying the structure and behavior in UML/OCL models using satisfiability solvers. *IET Cyber-Physical Systems: Theory & Applications* 1, 1 (2016), 49–59.
- [62] Nils Przigoda, Robert Wille, and Rolf Drechsler. 2016. Ground Setting Properties for an Efficient Translation of OCL in SMT-based Model Finding. In *Proceedings of the 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'16)*. 261–271.
- [63] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. 2012. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering* 73 (2012), 1–22.
- [64] Red Hat. [n. d.]. Hibernate. Everything data. <http://hibernate.org/>. Last accessed: September 2019.
- [65] Alan Robinson and Andrei Voronkov. 2001. *Handbook of Automated Reasoning*. Elsevier.
- [66] Nicolás Rosner, Juan P. Galeotti, Carlos López Pombo, and Marcelo F. Frias. 2010. ParAlloy: Towards a Framework for Efficient Parallel Analysis of Alloy Models. In *Proceedings of the 2nd Abstract State Machines, Alloy, B and Z (ABZ'10)*. 396–397.
- [67] Nicolás Rosner, Junaid Haroon Siddiqui, Nazareno Aguirre, Sarfraz Khurshid, and Marcelo F. Frias. 2013. Ranger: Parallel analysis of Alloy models by range partitioning. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. 147–157.
- [68] Per Runeson and Claes Wohlin. 1995. Statistical Usage Testing for Software Reliability Control. *Informatika* 19, 2 (1995), 195–207.
- [69] Stuart J. Russell and Peter Norvig. 1995. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc.
- [70] Hassan Sartaj, Muhammad Zohaib Iqbal, Atif Aftab Ahmed Jilani, and Muhammad Uzair Khan. 2019. A Search-Based Approach to Generate MC/DC Test Data for OCL Constraints. In *Proceedings of the 11th International Symposium on Search Based Software Engineering (SSBSE'19)*. 105–120.
- [71] Julia Seiter, Robert Wille, Mathias Soeken, and Rolf Drechsler. 2013. Determining relevant model elements for the verification of UML/OCL specifications. In *Design, Automation and Test in Europe (DATE'13)*. 1189–1192.
- [72] Oszkár Semeráth, Andraás Szabolcs Nagy, and Dániel Varró. 2018. A Graph Solver for the Automated Generation of Consistent Domain-Specific Models. In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE'18)*. 969–980.
- [73] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. 2009. Automatic Model Generation Strategies for Model Transformation Testing. In *Proceedings of the 2nd International Conference on Model Transformations (ICMT'09)*. 148–164.
- [74] Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. 2010. Verification-driven slicing of UML/OCL models. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*. 185–194.
- [75] Mathias Soeken, Robert Wille, and Rolf Drechsler. 2011. Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In *Proceedings of the 5th International Conference on Tests and Proofs (TAP'11)*. 152–170.
- [76] Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. 2017. Search-based Synthetic Data Generator. <http://people.svv.lu/tools/SDG/>, last accessed: January 2019.
- [77] Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. 2017. Synthetic Data Generation for Statistical Testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. 872–882.

- [78] Ting Su, Zhoulai Fu, Geguang Pu, Jifeng He, and Zhendong Su. 2015. Combining Symbolic Execution and Model Checking for Data Flow Testing. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE'15)*. 654–665.
- [79] Emina Torlak. 2012. Scalable test data generation from multidimensional models. In *Proceedings of the 20th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'12)*. 36.
- [80] Alexandre Torres, Renata Galante, Marcelo Soares Pimenta, and Alexandre Jonatan B. Martins. 2017. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information & Software Technology* 82 (2017), 1–18.
- [81] Mark Utting and Bruno Legeard. 2010. *Practical model-based testing: A tools approach*. Elsevier.
- [82] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel C. Briand, and Muhammad Zohaib Z. Iqbal. 2015. Automatic generation of system test cases from use case specifications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'15)*. 385–396.
- [83] Jianghao Wang, Hamid Bagheri, and Myra B. Cohen. 2018. An evolutionary approach for analyzing Alloy specifications. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*. 820–825.
- [84] Jos B Warmer and Anneke G Kleppe. 2003. *The Object Constraint Language: Getting your models ready for MDA*. Addison-Wesley Professional.
- [85] W. Eric Wong. 2001. *Mutation Testing for the New Century*. Kluwer Academic Publishers, Norwell, MA, USA.
- [86] Pingyu Zhang, Sebastian G. Elbaum, and Matthew B. Dwyer. 2011. Automatic generation of load tests. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*. 43–52.

Appendix

A RULES FOR DELEGATING SOLVING TASKS TO SEARCH AND SMT

Table 4 provides our rule set for labeling the AST nodes of OCL expressions. The rules cover *all* the AST node types in the OCL 2.4 metamodel (<https://www.omg.org/spec/OCL/2.4/>). Fig. 14 depicts a simplified version of this metamodel, with only the inheritance (and not the composition) relationships shown. In the figure, abstract AST node types are shaded grey. Given an AST (for an NNF constraint), our labeling procedure visits all the nodes in the AST and assigns labels to all the visited nodes except those representing constants. Our comprehensive labeling of the AST ensures that we always come down with a decision about whether to delegate a specific solving task to search, SMT, or both. Whether the decisions made by our labeling procedure lead to an effective solving of the constraints at hand is a separate question, which we investigated empirically in Section 7.

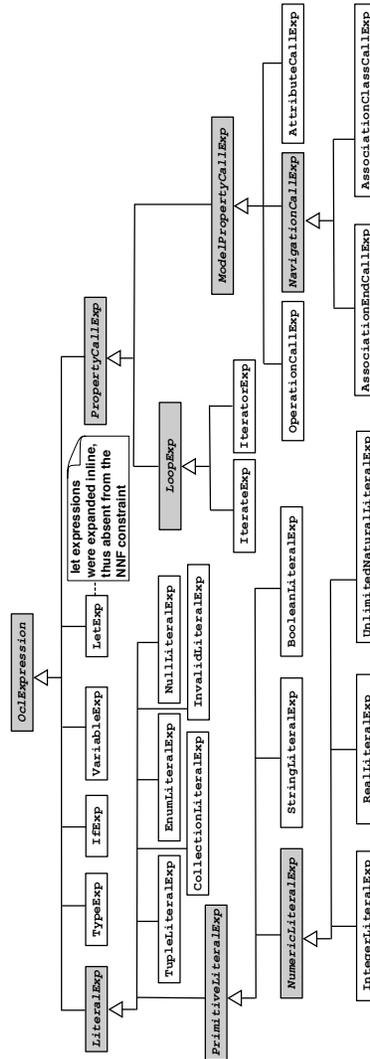


Fig. 14. Simplified View of the OCL 2.4 Metamodel

Each row in Table 4 defines one labeling rule. The first column of the table indicates the node type (from the metamodel of Fig. 14). When the different subtypes of the node type in the first column require different treatments (i.e., need to be assigned different labels), the second column enumerates the subtypes separately. The fourth column shows the label to attach to a node type (or subtype) if the condition specified in the third column holds. The rationale behind the label attached to a given node (sub)type is provided alongside the condition in the third row. We note that, when a given node type has multiple conditions, C_1, \dots, C_n , an if-then-else semantics is intended, i.e., **if** (C_1) **then** $label_1$ **else if** (C_2) **then** $label_2 \dots$ **else if** (C_n) **then** $label_n$. Here, the last condition, C_n , is always the default (catch-all) condition.

Table 4. Complete List of Rules for Labeling OCL AST Nodes (the Subtypes column is filled only when the labeling is not uniform across subtypes)

Node type	Subtypes	Condition & Rationale	Label
PrimitiveLiteralExp 's subtypes, EnumLiteralExp	-	Condition: - Rationale: The targeted nodes represent constants such as numeric and string literals. Constants are left unlabelled because they per se do not induce any solving tasks.	-
InvalidLiteralExp , TypeExp	-	Condition: - Rationale: SMT does not support OCL's invalid and type expressions. Such expressions, which are mostly used to constrain the structure of the instance model, are always assigned to search.	<i>search</i>
LoopExp 's subtypes	-	Condition: - Rationale: As discussed in Section 3, SMT is not our preferred alternative for handling OCL's loop expressions, including quantifiers. We therefore delegate all loop expressions to search.	<i>search</i>
CollectionLiteralExp , TupleLiteralExp	-	Condition: - Rationale: Similar to loop expressions, collections and tuple are not readily matched to efficient SMT decision procedure (see Section 4). Collections and tuple expressions are thus always assigned to search.	<i>search</i>
NavigationCallExp 's subtypes	-	Condition: - Rationale: Evaluating a navigation expression yields one of the following: an object, a collection, or an invalid / null. None of these return types are primitive. As discussed in Section 4, SMT's involvement is narrowed to subformulas containing primitive types only. We thus always assign navigations to search.	<i>search</i>

VariableExp, AttributeCallExp	-	<p>Condition 1: If the attribute / variable is non-primitive. Rationale: When the accessed attribute / variable represents an object, a collection, or an invalid / null value, the node is labeled search. The rationale is the same as that for the NavigationCallExp subtypes in the previous row of the table.</p>	<i>search</i>
		<p>Condition 2: If the attribute / variable is primitive, but is contained directly or indirectly within the body of an exists, select, reject, any, isUnique, or one. Rationale: In principle, nodes representing primitive attributes or variables should be delegated to SMT. This rule is an exception to the principle, allowing both SMT and search to alter the value of certain attributes / variables. For example, consider the subformula c->select(a = 3)->notEmpty(). Although a is a primitive attribute, search still needs visibility into it. This is because a appears within a select, which is to be handled by search. If search is barred from manipulating a, then search would be extremely unlikely to be able to evolve the structure of the instance model to a point where SMT can produce a satisfying assignment. The same explanation applies to exists, reject, any, isUnique, and one, all of which are to be handled by search.</p>	<i>both</i>
		<p>Condition 3: If neither Condition 1 nor Condition 2 holds. Rationale: Here, we are in the situation where the node in question represents an attribute / variable meeting both of the following criteria: (1) the node is primitive, and (2) the node is not contained in exists, select, reject, any, isUnique, or one. In this situation, SMT can find a satisfying assignment independently of search. We thus assign the node to SMT.</p>	<i>SMT</i>
IfExp	-	<p>Condition 1: If the body of the if-then-else has a non-primitive</p>	<i>search</i>

		<p>return type. Rationale: Same rationale as that given for Condition 1 of the VariableExp and AttributeCallExp types.</p>	
		<p>Condition 2: If the the body of the if-then-else has a primitive return type, but is directly or indirectly contained within an exists, select, reject, any, isUnique, or one. Rationale: Same rationale as that given for Condition 2 of the VariableExp and AttributeCallExp types.</p>	<i>both</i>
		<p>Condition 3: If neither Condition 1 nor Condition 2 holds. Rationale: The rationale is the same as that for Condition 3 of the VariableExp and AttributeCallExp types.</p>	<i>SMT</i>
OperationCallExp	<p>and, or, not, <, >, <=, >=, =, <>, +, -, *, /</p>	<p>Condition 1: If all children are unlabeled. Rationale: The labels of these operation nodes are deduced from those of their children. This particular condition covers the situation where all the children are unlabelled. For example, the node representing or in the following subformula is left unlabeled: false or true. This is because the children (false and true) are constants and unlabeled. Neither search nor SMT should attempt to solve an unlabeled subformula.</p>	-
		<p>Condition 2: If all labeled children have <i>search</i> as label. Rationale: Since all the (labeled) children are delegated to search, intervention from SMT is unwarranted; the whole subformula is therefore to be handled by search.</p>	<i>search</i>
		<p>Condition 3: If all labeled children have <i>SMT</i> as label. Rationale: Since all the (labeled) children are delegated to SMT, intervention from search is unwarranted; the whole subformula is therefore to be handled by SMT.</p>	<i>SMT</i>
		<p>Condition 4: If all labeled children have <i>both</i> as label.</p>	<i>both</i>

	<p>Rationale: Since all the (labeled) children require both search and SMT, so does the parent node.</p>	
	<p>Condition 5: If none of the above four conditions hold.</p> <p>Rationale: Here, the (labeled) children include at least two (and potentially all three) possible labels: <i>search</i>, <i>SMT</i>, and <i>both</i>. This means that, just like in Condition 4, the intervention of both search and SMT is warranted to solve the underlying subformula. As a result, the label assigned to the node is <i>both</i>.</p>	<i>both</i>
size, max, min	<p>Condition 1: If the operation is called over a collection.</p> <p>Rationale: The operations in this sub-type can be invoked over both collections and primitive attributes. For example, one can use the size operation over either a string attribute or over a set. This rule thus needs to first check the type of the element to which the operation is applied. If the calling element is a collection, then the whole subformula is delegated to search. The rationale is the same as that for CollectionLiteralExp and TupleLiteralExp, presented earlier.</p>	<i>search</i>
	<p>Condition 2: If the operation is called over a primitive type, and at the same time, the operation is directly or indirectly contained within the body of an exists, select, reject, any, isUnique, or one.</p> <p>Rationale: Same rationale as that given for Condition 2 of the VariableExp and AttributeCallExp types.</p>	<i>both</i>
	<p>Condition 3: If neither Condition 1 nor Condition 2 holds.</p> <p>Rationale: This rule covers the situation where: (1) the node represents an operation that is applied over a primitive attribute / variable, and (2) the underlying subformula can be solved independently of search. We therefore label the node as <i>SMT</i>.</p>	<i>SMT</i>

<p><code>concat, substring, toInteger, toReal, round, ceil, abs, floor, div, mod</code></p>	<p>Condition 1: If the operation in question is directly or indirectly contained within the body of an exists, select, reject, any, isUnique, or one. Rationale: All the operations here are applicable to primitive attributes / variables only. The rationale is the same as that for Condition 2 of the VariableExp and AttributeCallExp types.</p> <p>Condition 2: If Condition 1 does not hold. Rationale: The rationale is the same as that for Condition 2 of the VariableExp and AttributeCallExp types.</p>	<p><i>both</i></p> <p><i>SMT</i></p>
<p><code>toLower, sum, toUpper, union, allInstances, oclIsInvalid, oclIsKindOf, oclIsTypeOf, oclIsUndefined, oclAsType, any, asBag, collect, asOrderedSet, asSequence, asSet, collectNested, count, excludes, excludesAll, excluding, flatten, includes, includesAll, including, isEmpty, isUnique, notEmpty, one, product, reject, select, sortedBy, append, at, first, last, indexOf, insertAt, prepend, subsequence, intersection, subOrderedSet, symmetricDifference</code></p>	<p>Condition: - Rationale: The operations in this subtype are all labeled <i>search</i>. The reason is that each of these operations falls under one of the following two categories for which SMT is not an option: (1) the operation is invoked over objects, collections, and other non-primitive data types, or (2) the operation applies to primitive types, but does not have suitable counterpart functions in SMT-LIB as discussed in Section 6.4.</p>	<p><i>search</i></p>
<p><code>user-defined</code></p>	<p>Condition 1: If the operation is both non-recursive and further has a primitive return type. Rationale: The rationale is the same as that for Condition 3 of the VariableExp and AttributeCallExp types. We note that this rule excludes recursive user-defined operations from being handled by SMT, even when such operations have primitive return types. This is because we</p>	<p><i>SMT</i></p>

	found Z3 (the SMT solver used by PLEDGE) to be inefficient over recursive functions.	
	<p>Condition 2: If Condition 1 does not hold.</p> <p>Rationale: All remaining types of user-defined operations, i.e., recursive ones and those with non-primitive return types, are assigned to search. This is because: (1) as explained in Condition 1 above, SMT is not ideal for recursive functions, and (2) following the same rationale as that for Condition 1 of the VariableExp and AttributeCallExp types, we want only search to handle non-primitive constructs.</p>	<i>search</i>

B EXPANDING OCL'S BUILT-IN QUANTIFICATION SHORTCUTS DURING THE CONSTRUCTION OF AN SMT-LIB FORMULA

Table 5 lists all the OCL constructs that involve implicit quantification and thus need to be expanded over a given instance model when an SMT-LIB formula is being built. The first column of the table presents the OCL construct to be expanded. The second column describes how the expansion is performed when the construct is applied to non-empty collections. The third column provides the literal that replaces the construct (1) when the construct is called over an empty collection, or (2) in cases where the construct compares two collections, when at least one collection is empty.

Table 5. Full List of OCL Constructs with Implicit Quantification

OCL construct	Description of the expansion process	Expansion involving empty collections
excludes	Expanding $c \rightarrow \text{excludes}(x)$, where c is a collection and x is a given element, is equivalent to expanding: $(\text{not } c \rightarrow \text{exists}(i \mid i = x))$.	true
excludesAll	Expanding $c1 \rightarrow \text{excludesAll}(c2)$, where $c1$ and $c2$ are two collections, is equivalent to expanding: $(c2 \rightarrow \text{forAll}(i \mid \text{not } c1 \rightarrow \text{exists}(j \mid i = j)))$.	true
includes	Expanding $c \rightarrow \text{includes}(x)$, where c is a collection and x is a given element, is equivalent to expanding: $(c \rightarrow \text{exists}(i \mid i = x))$.	false
includesAll	Expanding $c1 \rightarrow \text{includesAll}(c2)$, where $c1$ and $c2$ are two collections, is equivalent to expanding: $(c2 \rightarrow \text{forAll}(j \mid c1 \rightarrow \text{exists}(i \mid i = j)))$.	false
isUnique	Expanding $c \rightarrow \text{isUnique}(\langle \text{condition over } c \rangle)$, where c is a collection, is equivalent to expanding: $(c \rightarrow \text{forAll}(i \mid c \rightarrow \text{forAll}(j \mid i = j \text{ or } \langle \text{condition over } c_i \rangle \Leftrightarrow \langle \text{condition over } c_j \rangle)))$.	true

one	Expanding $c \rightarrow \text{one}(\langle \text{condition over } c \rangle)$, where c is a collection, is equivalent to expanding: $c \rightarrow \text{exists}(i \langle \text{condition over } c_i \rangle)$ and $c \rightarrow \text{forall}(j i \triangleleft j \text{ and not } \langle \text{condition over } c_j \rangle)$.	false
= (for sequences)	Expanding $s1 = s2$, where $s1$ and $s2$ are two sequences, is equivalent to expanding: $(s1 \rightarrow \text{size}() = s2 \rightarrow \text{size}() \text{ and } s1 \rightarrow \text{asSet}() \rightarrow \text{asSequence}() \rightarrow \text{forall}(i s2 \rightarrow \text{asSet}() \rightarrow \text{asSequence}() \rightarrow \text{at}(s1 \rightarrow \text{asSet}() \rightarrow \text{asSequence}() \rightarrow \text{indexOf}(i)) = i))$.	If both collections are empty then true, otherwise false
\triangleleft (for sequences)	Expanding $s1 \triangleleft s2$, where $s1$ and $s2$ are two sequences, is equivalent to expanding: $(s1 \rightarrow \text{size}() \triangleleft s2 \rightarrow \text{size}() \text{ or } s1 \rightarrow \text{asSet}() \rightarrow \text{asSequence}() \rightarrow \text{exists}(i s2 \rightarrow \text{asSet}() \rightarrow \text{asSequence}() \rightarrow \text{at}(s1 \rightarrow \text{asSet}() \rightarrow \text{asSequence}() \rightarrow \text{indexOf}(i)) \triangleleft i))$.	If both collections are empty then false, otherwise true
= (for bags)	Expanding $b1 = b2$, where $b1$ and $b2$ are two bags, is equivalent to expanding: $(b1 \rightarrow \text{forall}(i b2 \rightarrow \text{exists}(j i = j \text{ and } b1 \rightarrow \text{count}(i) = b2 \rightarrow \text{count}(j))))$.	If both collections are empty then true, otherwise false
\triangleleft (for bags)	Expanding $b1 \triangleleft b2$, where $b1$ and $b2$ are two bags, is equivalent to expanding: $(b1 \rightarrow \text{exists}(i b2 \rightarrow \text{forall}(j i \triangleleft j \text{ or } b1 \rightarrow \text{count}(i) \triangleleft b2 \rightarrow \text{count}(j))))$.	If both collection are empty then false, otherwise true
= (for sets)	Expanding $s1 = s2$, where $s1$ and $s2$ are two sets, is equivalent to expanding: $(s1 \rightarrow \text{forall}(i s2 \rightarrow \text{exists}(j i = j)) \text{ and } s2 \rightarrow \text{forall}(i s1 \rightarrow \text{exists}(j i = j)))$	If both collections are empty then true, otherwise false
\triangleleft (for sets)	Expanding $s1 \triangleleft s2$, where $s1$ and $s2$ are two sets, is equivalent to expanding: $(s1 \rightarrow \text{exists}(i s2 \rightarrow \text{forall}(j i \triangleleft j)) \text{ or } s2 \rightarrow \text{exists}(i s1 \rightarrow \text{forall}(j i \triangleleft j)))$.	If both collection are empty then false, otherwise true
= (for ordered sets)	Expanding $os1 = os2$, where $os1$ and $os2$ are two ordered sets, is equivalent to expanding: $(os1 \rightarrow \text{size}() = os2 \rightarrow \text{size}() \text{ and } os1 \rightarrow \text{forall}(i os2 \rightarrow \text{exists}(j i = j)))$.	If both collections are empty then true, otherwise false
\triangleleft (for ordered sets)	Expanding $os1 \triangleleft os2$, where $os1$ and $os2$ are two ordered sets, is equivalent to expanding: $(os1 \rightarrow \text{size}() \triangleleft os2 \rightarrow \text{size}() \text{ or } os1 \rightarrow \text{exists}(i os2 \rightarrow \text{forall}(j i \triangleleft j)))$.	If both collection are empty then false, otherwise true
= (between an ordered Set and a set)	Expanding $os = s$, where os is an ordered set and s is a set, is equivalent to expanding: $(os \rightarrow \text{asSet}() \rightarrow \text{forall}(i s \rightarrow \text{exists}(j i = j)) \text{ and } s \rightarrow \text{forall}(i os \rightarrow \text{asSet}() \rightarrow \text{exists}(j i = j)))$	If both collections are empty then true, otherwise false

$\langle \rangle$ (between an ordered set and a set)	Expanding $os \langle \rangle s$, where os is an ordered set and s is a set, is equivalent to expanding: $(os \rightarrow asSet() \rightarrow exists(i \mid s \rightarrow forAll(j \mid i \langle \rangle j)) \text{ or } s \rightarrow exists(i \mid os \rightarrow asSet \rightarrow forAll(j \mid i \langle \rangle j))$	If both collection are empty then false , otherwise true
--	--	--

C RULES FOR TRANSLATING OCL TO SMT-LIB

Table 6 provides the rules that we use for translating NNF OCL expressions (after the execution of the expansion and substitution processes) to SMT-LIB. The first column of the table indicates the type of the traversed node (from the metamodel of Fig. 14). The second column shows the general shape of the OCL expression represented by the node in the first column. The third column shows the SMT-LIB fragment resulting from the translation.

Table 6. OCL to SMT-LIB Translation Rules

Node type possible within a processed NNF expression	Shape of the OCL expression	Corresponding SMT-LIB formula
PrimitiveLiteralExp 's subtypes, EnumLiteralExp	$\langle A \text{ constant literal } X \rangle$	X
IfExp	if ($\langle \text{condition} \rangle$) then $\langle \text{body1} \rangle$ else $\langle \text{body2} \rangle$ endif	(ite toSMT-LIB ($\langle \text{condition} \rangle$) toSMT-LIB ($\langle \text{body1} \rangle$) toSMT-LIB ($\langle \text{body2} \rangle$))
VariableExp	$\langle A \text{ variable named } X \rangle$	The name of the SMT-LIB variable that corresponds to X
AttributeCallExp	$\langle An \text{ access to an attribute named } X \rangle$	The name of the SMT-LIB variable that corresponds to X
OperationCallExp	$\langle \text{operand1} \rangle \langle \text{op} \rangle \langle \text{operand2} \rangle$; Binary infix operations where $\langle \text{op} \rangle$ can be: and , or , < , > , <= , >= , + , - , * , / , = , or $\langle \rangle$	($\langle \text{op} \rangle$ toSMT-LIB($\langle \text{operand1} \rangle$) toSMT-LIB($\langle \text{operand2} \rangle$))
	not ($\langle \text{expression} \rangle$)	(not toSMT-LIB($\langle \text{expression} \rangle$))

	<p>$\langle X \rangle . \langle \text{op} \rangle ()$; Calls to OCL built-in operations that admit no parameters. Specifically, $\langle \text{op} \rangle$ can be: size, toInteger, toReal, round, ceil, abs, or floor</p>	<p>$\langle \langle \text{op} \rangle \text{ toSMT-LIB}(\langle X \rangle) \rangle$</p> <p>Except for the abs operation, which has a corresponding built-in function in SMT-LIB with the same name, the remaining operations need to be explicitly defined in SMT-LIB as illustrated below:</p> <ol style="list-style-type: none"> (1) <code>(define-fun sizeString ((a String)) Int (str.len a))</code> (2) <code>(define-fun toInteger ((a String)) Int (str.to.int a))</code> (3) <code>(define-fun round ((a Real)) Real (ite (> (- a (floor a)) 0.5) (toReal (ceil a)) (toReal (floor a))))</code> (4) <code>(define-fun ceil ((x Real)) Int (+ (to_int x) 1))</code> (5) <code>(define-fun floor ((x Real)) Int (to_int x))</code> (6) <code>(define-fun toReal ((a Int)) Real (* a 1.0))</code>
	<p>$\langle X \rangle . \langle \text{op} \rangle (\langle Y \rangle)$; Calls to OCL built-in operations that admit one parameter. Specifically, $\langle \text{op} \rangle$ can be: max, min, div, mod, or concat</p>	<p>$\langle \langle \text{op} \rangle \text{ toSMT-LIB}(\langle X \rangle) \text{ toSMT-LIB}(\langle Y \rangle) \rangle$</p> <p>Except for div and mod, which have corresponding built-in functions in SMT-LIB (with the same names), the remaining operations need to be explicitly defined as illustrated below:</p> <ol style="list-style-type: none"> 7) <code>(define-fun max ((a Real)(b Real)) Real (ite (>= a b) a b))</code> 8) <code>(define-fun min ((a Real)(b Real)) Real (ite (<= a b) a b))</code> 9) <code>(define-fun concatString ((a String)(b String)) String (str.++ a b))</code>
	<p>$\langle X \rangle . \text{substr}(\langle Y \rangle, \langle Z \rangle)$ (this is the only OCL built-in operation on primitive variables that has two or more parameters)</p>	<p>$\langle \text{str.substr} \text{ toSMT-LIB}(\langle X \rangle) \text{ toSMT-LIB}(\langle Y \rangle) \text{ toSMT-LIB}(\langle Z \rangle) \rangle$</p>
	<p>$\langle X \rangle . \langle \text{user-defined operation } O \rangle (\langle \mathcal{P} \rangle \text{ parameters})$</p>	<p>$\langle \langle O \rangle \langle \text{forEach } (\mathcal{P}_i \in \mathcal{P}) \text{ do } \{ \text{toSMT-LIB}(\mathcal{P}_i) \} \rangle \rangle$</p>