Incremental Verification of UML/OCL Models

Robert Clarisó[*], **Carlos A. González**[†], **and Jordi Cabot**[‡]
[*‡]Universitat Oberta de Catalunya (UOC), Spain
[†]Grantecan S.A., Spain
[‡]ICREA, Spain

ABSTRACT Model-Driven Development employs models as core artifacts of the software development process. This requires ensuring the correctness of models, an analysis which is computationally complex. However, models may evolve over time and these changes usually require re-checking models from scratch. To this end, this paper proposes techniques for the *incremental verification* of a fundamental correctness property: internal consistency of UML class diagrams annotated with OCL constraints. These techniques allow modelers to significantly reduce (or even avoid altogether) the cost of re-verifying a class diagram after model updates.

KEYWORDS UML, OCL, Formal Verification, Model Evolution, Model Certificate, Scalability

1. Introduction

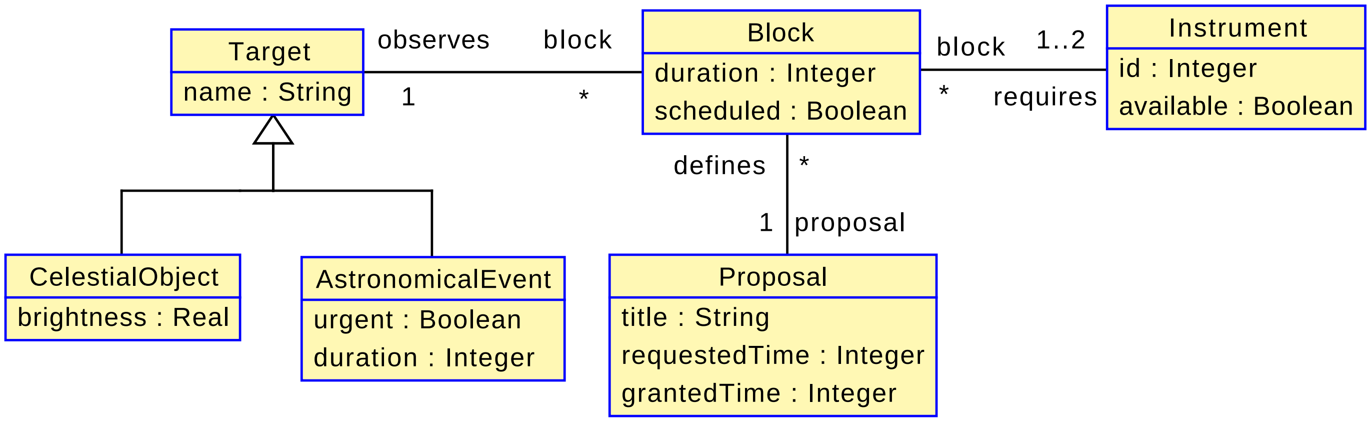
Model-Driven Development is a software development paradigm where models play a central part. In this paradigm, models are used beyond documentation or communication purposes: some elements of the implementation may be generated (semi)automatically from models. Thus, the correctness of models is critical as any error in the model may translate into errors in the final implementation, where it will be much harder to detect, debug and fix.

There are many types of software models, and one the most frequently used is the *UML class diagram* (Petre 2013), which describes the static structure of a system in terms of classes and their relationships. Several correctness properties for this type of diagram can be defined. A fundamental correctness property is called *strong satisfiability* (Rull et al. 2015; González & Cabot 2014): a lack of contradictions in the diagram allowing us to construct a finite instance with a non-empty population for each class and association that satisfies all the constraints, *e.g.*, inheritance relationships and multiplicities of associations. Checking satisfiability is computationally expensive, and it becomes undecidable if the model is annotated with complex integrity constraints written in the Object Constraint Language (OCL) (Cabot & Gogolla 2012).

As an example, we present the running example we will use throughout this paper: the UML/OCL model depicted in Figure 1. This class diagram describes the management of observing requests in a telescope facility. Researchers submit proposals to observe a particular event or celestial object during a time frame, specified as a series of observing blocks. Astronomers at the telescope use observing instruments (*e.g.*, spectrographs) to run the observing blocks and gather data about the requested target. Several integrity constraints establish limits on the duration of an observing proposal, observing blocks and the identifiers of observing instruments. This model is satisfiable: it is possible to populate all classes in this model while satisfying all graphical constraints in the class diagram and all textual invariants simultaneously. For instance, Figure 2 shows a valid object diagram describing the objects (with their type and attribute values) and the links connecting them.

Several tools support this verification using different types of solvers and reasoning tools. We will refer to them using the generic term *model finders*. Nevertheless, models may change during the development of a software system, for instance, due to changes in the domain or the business rules. A problem of model verification tools for satisfiability is that they do not support *incremental verification*, *i.e.* the ability to re-use results from previous verifications to facilitate the analysis of similar models. Thus, time-consuming verification runs should be

JOT reference format:
Robert Clarisó, Carlos A. González, and Jordi Cabot. *Incremental Verification of UML/OCL Models.* Journal of Object Technology. Vol. 19, No. 3, 2020. Licensed under Attribution - NonCommercial - No Derivatives 4.0 International (CC BY-NC-ND 4.0) http://dx.doi.org/10.5381/jot.2020.19.3.a7



```
-- No duplicate instrument ids
context Instrument inv UniqueInstrument:
  Instrument.allInstances()->forAll(i| (self.id = i.id) implies (self = i))

-- Time limits on observing proposals
contest Proposal inv TimeLimits:
  (self.requestedTime > 0) and (self.grantedTime >= 0) and
  (self.grantedTime <= self.requestedTime)

-- Observing blocks must be longer than the events they observe
context AstronomicalEvent inv BlockDuration:
  self.block->forAll( b | (b.duration >= self.duration) )
```

Figure 1 Running example: (top) A UML class diagram model describing the management of observing requests in a telescope facility; (bottom) additional integrity constraints, described as OCL invariants.

repeated after each model change, a problem that impairs its practical application in an industrial setting.

In this paper, we propose techniques for the incremental verification of satisfiability in UML/OCL class diagrams. These techniques combine known methods for speeding verification, such as *model slicing* (Shaikh, Clarisó, et al. 2010), with a novel approach: the use of instances of the model as *certificates* of satisfiability. Whenever a model is checked for satisfiability, a valid instance of the model is provided as an output of the model finder. After an update in the model, rather than re-verifying the model it may be sufficient to adjust the original valid instance as an instance of the new model to certificate that the model is still satisfiable.

For instance, if a new OCL invariant is added to the model, it is sufficient to check if our certificate satisfies the new invariant: if it does, our old certificate is still valid. Checking an OCL invariant on a given instance is orders of magnitude faster than invoking a model finder. Moreover, if the certificate is not valid and a model finder needs to be invoked, it may be possible to limit the verification to a subset (a *slice*) of the original model.

Paper organization. The remainder of this paper is organized as follows. Section 2 presents related work and the relationship of this paper with other problems like model or database repair. Section 3 describes the method. Then, Section 4 describes the tool support for the method. Finally, Section 5 concludes and presents future work.

2. Related work

In this section, we discuss different types of works that are related to the incremental verification of UML/OCL models.

2.1. Verifying declarative specifications

Many different paradigms have been considered for the verification of UML/OCL models (Gabmeyer et al. 2019; González & Cabot 2014): SAT solvers (Soeken et al. 2010; Kuhlmann et al. 2011), SMT solvers (Clavel et al. 2010; Dania & Clavel 2016; Wu 2017; Soltana et al. 2020), constraint programming (Cabot et al. 2014; E. K. Jackson et al. 2011), theorem proving (Brucker & Wolff 2008), query containment (Rull et al. 2015), term rewriting (Romero et al. 2007), graph solvers (Semeráth et al. 2018) and search-based methods (Soltana et al. 2020). On the other hand, Alloy (D. Jackson 2012) provides both a textual notation and an analyzer that can verify models using relational logic and SAT solvers (Torlak & Jackson 2007). Nevertheless, the underlying verification problem for these declarative notations is undecidable. Therefore, verification techniques exhibit a combinatorial explosion which hampers their scalability and limits their applicability in large and complex models.

Several strategies have been considered to improve the efficiency of the verification, both in the context of UML/OCL and Alloy:

– *Parallelization* (Rosner et al. 2013): splitting the search

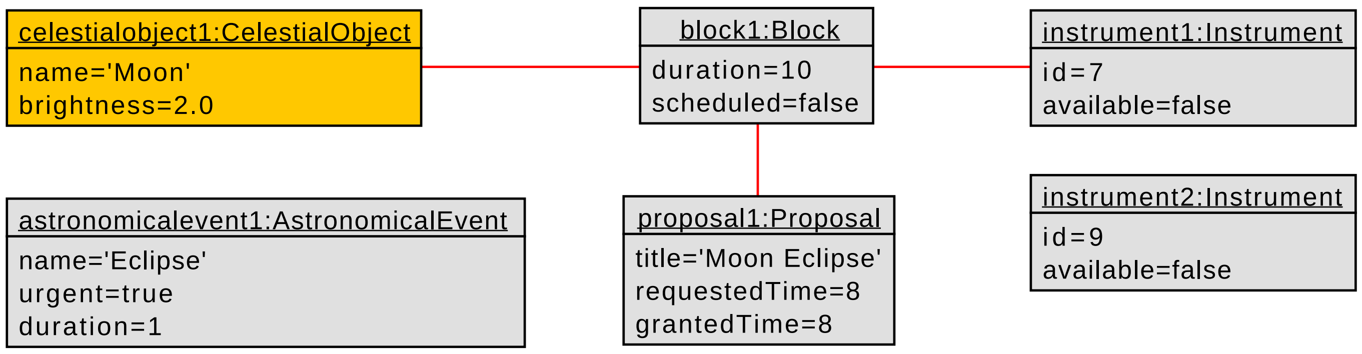


Figure 2 Object diagram with a valid instance for the running example in Figure 1.

space into smaller domains that are explored in parallel by different threads of execution.

- *Model slicing* (Shaikh, Clarisó, et al. 2010; Sen et al. 2009; Uzuncaova & Khurshid 2008): dividing the original models into submodels that abstract unnecessary details from the original model and can be verified separately.
- *Domain-specific solvers* (Ganov et al. 2012): These solvers can analyze parts of the model in parallel, speeding up the verification.
- *Bound reduction* (Clarisó et al. 2017): A quick pre-analysis of the model can discard parts of the search space that do not include any potential solution, speeding up the analysis.
- *Limiting the expressiveness* (Queralt et al. 2012; Oriol & Teniente 2017; Balaban & Maraee 2013): Reducing the set of constructs that can be used in defining integrity constraints can make the problem decidable and/or provide efficient algorithms for finding solutions.

In this paper we consider *incremental verification*: given a verified model and a change in the model, we reduce the amount of work required to check the new model by reusing information about the verification of the original model.

2.2. Conflict resolution approaches

Evolving a model (Khalil & Dingel 2013) often implies dealing with potential evolution conflicts, especially in a collaborative modeling scenario (Izquierdo & Cabot 2016).

While these papers mainly address the challenge of conflict resolution (*i.e.*, how to reconcile the different versions of a model created in parallel by different designers evolving and refactoring the original model), they all need to represent the sequence of model changes to reason on it. In this sense, the conflict (meta)models they propose (see, for instance, (Brosch et al. 2012)) could be useful in the context of this work as an internal representation for the reasoning tool.

To optimize the reasoning process, the above works also typically include a taxonomy of potential changes in a model, *e.g.*, (Altmanninger et al. 2009). These taxonomies have inspired our own list of model changes in Section 3.2.

2.3. Incremental analysis

Different types of incremental analysis have been proposed in the context of declarative models and integrity constraints. We discuss their relationship with incremental verification.

Several previous works have considered the *incremental evaluation* of integrity constraints, *e.g.*, (Cabot & Teniente 2009; Oriol & Teniente 2015): given an instantiation of the UML class diagram where the OCL integrity constraints are satisfied and a change (create/update/delete) in that instantiation, propose efficient mechanisms to reevaluate the contraints in the modified instantiation. Notice that, in contrast to incremental evaluation, incremental verification analyzes the model ("is there any instantiation that satisfies the constraint?") rather than a specific instantiation ("is the constraint satisfied in a given instance?") and also deals with changes at the UML/OCL model (modifying a class) rather than changes in its instantiation (creating a new object).

The execution of model transformations may also exhibit scalability problems for large models. Hence, several works have considered the incremental execution of model transformations (Jouault & Beaudoux 2016; Varró et al. 2016). Transformation rules usually have guards or application conditions, expressed as queries over models in a similar way to integrity constraints. Therefore, incremental model transformation is closely related to the incremental evaluation of integrity constraints.

Finally, some previous works like (Egyed 2007; Blanc et al. 2009; Reder & Egyed 2012) have used incrementality while checking the *well-formedness* of UML models *e.g.* all methods used in a sequence diagram are declared in the corresponding class diagram. Instead, in this paper we focus on the *intra-model consistency* for UML class diagrams, considering the satisfiability of OCL integrity constraints rather than well-formedness rules.

2.4. Incremental verification

For instance, in (Semeráth et al. 2016) verification is incremental in the sense that the analysis of a declarative model is divided into a sequence of steps to improve scalability and guide the search towards more realistic solutions. Nevertheless, few approaches have considered the incremental verification of a declarative model after a set of changes in the model. In this section, we describe three of the closest ones to our approach, which are defined in the context of Alloy: Titanium (Bagheri & Malek 2016), iAlloy (Wang et al. 2019) and Platinum (Zheng et al. 2020).

Titanium (Bagheri & Malek 2016) is a tool for checking of evolving Alloy specifications. The goal of Titanium is using information about the verification of the original model to tighten the lower and upper bound for the shared relational values, without losing any potential solution. Using these reduced bounds improves the efficiency of the analysis for the modified model. Titanium supports a rich set of edit operations, including updates in the definitions of signatures, but does not avoid re-running the verification (it only improves its efficiency) and is specific to the Alloy notation and solver.

iAlloy (Wang et al. 2019) is a tool for the incremental verification of Alloy models. This tool performs static analysis to detect changes in predicates and dependencies between them, and records the value of intermediate predicates to reuse them if the current predicate and the predicates it depends on have not been modified. Nevertheless, iAlloy (a) does not support updates in the definition of signatures and (b) only reuses solutions (it is not able to *update* them to avoid recomputation).

Platinum (Zheng et al. 2020) is another extension of the Alloy Analyzer to support evolving specifications. It allows reusing checks performed in previous versions of the specification (whole or parts). Nevertheless, it again does not allow *updating* previous results to avoid recomputation.

3. Method

Verifying the satisfiability of a UML/OCL class diagram is a computationally complex task. When the model is satisfiable, the result of the verification is an instance of the class diagram, i.e. an object diagram in UML terminology, which satisfies all textual and graphical constraint in the model.

Let us consider the problem of incremental verification of UML/OCL class diagrams. In this problem, we start from a UML class diagram annotated with OCL invariants, where we have verified a correctness property (satisfiability). This class diagram is called the *original model*. Moreover, we have another UML/OCL class diagram obtained by adding, changing or deleting elements from the original model. This second class diagram is called the *updated model*. Our goal is verifying the updated model efficiently, taking advantage of (a) the information about the verification of the original model and (b) the differences between the original and the updated model.

In this paper, we propose using the instance computed by the model finder on the original model as a *certificate of satisfiability*. The notion of *certificate* originates from theoretical computer science. In that field, a certificate is a piece of information that can be used to verify whether a particular string belongs to a language. For instance, when checking if a positive integer n is *composite* (it is the product of two smaller positive integers), any pair of positive integers $\langle x, y \rangle$ such that $x \cdot y = n, x < n$ and $y < n$ can be used to certify that n is composite. Certificates play a major role in complexity theory, for instance, in the definition of problems where certificates can be checked efficiently but no efficient algorithm for computing solutions has been proposed (the P vs NP problem). Similarly, the concept of certificate is also common in computational logic (Benedetti 2005), *e.g.*, a certificate of satisfiability for a logic formula.

The major contribution of our proposed method is that it is not limited to reusing the results of the analysis of the original specification, as (Wang et al. 2019; Zheng et al. 2020) Instead, whenever a change is performed to the original class diagram, we will query and update the certificate to make sure it still proves the satisfiability of the updated model, avoiding new calls to the model finder whenever possible. Notice that some actions such as checking if an OCL constraint holds in a certificate do not require invoking the model finder and can be thus performed much more efficiently. Thus, if we can replace a call to the model finder by the evaluation of an invariant in our certificate, we will greatly reduce the computational overhead. Nevertheless, for complex changes we may still need to invoke the model finder to create a new certificate. In those cases, we will nevertheless describe efficient strategies to generate a new certificate. Figure 3 shows an overview of this process, where the inputs are (a) the list of changes applied to the model, (b) the updated model and (c) an object diagram that acts as a certificate of satisfiability of the original model. This method produces a certificate of satisfiability for the updated model, if it is indeed satisfiable.

The following sections will cover the major steps in the process: (a) how to identify the list of changes that have been applied to a UML/OCL class diagram (Subsection 3.1); (b) how to assess the impact of these changes on the validity of the certificate (Subsection 3.2); (c) how to adapt the certificate to take into account these changes (Subsections 3.3 and 3.4); and (d) how to regenerate the certificate efficiently if all else fails (Subsection 3.5).

3.1. Detecting model changes

The first challenge in incremental verification is identifying the set of updates that have been applied to transform the original model into the updated model. In this Section, we outline several potential ways to detect model changes. Nevertheless, the computation of this list is out of the scope of this paper: we assume that the list of updates is provided as an input to our method.

First of all, if the designer is using a modeling tool or IDE (Magic Draw, Papyrus, Eclipse Modeling Tools) to create the UML class diagram, recording the log of actions performed by the designer inside the tool yields the list of updates. This approach has two drawbacks. First, the implementation is tightly coupled with a particular tool or IDE, so adapting it to a different tool requires development effort. Moreover, the log of actions must be analyzed to detect if there are any actions that invalidate previous actions (*e.g.* add an attribute and then delete it). These useless actions may interfere with our incremental reasoning and therefore should be removed from the list before applying our method.

Another strategy to obtain the list of updates is using a tool

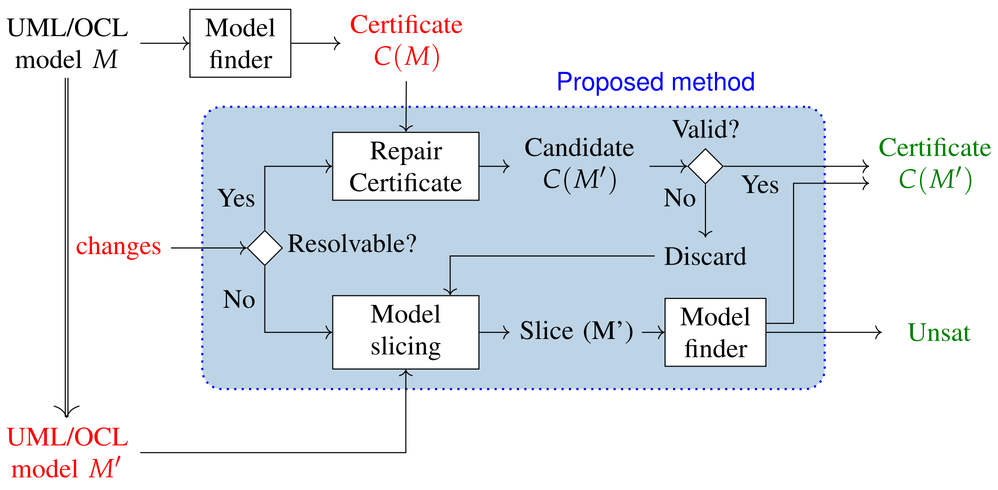


Figure 3 Overview of the proposed method (red: inputs, green: outputs).

for *model comparison* such as EMFCompare (Brun & Pierantonio 2008), DiffMerge (Thales 2015), the Epsilon Comparison Language (Kolovos 2009) or (Falleri et al. 2008). These tools are able to compute a list of differences between two models, that can be provided as an input to our approach.

Finally, some tools like USE (Gogolla et al. 2007) define a textual notation to encode both the UML class diagram and the OCL constraints. In this case, it is possible to use tools for *text comparison* such as `diff` to identify the differences between two models. The list of changes identifies the set of lines that have been added, modified or deleted, by their line number. To use this information at the model level, it is necessary to establish the correspondence between line numbers in the textual specification and the elements in the model. This requires that the parser records the line numbers where each element in the UML/OCL class diagram is defined.

3.2. List of model changes and their impact on the certificate

In this Section, we discuss the list of model changes to UML/OCL class diagrams that will be supported by our method. The changes will be classified according to their impact to the satisfiability of the model. Thus, different types of changes will be addressed differently, from no change in the certificate at all, to performing small updates in the certificate or recomputing the certificate from scratch (if an incremental update is not possible).

As a general rule, we will only consider changes in the model that result in a well-formed UML/OCL class diagram: it does not make sense to check the satisfiability of a UML/OCL model that is not well-formed. Thus, problems such as “removing the class in one association end of an association” or “assigning the same name to two classes” will not be included in our discussion.

Several works have proposed taxonomies to classify the wide variety of changes that can be applied to conceptual models (Cicchetti et al. 2008; Roddick et al. 1993; Lehnert et al. 2012; Gómez & Olivé 2002; Altmanninger et al. 2009). In this paper, we focus our taxonomy on the incremental verification of satisfiability UML/OCL models. Thus, we will classify the changes that a designer can perform to a UML class diagram according to three different criteria: (a) the type of change; (b) its effects on satisfiability; and (c) its impact on the validity of the certificate. In our description, we borrow and adapt some of the terminology used by (Cicchetti et al. 2008), tailoring it to our problem of interest (satisfiability).

Considering the type of change in the UML/OCL model, we will distinguish four categories:

Addition The inclusion of new information in the UML class diagram, *e.g.* creating a new class.

Deletion The removal of an element in the UML class diagram, *e.g.* deleting an attribute of a class.

Update The modification of the properties of an existing element in the UML class diagram, *e.g.* rewriting an OCL constraint or changing the multiplicity of an association end.

Composite A non-atomic refactoring that aggregates several atomic changes, *e.g.* pulling an attribute from a subclass to a superclass.

Then, from the point of view of the satisfiability of the model, model changes belong to one of these two categories:

Non-breaking: A change where if the original UML/OCL class diagram was satisfiable, the updated model remains satisfiable, *e.g.*, renaming an attribute[1].

Potentially breaking A change where, even if the original UML/OCL class diagram was satisfiable, the updated model may or may not be satisfiable. An example of a potentially breaking change is adding an OCL constraint to the UML/OCL model (it may make the model unsatisfiable

[1] As mentioned previously, we assume that the change preserves the well-formedness of the model, *e.g.*, the name has been changed consistently in all references to the attribute.

Element	Change	Type	Breaking?	Resolvable?
Attribute	Create with no modifiers	Addition	No	Yes
	Delete	Deletion	No	Yes
	Rename	Update	No	Yes
	Change (basic) type	Update	No	Yes
	Add `unique` modifier	Update	Yes	Depends
	Remove `unique` modifier	Update	No	Yes
	Pull to superclass	Composite	Yes	Depends
	Push to subclass	Composite	Yes	Depends
Class	Create empty class	Addition	No	Yes
	Delete	Deletion	No	Yes
	Rename	Update	No	Yes
	Make `abstract`	Update	Yes	Depends
	Make non-`abstract`	Update	Yes	No
Association	Create	Addition	Yes	Depends
	Delete	Deletion	No	Yes
	Rename association	Update	No	Yes
	Rename association end	Update	No	Yes
	Change multiplicity of an association end	Update	Yes	Depends
	Move association end from one class to another	Composite	Yes	No
Inheritance	Create generalization	Addition	Yes	No
	Create generalizaton set	Addition	Yes	No
	Delete generalization	Deletion	Yes	No
	Delete generalization set	Deletion	Yes	No
	Make a generalization set `disjoint/complete`	Update	Yes	Depends
	Make a generalization set `overlapping/incomplete`	Update	No	Yes
Invariant	Create	Addition	Yes	Depends
	Delete	Deletion	No	Yes
	Rewrite body (constraint)	Update	Yes	Depends
	Move to another class	Composite	Yes	No

Table 1 Summary of potential changes to a UML/OCL class diagram.

if it contradicts any of the previous graphical or textual constraints).

That is, potentially breaking changes may turn a satisfiable model into an unsatisfiable one, while non-breaking changes preserve satisfiability. Intuitively, any change that introduces a new constraint in the model (that may interact with previously defined constraints) or that makes a previously existing constraint more restrictive will be potentially breaking.

Finally, from the point of view of the validity of the certificates of satisfiability, model changes will be classified into two categories:

Element	Change	Action
Attribute	Create with no modifiers	Add default value to all objects of this class and all their subclasses
	Delete	Remove the value for this attribute in all objects of this class and subclasses
	Rename	Change the name of the attribute
	Change (basic) type	Assign a default value of the new type to all objects of this class and subclasses
	Remove `unique` modifier	No change required
Class	Create empty class	Add empty object of this type
	Delete	Remove all objects of this type
	Rename	Change the name of the class
Association	Delete	Remove all links of the association
	Rename association	Change the name of the association
	Rename association end	No change required
Inheritance	Make a generalization set `overlapping/incomplete`	No change required

Table 2 Actions required to adapt certificates to non-breaking model changes.

Resolvable A potentially breaking change is resolvable if the certificate of satisfiability can be (trivially) updated to provide assurance of the satisfiability of the updated model. Naturally, all model changes that are non-breaking are also resolvable.

Non-resolvable A potentially breaking change is non-resolvable if the certificate of satisfiability needs to be recomputed from scratch using a model finder.

Some changes may be resolvable or non-resolvable depending not only on the type of change, but on the specific details of what properties are being modified and what are the original and updated values for those properties. For example, adding an OCL invariant to a model may be a resolvable change: if the current certificate already satisfies the OCL invariant, no change to the certificate is required. Otherwise, the change becomes non-resolvable: we need to invoke the model finder in order to generate a new certificate of satisfiability.

Table 1 summarizes the list of changes that will be considered by our approach and classifies them according to the three criteria described in this Section. This table does not intend to be exhaustive, as many other changes could be considered, *e.g.* adding classes to generalization sets or creating an association class from an association. Nevertheless, the set of proposed changes is sufficiently rich to describe typical edit operations while creating or refining a UML class diagram.

From this Table, we can see that changes that remove constraints from elements in the model (*e.g.*, deleting an invariant) or relax these constraints (*e.g.*, setting the `isUnique` modifier of an attribute to false) are non-breaking. The rationale is that the same certificate that was satisfied in the original model will still be valid if the updated model is less restrictive. For this reason, the majority of deletion changes are non-breaking. Similarly, addition changes are also non-breaking if they add elements with no constraint attached (empty classes or attributes without modifiers) but become potentially breaking otherwise. For changes that add or modify constrained elements, their resolvability will depend on whether the certificate satisfies the new constraints or not. Finally, most composite changes are too complex and thus non-resolvable.

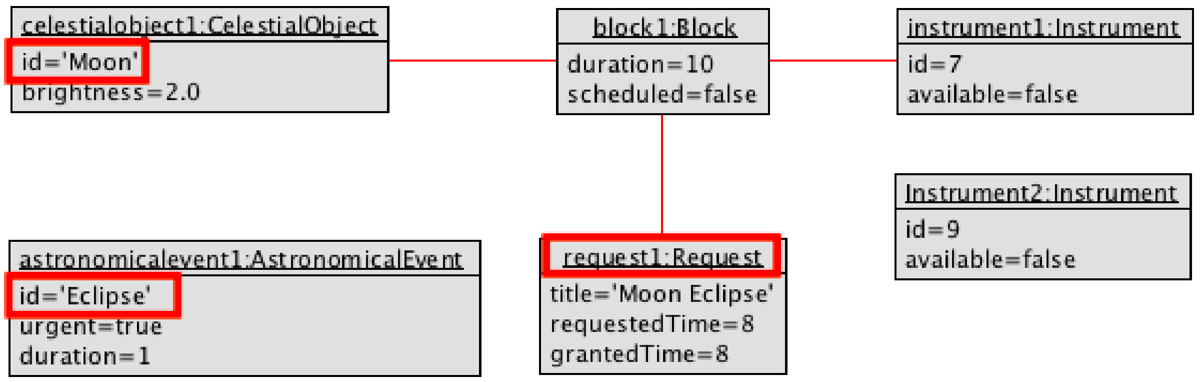
As invoking a model finder to verify a model is computationally very expensive, the goal of our approach is to avoid this call whenever possible. The best scenario is a non-breaking change, as satisfiability is preserved and few adjustments are required (see Section 3.3). If the change is potentially breaking, whenever the change is resolvable, it should be solved by adapting the certificate. Moreover, if a change is considered non-resolvable, the computation of the new certificate should be performed as efficiently as possible. The following subsections 3.4 and 3.5 consider how these type of changes are addressed.

3.3. Dealing with non-breaking changes

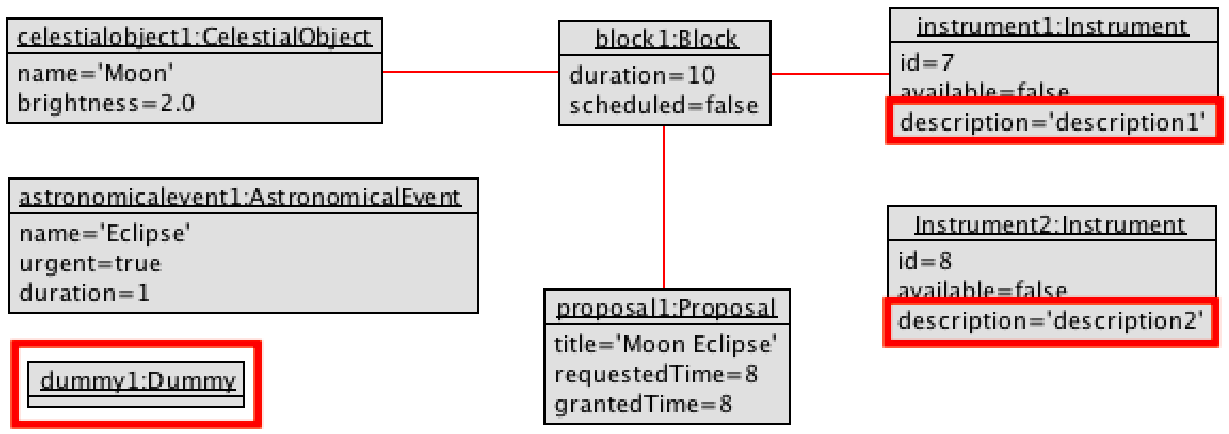
By definition, non-breaking changes preserve the satisfiability of the model. This means that the existence of a valid certificate for the updated model is assured. In particular, we can adapt the existing certificate so that it conforms to the updated model by applying minor adjustments.

Table 2 summarizes the actions that need to be applied to the certificate (an object diagram) to respond to non-breaking changes. Intuitively, deletions will make us remove information (values, objects or links) from the object diagram. Meanwhile, additions will require us to create empty objects or use default

(a) Rename class `Proposal` to `Request` and rename attribute `Target::name` to `id`



(b) Create class `Dummy` and attribute `description:String` in class `Instrument`



(c) Delete association between classes `Block` and `Instrument`

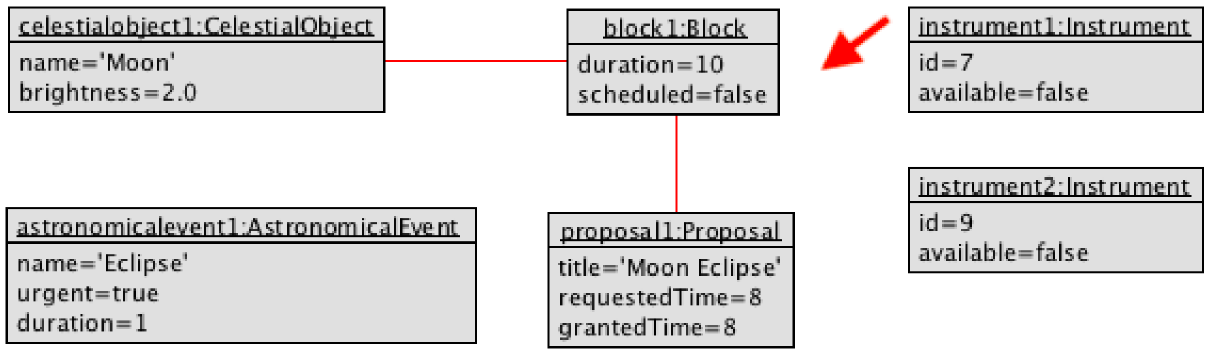


Figure 4 Repairing the object diagram from Figure 2 after non-breaking changes.

Element	Change	Action
Attribute	Add `unique` modifier	Check if there are duplicate values
		If there are, randomly remove all but one
		Re-check invariants accessing this attribute
	Pull to superclass	Apply two changes: delete attribute + add attribute; then, re-check all invariants that include this attribute
	Push to subclass	Same as “Pull to superclass”
Class	Make `abstract`	Remove all objects for this class; re-check invariants and multiplicity constraints of this class, superclasses and subclasses
Association	Create	Randomly add links between objects to satisfy the multiplicity of association ends
	Change multiplicity of an association end	Check if the new multiplicity holds. If not, randomly add / remove links to meet multiplicity constraints and re-check invariants that traverse the association
Inheritance	Make a generalization set `disjoint/complete`	Check if inheritance in this generalization set is is already `disjoint/complete` in the certificate
Invariant	Create	Check if the certificate already satisfies the invariant; if that fails, check if removing all objects that violate this invariant breaks other invariants or multiplicity constraints
	Rewrite body (constraint)	Same as “Create invariant”

Table 3 Actions that attempt to repair certificates after resolvable model changes.

values for the new elements. Then, updates may require no action at all (if the update makes a constraint less restrictive) or change the name of a class, association or link.

To illustrate this process, Figure 4 shows several examples of repairs after performing non-breaking changes to the running example (Figure 1). The baseline certificate is the object diagram shown in Figure 2. Three sets of changes are considered: (a) renaming a class and an attribute; (b) creating an empty class an attribute with no modifiers and (c) deleting an association. These three scenarios are studied separately, that is, they are not applied one after the other.

In scenario (a), the renaming applies only to the objects of the modified classes (and, for the renaming of attributes, their subclasses), changing the type name and the attribute name. In case (b), we need to create an empty object for the new class `Dummy` and add an attribute value to each object of class `Instrument`. As there is no restriction about the values of attribute `description`, we can choose any value for this attribute. We can use a default value for all objects or, as we have done in Figure 4, a random different value for each object. Finally, in scenario (c), all links of the deleted association are removed from the object diagram.

3.4. Dealing with potentially breaking resolvable changes

In contrast with non-breaking changes, potentially breaking changes do not provide the assurance that satisfiability can be preserved. Also, due to potential interactions with other constraints in the model, certificates for the updated model may be very different to those in the original model. Thus, in the worst case, we may need to invoke the model finder to generate a new certificate. However, in some cases we will be able to (a) decide the certificate is already valid or (b) repair the certificate to make it valid in the updated model.

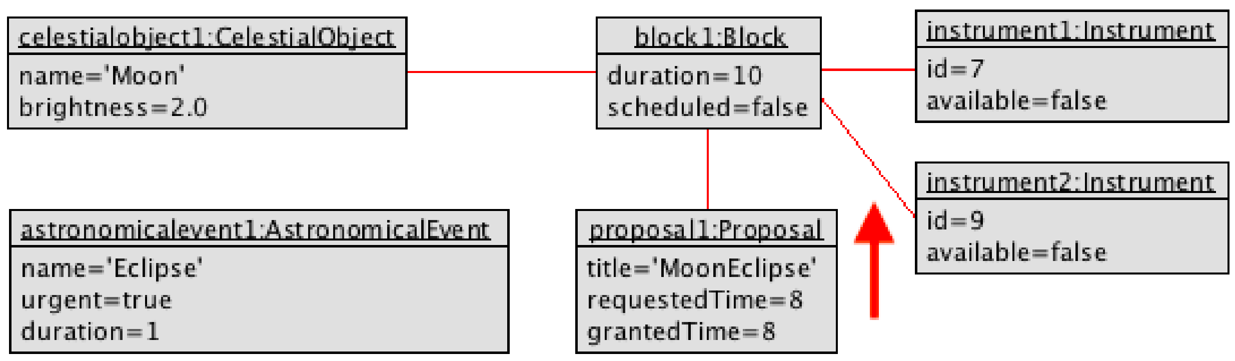
For the sake of conciseness, in the remainder of this Section we will use the term “resolvable” to refer to potentially breaking resolvable changes. Non-breaking changes, which are also resolvable, have been considered in the previous Section.

Table 3 describes the conditions for the validity of a certificate after a resolvable change and, in case the certificate can be repaired, the required actions that perform that repair. The core operation in this step is the ability to evaluate OCL conditions on the certificate and the ability to perform atomic changes in the model such as creating objects or links or modifying the value of an attribute.

To illustrate these repairs, we depict in Figure 5 the outcome of applying several resolvable changes to the model of our running example (Figure 1). This time we are considering two changes independently: (i) modifying the multiplicity of an association and (ii) adding a new invariant.

In scenario (i), we first need to evaluate whether the current certificate already satisfies the new multiplicity. This is not the case as object `block1` is only connected to one `Instrument`. Then, we check whether the change has increased the lower bound, decreased the upper bound or both. If it has increased the lower bound, we will need to add new links to the objects

(i) Modify multiplicity of association end `requires` from [1..2] to [2..2]



(ii) Add invariant `context Instrument inv maxId: (self.id <= 7)`

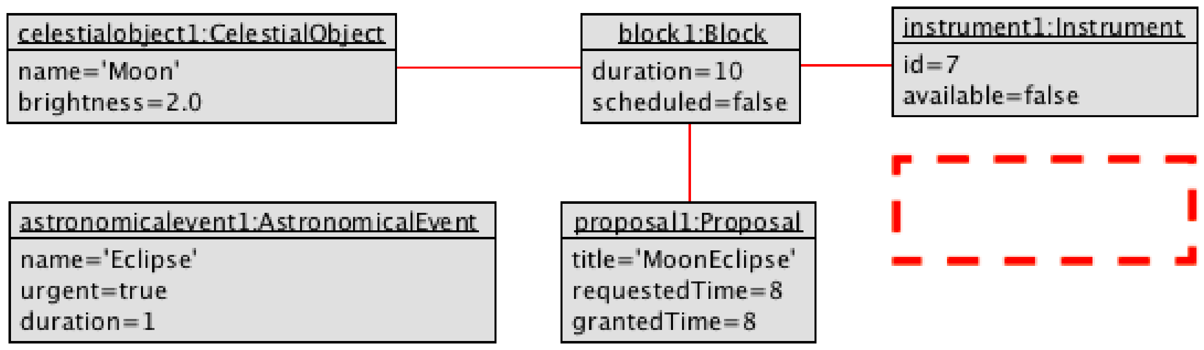


Figure 5 Repairing the object diagram from Figure 2 after resolvable changes.

that do not reach this minimum. Conversely, if it has decreased the upper bound, we will need to remove links from objects above this maximum. If the change has both increased the lower bound and decreased the upper bound, both adding and removing links is required. The choice of which links should be added or removed is performed randomly. It may happen that adding or removing links is not enough: the number of objects of the classes linked in the association ends may not be compatible with the new multiplicity. If it is necessary to create or delete objects, we will rely on the model finder to regenerate the certificate. We will also resort to the model finder if our random choice of links to be added or deleted does not satisfy the multiplicity constraints or the invariants in the model.

Regarding scenario (ii), we start by checking if the object diagram already satisfies the new invariant. This is not the case as object `instrument2` has `id = 9`. Before giving up and invoking the model finder, we can try to repair the object diagram by deleting the objects that do not satisfy the new invariant. In this case, deleting object `instrument2` does not violate any other invariant or multiplicity constraint, so we can repair the object diagram without resorting to the model finder.

The examples shown in Figure 5 manage to successfully repair the certificate. Nevertheless, in the general case the changes described in these examples may fail to repair the certificate, *e.g.*, removing the objects that do not satisfy the new invariant may violate other invariants or multiplicity constraints. If that happens, it will be necessary to invoke the model finder to regenerate the certificate.

Section 4 provides more details about how these repairs are computed and applied to the object diagram.

3.5. Dealing with potentially breaking non-resolvable changes

Breaking non-resolvable changes require to recompute the certificate from scratch using a model finder. If a certificate cannot be found, we can safely conclude the model is unsatisfiable at this point and rollback the latest changes (or report the error to the designer).

This model finding process is computationally very expensive, which is exactly why we aim to minimize the number of times is required with our certificate-based technique. Nevertheless, we can at least reuse the sequence of changes since the last checkpoint to reduce the verification effort.

Starting from the list of changes, we can perform a change impact analysis: identify the parts of the model that have been affected by the changes (and that, therefore, could be potentially in conflict) while immediately proposing a partial trivial certificate for the others. This model slicing (Shaikh, Clarisó, et al. 2010) exponentially reduces the complexity of the model finding process.

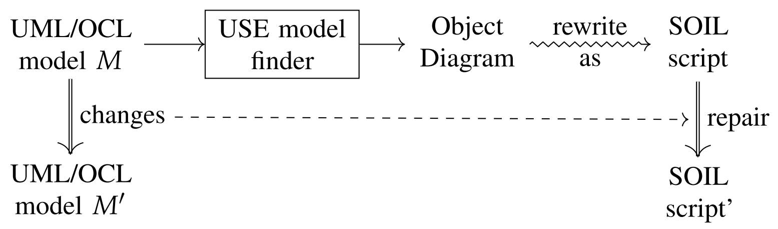


Figure 6 Tool support for incremental verification based on USE.

```
!celestialobject1 := new CelestialObject
!celestialobject1.name := 'Moon'
!celestialobject1.brightness := 2.0

!astronomicalevent1 := new AstronomicalEvent
!astronomicalevent1.name := 'Eclipse'
!astronomicalevent1.urgent := true
!astronomicalevent1.duration := 1

!proposal1 := new Proposal
!proposal1.title := 'Moon Eclipse'
!proposal1.requestedTime := 8
!proposal1.grantedTime := 8

!instrument1 := new Instrument
!instrument1.id := 7
!instrument1.available := false
!instrument2 := new Instrument
!instrument2.id := 9
!instrument2.available := false

!block1 := new Block
!block1.duration := 10
!block1.scheduled := false

!insert(proposal1, block1) into ProposalBlock
!insert(block1, celestialobject1)  into BlockTarget
!insert(block1, instrument1) into BlockInstrument
```

Figure 7 SOIL script that can be generated from the object diagram in Figure 2.

A description of the algorithm to compute the optimal submodel to be re-verified is out of the scope of this paper. Nevertheless, we provide an overview of the operation of this method: Starting by an empty submodel, the construction process could proceed by initializing the submodel with the list of all model elements directly affected by the change. Then, we would add to the submodel, transitively, all other elements connected to those already in the submodel, until no more elements can be added.

The key point is how we define "connectedness", *i.e.*, which model elements depend on each other. If we follow a conservative approach (where all elements linked to the affected ones are added), we may end up with a large submodel. A more fine-grained approach could stop the propagation when certain that an element, though physically linked to an affected one, has no impact on the verification process. As an example, consider two classes A and B related via an association R with multiplicities $0..* : 0..*$. If there is no OCL expression navigating through R in the model we can be sure that the instantiation of B has no impact on the satisfiability of A. Note that we could just make sure that R is empty, which is allowed by the multiplicity constraints, to decouple both classes. Therefore, there is no need to add B (and the transitive closure of B) to the submodel.

For instance, let us consider adding the following invariant to class `CelestialObject`:

```
-- Brightness value between 0 and 1.0
context CelestialObject inv BrightnessLimits:
  (self.brightness >= 0.0) and
  (self.brightness <= 1.0)
```

This new invariant is a *local* constraint (Shaikh, Clarisó, et al. 2010): it only restricts the values of attributes of the objects of a single class, and its validity can be established by evaluating this condition on individual objects. Moreover, no other invariant in the model (neither local nor global) constrains attribute `brightness`. As a result, we can simply invoke the solver to check this constraint considering a submodel consisting only of class `CelestialObject`. Given that our certificate in Figure 2 only has one object of type `CelestialObject`, we can instruct the bounded verification solver to perform verification with a lower and upper bound of 1 for the population of this class. If the verification is successful, we can update the values of `celestialObject1.brightness` using the output of the solver.

Depending on the list of changes and the characteristics of the model, a number of other similar rules could be proposed to minimize the size of the submodel to be reverified.

4. Tool support

In this section, we discuss the implementation of the method described in Section 3 with the help of the tool USE (UML-based Specification Environment) (Gogolla et al. 2007). For the sake of conciseness, we will focus on the repair of certificates described in subsections 3.3 and 3.4.

The USE environment offers a wide variety of features for the design, validation and verification of UML/OCL specifications. Among others, we can highlight a textual notation for describing UML/OCL models, a GUI for editing and inspecting both specifications and system states and an OCL interpreter.

(a) Rename class `Proposal` to `Request` and rename attribute `Target::name` to `id`

Original SOIL	Repaired SOIL
`!proposal1 := new Proposal`	`!proposal1 := new Request`
`!celestialobject1.name := 'Moon'`	`!celestialobject1.id := 'Moon'`
`!astronomicalevent1.name := 'Eclipse'`	`!astronomicalevent1.id := 'Eclipse'`

(b) Create class `Dummy` and attribute `description:String` in class `Instrument`

Original SOIL	Repaired SOIL
	`!dummy1 := new Dummy`
	`!instrument1.description := 'description1'`
	`!instrument2.description := 'description2'`

(c) Delete association between classes `Block` and `Instrument`

Original SOIL	Repaired SOIL
`!insert(block1, instrument1)`	
`  into BlockInstrument`	

Figure 8 SOIL scripts that repair the non-breaking changes in Figure 4.

For our work on incremental verification, we will take advantage of the following features of USE:

– A model finder plug-in (Kuhlmann et al. 2011) that relies on the relational solver Kodkod (Torlak & Jackson 2007) to compute valid system states for a UML class diagram annotated with OCL invariants. This model finder will be used to compute certificates and to regenerate certificates when a repair is not possible.
– An imperative language called SOIL (Simple OCL-based Imperative Language) (Büttner & Gogolla 2011) that allows the creation and update of system states in an specification. USE implements an interpreter for this language in its console. This language will be used as the formalism to encode, query and repair the certificate of satisfiability.

The strategy to take advantage of these tools is summarized in Figure 6. The model finder plug-in available in USE will allow us to compute a valid object diagram, which can be trivially rewritten as a SOIL script that creates each object and links and assigns the proper attribute values. For example, Figure 7 shows the SOIL script corresponding to the object diagram for the running example provided in Figure 2. Then, information about the changes applied to the model will be used to repair the certificate, that is, rewrite parts of the SOIL script. The changes correspond to the actions described in Tables 2 and 3: adding and deleting objects and links; changing attribute values; renaming model elements; or evaluating OCL expressions on the model to the check if the repair is successful.

To illustrate this process, we show in Figure 8 and 9 how to perform the repairs described in the Section 3. First, Figure 8 describes how the SOIL scripts should be modified as a result of the non-breaking changes described in Figure 4. In this case, the adaptations are straightforward.

Meanwhile, Figure 9 describes the repairs for the resolvable changes in Figure 5. In this case, the repairs need to query the certificate to check if it is still valid after the changes. We will also have to make sure that our repair produces a correct object diagram. If that is the case, we can again rewrite it as a SOIL script to process further changes.

5. Conclusion

Model finders can prove the satisfiability of UML/OCL class diagrams by computing instances of the model that fulfill all textual or graphical constraints. Nevertheless, these tools require many computational resources, so whenever possible more lightweight methods would be preferable.

In this paper, we have presented a method for the incremental verification of the satisfiability of UML/OCL class diagrams. The key element of our method is the use of valid instances of the model as certificates of satisfiability. Given a satisfiable model and a set of changes to be applied to the model, this approach attempts to repair the certificate without having to invoke a model finder. A drawback of this approach is that it only works on *satisfiable models*: it cannot help us track if a set of changes to an unsatisfiable model have made it satisfiable.

As future work, we want to advance in the implementation of the method and explore effective strategies for the *management* of certificates. So far, our discussion of the method has assumed that we only maintain a single certificate at all times. However,

(i) Modify multiplicity from [1..2] to [2..2]

```
−− Check if the certificate already satisfies the new lower bound
−− Alternatively, we could use 'check −d −a' for this check
newLowBound := 2;
problemSet := Block.allInstances()→select(b: Block | b.requires→size()<newLowBound);
if (problemSet→isEmpty()) then
  WriteLine('Certificate already valid. No change needed');
end
−− Add links randomly
for blk in problemSet do
  num :=  newLowBound − blk.requires→size();
  −− Find 'num' instruments not connected to 'blk' and create a link between them
  −− First, check that there are enough of them
  enough := num <=
     Instrument.allInstances()→select(y: Instrument | y.block→excludes(blk))→size();
  if not enough then
    WriteLine('Repair failed: multiplicity not satisfied. Invoke the model finder');
  end
  for i in Sequence{1..num} do
    instr := Instrument.allInstances()→any(y: Instrument | y.block→excludes(blk) );
    insert (blk, instr) into BlockInstrument;
  end
end

−− The multiplicity on the other side of the association is "∗"
−− This means the added links cannot violate that multiplicity constraint
−− There also no invariants traversing this association: no invariant to be rechecked
```

(ii) Add new invariant

```
−− Check if the certificate satisfies the new invariant
−− The '−d' flag lists the objects that do not  satisfy the invariant
check −d −a
−− Answer: Instrument ins2 violates maxId

−− Destroy the objects that do not satisfy the invariant
!destroy ins2

−− Check if there are some objects remaining to ensure strong satisfiability
!if Instrument.allInstances()→isEmpty() do
  WriteLine('Repair failed: A class has an empty population. Invoke the model finder');
end

−− Re−check all invariants and multiplicity constraints to see if all are fulfilled
check −d −a
−− Answer: affirmative. The certificate is now valid
```

Figure 9 SOIL scripts to repair the resolvable changes in Figure 5.

it might be more efficient to maintain several certificates. These certificates could be computed offline, when the designer is performing other tasks. Then, when a resolvable change is applied, we can try to repair each of them separately. In this way, the likelihood that at least one of the available certificates can be repaired is increased. Moreover, while we have focused on UML class diagrams, most of the components of our proposal could be easily generalizable to other types of structural models and UML-like domain-specific languages. We plan to develop such adaptations to broaden the scope of our approach.

Acknowledgments

This work is partially funded by the H2020 ECSEL Joint Undertaking Project "MegaM@Rt2: MegaModelling at Run-time" (737494) and the Spanish Ministry of Economy and Competitivity through the project "Open Data for All: an API-based infrastructure for exploiting online data sources" (TIN2016-75944-R).

References

Altmanninger, K., Seidl, M., & Wimmer, M. (2009). A survey on model versioning approaches. *International Journal of Web Information Systems*, 271–304. doi: 10.1108/17440080910983556

Bagheri, H., & Malek, S. (2016). Titanium: efficient analysis of evolving Alloy specifications. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering (fse)* (pp. 27–38). doi: 10.1145/2950290.2950337

Balaban, M., & Maraee, A. (2013). Finite satisfiability of UML class diagrams with constrained class hierarchy. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *22*(3), 24:1–24:42. doi: 10.1145/2491509.2491518

Benedetti, M. (2005). Extracting certificates from quantified boolean formulas. In *19th international joint conference on artificial intelligence (ijcai'2005)* (pp. 47–53). Professional Book Center.

Blanc, X., Mougenot, A., Mounier, I., & Mens, T. (2009). Incremental detection of model inconsistencies based on model operations. In *International conference on advanced information systems engineering (caise)* (Vol. 5565, pp. 32–46). doi: 10.1007/978-3-642-02144-2_8

Brosch, P., Seidl, M., Wimmer, M., & Kappel, G. (2012). Conflict visualization for evolving UML models. *Journal of Object Technology*, *11*(3), 2: 1–30. doi: 10.5381/jot.2012.11.3.a2

Brucker, A. D., & Wolff, B. (2008). HOL-OCL: a formal proof environment for UML/OCL. In *International conference on fundamental approaches to software engineering (fase)* (Vol. 4961, pp. 97–100). doi: 10.1007/978-3-540-78743-3_8

Brun, C., & Pierantonio, A. (2008). Model differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional*, *9*(2), 29–34.

Büttner, F., & Gogolla, M. (2011). Modular embedding of the Object Constraint Language into a programming language. In *14th brazilian symposium on formal methods, foundations and applications (SBMF)* (Vol. 7021, pp. 124–139). Springer. doi: 10.1007/978-3-642-25032-3_9

Cabot, J., Clarisó, R., & Riera, D. (2014). On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, *93*, 1–23. doi: 10.1016/j.jss.2014.03.023

Cabot, J., & Gogolla, M. (2012). Object constraint language (OCL): A definitive guide. In *Formal methods for model-driven engineering - 12th international school on formal methods for the design of computer, communication, and software systems, SFM 2012, bertinoro, italy, june 18-23, 2012. advanced lectures* (pp. 58–90). doi: 10.1007/978-3-642-30982-3_3

Cabot, J., & Teniente, E. (2009). Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software*, *82*(9), 1459–1478. doi: 10.1016/j.jss.2009.03.009

Cicchetti, A., Ruscio, D. D., Eramo, R., & Pierantonio, A. (2008). Automating co-evolution in model-driven engineering. In *12th international IEEE enterprise distributed object computing conference, (EDOC)* (pp. 222–231). IEEE Computer Society. doi: 10.1109/EDOC.2008.44

Clarisó, R., González, C. A., & Cabot, J. (2017). Smart bound selection for the verification of UML/OCL class diagrams. *IEEE Transactions on Software Engineering*, *45*(4), 412–426. doi: 10.1109/TSE.2017.2777830

Clavel, M., Egea, M., & de Dios, M. A. G. (2010). Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST*, *24*. doi: 10.14279/tuj.eceasst.24.334

Dania, C., & Clavel, M. (2016). OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints. In *Acm/ieee 19th international conference on model driven engineering languages and systems (models)* (pp. 65–75).

Egyed, A. (2007). UML/Analyzer: A tool for the instant consistency checking of UML models. In *Proceedings of the 29th international conference on software engineering (icse)* (pp. 793–796). doi: 10.1109/ICSE.2007.91

Falleri, J.-R., Huchard, M., Lafourcade, M., & Nebut, C. (2008). Metamodel matching for automatic model transformation generation. In *International conference on model driven engineering languages and systems (models)* (Vol. 5301, pp. 326–340). doi: 10.1007/978-3-540-87875-9_24

Gabmeyer, S., Kaufmann, P., Seidl, M., Gogolla, M., & Kappel, G. (2019). A feature-based classification of formal verification techniques for software models. *Software and Systems Modeling*, *18*(1), 473–498. doi: 10.1007/s10270-017-0591-z

Ganov, S., Khurshid, S., & Perry, D. E. (2012). Annotations for Alloy: Automated incremental analysis using domain specific solvers. In *International conference on formal engineering methods (icfem)* (Vol. 7635, pp. 414–429). doi: 10.1007/978-3-642-34281-3_29

Gogolla, M., Büttner, F., & Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, *69*(1-3), 27–34. doi: 10.1016/j.scico.2007.01.013

Gómez, C., & Olivé, A. (2002). Evolving partitions in conceptual schemas in the UML. In *14th international conference on* *advanced information systems engineering (caise)* (Vol. 2348, pp. 467–483). Springer. doi: 10.1007/3-540-47961-9_33

González, C. A., & Cabot, J. (2014). Formal verification of static software models in MDE: A systematic review. *Information and Software Technology*, *56*(8), 821–838. doi: 10.1016/j.infsof.2014.03.003

Izquierdo, J. L. C., & Cabot, J. (2016). Collaboro: a collaborative (meta) modeling tool. *PeerJ Computer Science*, *2*, e84. doi: 10.7717/peerj-cs.84

Jackson, D. (2012). *Software abstractions: logic, language, and analysis*. MIT press.

Jackson, E. K., Levendovszky, T., & Balasubramanian, D. (2011). Reasoning about metamodeling with formal specifications and automatic proofs. In *International conference on model driven engineering languages and systems (models)* (pp. 653–667). doi: 10.1007/978-3-642-24485-8_48

Jouault, F., & Beaudoux, O. (2016). Efficient OCL-based incremental transformations. In *International workshop on ocl and textual modeling (ocl@models)* (Vol. 1756, pp. 121–136). CEUR-WS.org.

Khalil, A., & Dingel, J. (2013). *Supporting the evolution of UML models in model driven software development: a survey* (Tech. Rep. No. 602). School of Computing, Queen's University.

Kolovos, D. S. (2009). Establishing correspondences between models with the Epsilon comparison language. In *European conference on model driven architecture-foundations and applications (ecmfa)* (Vol. 5562, pp. 146–157). doi: 10.1007/978-3-642-02674-4_11

Kuhlmann, M., Hamann, L., & Gogolla, M. (2011). Extensive validation of OCL models by integrating SAT solving into USE. In *49th international conference on objects, models, components, patterns (tools)* (Vol. 6705, pp. 290–306). doi: 10.1007/978-3-642-21952-8_21

Lehnert, S., Farooq, Q., & Riebisch, M. (2012). A taxonomy of change types and its application in software evolution. In *IEEE 19th international conference and workshops on engineering of computer-based systems (ECBS)* (pp. 98–107). IEEE Computer Society. doi: 10.1109/ECBS.2012.9

Oriol, X., & Teniente, E. (2015). Incremental checking of OCL constraints with aggregates through SQL. In *International conference on conceptual modeling (er)* (Vol. 9381, pp. 199–213). doi: 10.1007/978-3-319-25264-3_15

Oriol, X., & Teniente, E. (2017). OCL$_{\mathrm{univ}}$: Expressive UML/OCL conceptual schemas for finite reasoning. In *International conference on conceptual modeling (er)* (Vol. 10650, pp. 354–369). doi: 10.1007/978-3-319-69904-2_28

Petre, M. (2013). UML in practice. In *International conference on software engineering (icse)* (pp. 722–731). IEEE Computer Society. doi: 10.1109/ICSE.2013.6606618

Queralt, A., Artale, A., Calvanese, D., & Teniente, E. (2012). OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering*, *73*, 1–22. doi: 10.1016/j.datak.2011.09.004

Reder, A., & Egyed, A. (2012). Incremental consistency checking for complex design rules and larger model changes. In *International conference on model driven engineering languages and systems (models)* (Vol. 7590, pp. 202–218). doi: 10.1007/978-3-642-33666-9_14

Roddick, J. F., Craske, N. G., & Richards, T. J. (1993). A taxonomy for schema versioning based on the relational and entity relationship models. In *12th international conference on the entity-relationship approach (er)* (Vol. 823, pp. 137–148). Springer. doi: 10.1007/BFb0024363

Romero, J. R., Rivera, J. E., Durán, F., & Vallecillo, A. (2007). Formal and tool support for model driven engineering with Maude. *Journal of Object Technology*, *6*(9), 187–207. doi: 10.5381/jot.2007.6.9.a10

Rosner, N., Siddiqui, J. H., Aguirre, N., Khurshid, S., & Frias, M. F. (2013). Ranger: Parallel analysis of Alloy models by range partitioning. In *28th ieee/acm international conference on automated software engineering (ase)* (pp. 147–157). doi: 10.1109/ASE.2013.6693075

Rull, G., Farré, C., Queralt, A., Teniente, E., & Urpí, T. (2015). AuRUS: explaining the validation of UML/OCL conceptual schemas. *Software and Systems Modeling*, *14*(2), 953–980. doi: 10.1007/s10270-013-0350-8

Semeráth, O., Nagy, A. S., & Varró, D. (2018). A graph solver for the automated generation of consistent domain-specific models. In *International conference on software engineering (icse)* (pp. 969–980). doi: 10.1145/3180155.3180186

Semeráth, O., Vörös, A., & Varró, D. (2016). Iterative and incremental model generation by logic solvers. In *International conference on fundamental approaches to software engineering (fase)* (Vol. 9633, pp. 87–103). doi: 10.1007/978-3-662-49665-7_6

Sen, S., Moha, N., Baudry, B., & Jézéquel, J.-M. (2009). Metamodel pruning. In *International conference on model driven engineering languages and systems (models)* (Vol. 5795, pp. 32–46). doi: 10.1007/978-3-642-04425-0_4

Shaikh, A., Clarisó, R., Wiil, U. K., & Memon, N. (2010). Verification-driven slicing of UML/OCL models. In *25th IEEE/ACM international conference on automated software engineering (ASE)* (pp. 185–194). doi: 10.1145/1858996.1859038

Shaikh, A., Clarisó, R., Wiil, U. K., & Memon, N. (2010). Verification-driven slicing of UML/OCL models. In *Ieee/acm international conference on automated software engineering (ase)* (pp. 185–194).

Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., & Drechsler, R. (2010). Verifying UML/OCL models using boolean satisfiability. In *Design, automation and test in europe (date)* (pp. 1341–1344). IEEE Computer Society. doi: 10.1109/DATE.2010.5457017

Soltana, G., Sabetzadeh, M., & Briand, L. C. (2020). Practical constraint solving for generating system test data. *ACM Trans. Softw. Eng. Methodol.*, *29*(2), 11:1–11:48. Retrieved from https://doi.org/10.1145/3381032 doi: 10.1145/3381032

Thales. (2015). *EMF Diff/Merge.* Retrieved from https://projects.eclipse.org/projects/modeling.emf.diffmerge

Torlak, E., & Jackson, D. (2007). Kodkod: A relational model finder. In *International conference on tools and algorithms for the construction and analysis of systems (tacas)* (Vol. 4424, pp. 632–647). doi: 10.1007/978-3-540-71209-1_49

Uzuncaova, E., & Khurshid, S. (2008). Constraint prioritization for efficient analysis of declarative models. In *International symposium on formal methods (fm)* (Vol. 5014, pp. 310–325). doi: 10.1007/978-3-540-68237-0_22

Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., & Ujhelyi, Z. (2016). Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and Systems Modeling*, *15*(3), 609–629. doi: 10.1007/s10270-016-0530-4

Wang, W., Wang, K., Gligoric, M., & Khurshid, S. (2019). Incremental analysis of evolving Alloy models. In *International conference on tools and algorithms for the construction and analysis of systems (tacas)* (Vol. 11427, pp. 174–191). doi: 10.1007/978-3-030-17462-0_10

Wu, H. (2017). MaxUSE: A tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In *International conference on integrated formal methods (ifm)* (Vol. 10510, pp. 348–356). doi: 10.1007/978-3-319-66845-1_23

Zheng, G., Bagheri, H., Rothermel, G., & Wang, J. (2020). Platinum: Reusing constraint solutions in bounded analysis of relational logic. In *International conference on fundamental approaches to software engineering (fase)* (Vol. 12076, pp. 29–52). Springer. doi: 10.1007/978-3-030-45234-6_2

About the authors

Robert Clarisó received his BSc (2000) and PhD (2005) in Computer Science from UPC-Barcelona Tech. Since 2005, he is a lecturer at the IT, Multimedia and Telecommunications Department of Universitat Oberta de Catalunya (UOC). He is also a member of the SOM Research Lab within the Internet Interdisciplinary Institute (IN3-UOC). His research interests include formal methods, model-driven engineering and tools for e-learning You can contact him at rclariso@uoc.edu or visit https://robertclariso.github.io/.

Carlos A. González received his PhD degree from the École des Mines de Nantes (EMN) in 2014. After that, he was a research associate at the SnT Centre for Security, Reliability and Trust of the University of Luxembourg. Since 2018, he is a member of the R&D area of Grantecan S.A., the company responsible for the design, construction and operation of GTC, the world's largest, single-aperture, optical telescope currently in operation. His research interests include, but are not limited to, model-driven engineering and software verification and validation. You can contact the author at carlos.gonzalez@gtc.iac.es.

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