Adapting Integrity Checking Techniques for Concurrent Operation Executions

Xavier Oriol and Ernest Teniente

Department of Service and Information System Engineering
Universitat Politècnica de Catalunya – BarcelonaTech
{xoriol,teniente}@essi.upc.edu

Abstract. One challenge for achieving executable models is preserving the integrity of the data. That is, given a structural model describing the constraints that the data should satisfy, and a behavioral model describing the operations that might change the data, the integrity checking problem consists in ensuring that, after executing the modeled operations, none of the specified constraints is violated.
A multitude of techniques have been presented so far to solve the integrity checking problem. However, to the best of our knowledge, all of them assume that operations are not executed concurrently. As we are going to see, concurrent operation executions might lead to violations not detected by these techniques.
In this paper, we present a technique for detecting and serializing those operations that can cause a constraint violation when executed concurrently, so that, previous incremental techniques, exploiting our approach, can be safely applied in systems with concurrent operation executions guaranteeing the integrity of the data.

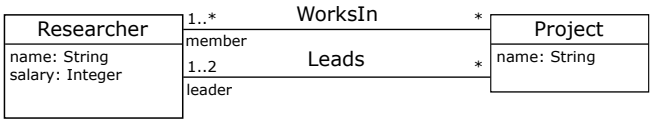
Keywords: Integrity Checking, Concurrent Operations, UML/OCL

1 Introduction

One of the main challenges for achieving executable models is ensuring data integrity[1]. That is, given a structural model describing the data and its integrity constraints, such as an UML diagram with OCL invariants; and a behavioral model describing the operations that can change this data, like OCL operation contracts for instance, the integrity checking problem consists in assessing whether the particular execution of a given operation in the current data state may induce a constraint violation. The difficulty of this problem is clear since, in the context of SQL databases, the integrity checking problem was already defined more than 25 years ago (under the form of SQL assertions checking [2]) and, still, none of the current major database management systems has implemented a solution for it (Oracle, SQL Server, DB2, PostreSQL, MySQL).

As an example, consider the structural model of Figure 1, written in UML/OCL, of a system for managing a research group. In this system, we have some *researchers* who *work in* some *projects*. Moreover, some of these researchers *lead*

some of these projects, although a project might have a maximum of two leaders. The OCL invariants states that researchers and projects are identified by their name (*ReserchерPK*, and *ProjectPK* invariants), a leader of a project is also a member of the project (*LeaderIsMember* invariant), and that the salary of a leader of a project is higher than the salary of all its members (*LeaderEarnsMore* invariant). Note that these constraints might be violated because of the actions of the operations, as they are specified in the behavioral model.



context Resercher **inv** ReserchерPK:
Researcher.allInstances()->isUnique(name)

context Project **inv** ProjectPK:
Project.allInstances()->isUnique(name)

context Project **inv** LeaderIsMember:
self.member->includesAll(self.leader)

context Project **inv** LeaderEarnsMore:
self.leader->forAll(l|self.member->forAll(m|l.salary > m.salary))

Fig. 1. Structural model of a research group management system

In Figure 2, we show a fragment of the behavioral model for this system. In this model, we show the operation contracts, written in OCL, of four operations. The first one is required for adding new researchers (*hireResearcher*), the second one for assigning a leader to a project (*addLeader*), the third one for including a member in a project (*addMember*), and the last one for removing a member from a project (*removeMember*).

Op hireResearcher(name: String, salary: Integer)
post: Researcher.allInstances()->exists(r|r.oclIsNew() and r.name = name and r.salary = salary)

Op addLeader(rName: String, pName: String)
pre: Researcher.allInstances().name->includes(rName)
pre: Project.allInstances().name->includes(pName)
post: Project.allInstances()->select(p|p.name = pName).leader.name->includes(rName)

Op addMember(rName: String, pName: String)
pre: Researcher.allInstances().name->includes(rName)
pre: Project.allInstances().name->includes(pName)
post: Project.allInstances()->select(p|p.name = pName).member.name->includes(rName)

Op removeMember(rName: String, pName: String)
pre: Researcher.allInstances().name->includes(rName)
pre: Project.allInstances().name->includes(pName)
post: Project.allInstances()->select(p|p.name = pName).member.name->excludes(rName)

Fig. 2. Behavioral model of a research group management system

Depending on the current state of the information base, executing some of these operations with certain parameters can lead to a constraint violation. For instance, if we execute *addLeader* with parameters *Mary* and *ModelsProject*, but *Mary* is not currently a member of *ModelsProject*, the execution of the operation violates the *LeaderIsMember* constraint. The difficulty of this problem scales rapidly when complicating the operations and constraints involved.

To solve this problem, several proposals have been made in the modeling community based on incremental techniques [3,4,5,6,7]. Briefly, incremental integrity checking techniques are based on the idea that, assuming that the current data state satisfies all the constraints, they check whether the data updated by an operation execution leads to a violation without inspecting the rest of the data. For instance, following our previous example, we would only need to check whether *Mary* is a member of *ModelsProject* and, thus, there is no need to check other project leaders such as *John*, since *John* is not affected by the update.

However, to the best of our knowledge, all the presented techniques assume that operations are executed isolatedly, and thus, are not able to detect integrity violations when two operations executed concurrently collaborate to cause a constraint violation. For instance, assume that in our current data state *Mary* is a member of *ModelsProject*. In this situation, executing the operation to make *Mary* a leader of the project does not violate a constraint. In the same situation, executing, instead, an operation to remove *Mary* from the *ModelsProject* does not violate a constraint either. However, when executing both operations simultaneously, both collaborate to reach a new state in which *Mary* leads a project where she is not a member of. Thus, they raise a constraint violation.

This means that, right now, if we use the previous incremental techniques with systems that admit concurrent operation executions, some violations are going to be missed (i.e., previous incremental checking techniques are not complete when considering concurrency). Clearly, the problem can be solved by forcing all the operations to be executed in a serialized manner, but this might heavily penalize the runtime efficiency of the system.

Fortunately, not all operations must be executed in a serialized manner to avoid these violations. Indeed, not all operations can collaborate to cause a constraint violation. For instance, operations *addLeader* and *removeMember* can collaborate to violate *LeaderIsMember* and must be serialized, but operations *addLeader* and *hireResearcher* cannot collaborate to violate any constraint, and thus, can be executed concurrently.

In this paper, we define a method for identifying, and serializing, those operation executions that can collaborate to cause a constraint violation, permitting the rest of operations to be executed concurrently. In this way, we allow using the previous incremental techniques in systems with concurrent operations, without the penalization of serializing every execution, neither loosing completeness. Our technique has been implemented in a tool for executing UML/OCL models [8], thus, showing that it is feasible in practice. In any case, since the core of our technique is fully based on logics, it can be adapted and implemented in other model executor tools using UML/OCL[9,10] or other modeling languages.

It is worth to mention that our work is, somehow, similar to the one in [11]. In particular, [11] detects operations invoked in a wrong order due to CRUD inconsistencies (e.g. reading some information deleted). We argue that our method and theirs can be combined, since both deal with different problems due to concurrency. Note, additionally, that our work is about checking a constraint on runtime assuming concurrency, and not on verifying/validating the models at compile time such as [12].

2 Basic Concepts and Notation

We review some key concepts and the basics of the notation used in the paper.

Terms, atoms and literals A *term* t is either a variable or a constant. An *atom* is formed by a n-ary *predicate* p together with n terms, i,e. $p(t_1, ..., t_n)$. We may write $p(\bar{t})$ for short. If all the terms $\bar{t}$ of an atom are constants, we call the atom to be *ground*. A literal l is either an atom $p(\bar{t})$, a negated atom $\neg p(\bar{t})$, or a built-in literal $t_i\ \omega\ t_j$, where ω is an arithmetic comparison (i,e. $<$,$\leq$,$=$,$\geq$,$>$,$\neq$).

Derived/Base predicates A predicate p is said to be *derived* if the boolean evaluation of an atom $p(\bar{t})$ depends on some derivation rules, otherwise, it is said to be *base*. A *derivation rule* has the form: $\forall \bar{t}.\ p(\overline{t_p}) \leftarrow \phi(\bar{t})$ where $\overline{t_p} \subseteq \bar{t}$. In the formula, $p(\overline{t_p})$ is an atom called the *head* of the rule and $\phi(\bar{t})$ is a conjunction of literals called the *body*. We suppose all derivation rules to be safe (i.e. all the variables appearing in the head or in a negated or built-in literal of the body also appears in a positive literal of the body) and non-recursive. Given several derivation rules with predicate p in its head, $p(\bar{t})$ is evaluated to true if and only if one of the bodies of such derivation rules is evaluated to true.

Logic Formalization of the UML Schema. As proposed in [13] we formalize each class C in the class diagram with attributes $\{A_1, \ldots, A_n\}$ by means of a base atom $c(Oid, A_1, ..., A_n)$, each association R between classes $\{C_1, \ldots, C_k\}$ by means of a base atom $r(C_1, \ldots, C_k)$ and, similarly, each association class R between classes $\{C_1, \ldots, C_k\}$ and with attributes $\{A_1, \ldots, A_n\}$ by means of a base atom $r(Oid, C_1, \ldots, C_k, A_1, \ldots, A_n)$.

Roughly speaking, when an object/relation encoded as $P(\overline{x})$ exists in some data state, the ground literal $P(\overline{x})$ evaluates to true in such data state. Conversely, when an object/relation encoded as $P(\overline{x})$ does not exists in some data state, the ground literal $P(\overline{x})$ evaluates to false in such data state.

3 Our Approach

Our approach is based on the notion of structural events. A structural event is an elementary change in the population of the data, that is, an insertion or deletion of a class/association instance. For instance, inserting *Leads(Mary, ModelsProject)*, or deleting *WorksIn(Mary, ModelsProject)* are structural events. For our purposes, we encode insertion structural events with the prefix *ins*, and deletion structural events with the prefix *del*, e.g., the previous structural events

are encoded as *ins_Leads(Mary, ModelsProject)*, and *del_WorksIn(Mary, ModelsProject)*, respectively. Attribute updates can be seen as an insertion/deletion of the same object.

Executing an operation leads to structural events in the data, and these structural events might change the evaluation of a constraint, that is, the structural events might violate a constraint, or even repair a constraint that was going to be violated. For instance, executing the operation *addLeader* causes the structural event *ins_Leads* that might violate *LeaderIsMember*; on the contrary, executing the operation *add_Member* causes the structural event *ins_WorksIn* that might repair such violation.

The operations that must be serialized depend on the time where the chosen integrity checking technique takes place. In essence, the integrity checking techniques can be applied *before* executing the structural events (such as [3]), which we refer as *precondition-time checking*; or *after* it (such as [7]), which we refer as *postcondition-time checking*. In the first case, we need to serialize two operations O_1, O_2 that can collaborate to cause a violation; on the second, we need to serialize two operations O_1 and O_2 if the structural events of O_1 might compensate the effects of O_2, since a rollback of O_1 might affect the consistency of O_2.

For instance, consider the operations *addLeader*, *removeMember*, and *addMember* from our running example. Using a preconditiom-time checking, the operations *addLeader* and *removeMember* should never be applied concurrently since they might collaborate to cause a constraint violation, and the checking technique will not realize of it since it makes the analysis separately. Note, however, that a postcondition-time checking will find the violation since, at the time of performing the analysis, both operations have been executed and all their effects are in the information system (and thus, at the time of checking the consistency of the data, the postcondition-time checking can find a leader not being a member of its project). However, in the case of a postconditiom-time checking, the operations that should not be executed (or at least analyzed) together are *addMember* and *addLeader*, since a rollback (or not) of the first might imply a violation (or not) of the second operation. Indeed, if we execute *addMember* and *addLeader*, and analyze together the consistency of the data, we might find that *addLeader* does not violate the *LeaderIsMember* constraint because the operation *addMember* adds the new leader as a member for the project, but if *addMember* violates any other constraint and must rollback, this rollback makes *addLeader* violate the *LeaderIsMember*. Thus, we should analyze the consistency of *addLeader* after the consistency analysis of *addMember*. Note that this problem does not occur in precondition-time checking techniques.

Formally, when dealing with integrity checking in systems with concurrent operations, we identify two kinds of interactions between operations that must be taken into account:

- *Violation collaboration*. There is a violation collaboration between two operations O_1 and O_2 if, for some constraint C, the structural events applied by O_1 and O_2 might violate C.

– *Compensation interaction.* There is a compensation interaction from O_1 to O_2 if, for some constraint C, the structural events applied by O_1 might repair a violation of C caused by the structural events of O_2.

In the case of *precondition-time checking*, we must serialize two operations O_1 and O_2 if they have a violation collaboration; in the case of *postcondition-time checking*, two operations O_1 and O_2 must be serialized if O_1 has a compensation interaction with O_2.

In this paper we focus on detecting this kind of interactions, and we suggest a serialization to deal with the problems they can carry out. Other approaches different than serialization, or a more refined versions of serialization, can be studied, but they are left for further work.

To detect this kind of interactions, we apply the following steps: 1) given all the operation contracts $\mathcal{O}$, we detect the kind of structural events applied by each operation $O \in \mathcal{O}$, 2) given all constraints $\mathcal{C}$, we detect all the kind of structural events that can violate/repair each $C \in \mathcal{C}$, 3) for each pair of operations O_1, O_2, and each constraint C, we use the structural events to analyze if there is any kind of interaction between them w.r.t. C. Note that all these analysis can be performed at compile time since they purely rely on the model specification of the operations and constraints.

3.1 Detecting the kind of structural events applied by some operation

Given an operation contract, it is possible to identify, at compile time, which are the kind of structural events applied by the operation [14,15]. For our purposes, we rely on the approach of [14] to detect them. In essence, the idea behind this approach is to translate any operation contract to an equivalent logic formula that, intuitively, states that executing of an operation implies the application of certain structural events.

In particular, the previous operations from Figure 2 can be encoded by means of the following logic formulas:

```
ins_Researcher(R, Name, Salary) :- hireResearcher(Name, Salary)
ins_Leads(R, P) :- addLeader(RN, PN), Researcher(R, RN, S),Project(P, PN)
del_WorksIn(R, P) :- removeMember(RN, PN), Researcher(R, RN, S),Project(P, PN)
ins_WorksIn(R, P) :- addMember(RN, PN), Researcher(R, RN, S),Project(P, PN)
```

Intuitively, the first formula states that invoking the operation *hireResearcher* with parameters *Name*, *Salary* causes the structural event of *ins_Researcher(R, Name, Salary)* to happen, where R is a new object identifier value. The second one states that, when invoking the operation *addLeader* with parameters *RN* and *PN*, there is a structural event *ins_Leads(R, P)* provided that R and P are the researcher and project identified by *RN* and *PN*, respectively. Similarly, the third formula states that, when invoking the operation *removeMember*, there is a *del_WorksIn(R,P)* structural event.

Thus, and thanks to this translation which is already implemented [16], the structural events implied by each operation become explicit in the head of each

rule. Thus, we can build a program that reads this translation, and realizes that executing *hireResearcher* implies the structural event *ins_Reseracher*, *addLeader* implies *ins_Leads*, and *removeMember* implies *del_WorksIn*.

Note that, in general, an operation will apply more than one kind of structural event when executed. For instance, we could specify an operation that creates a new researcher and adds his membership associations. In this case, and following [14], an operation is translated into several logic formulas, each one implying a different structural event. Thus, the structural events implied by such operation is the union of all the structural events appearing in all the formulas.

3.2 Detecting the structural events that violate/repair a constraint

Given a constraint C, it is possible to determine, at compile time, which are the kind of structural events that might violate a constraint, and also those that may repair it [17,7]. For our purposes, we use the approach defined in [17] since it is based on logics in a similar way as we did in previous section.

In essence, we first translate the UML and OCL constraints into logic denials, that is, logic formulas stating the condition that rise a constraint violation. Following, for instance, the automatic translation of UML/OCL constraints to denials defined in [13], our running example would bring the following logic formulas:

```
:- Researcher(R1, N, S1), Researcher(R2, N, S2), R1<>R2
:- Project(P1, N), Project(P2, N), P1<>P2
:- Leads(R,P), not(WorksIn(R,P))
:- WorksIn(R,P), Leads(L,P), Researcher(R,RN,RS), Researcher(L,LN,LS),RS>LS
```

The first and second formulas, encode that, if there are two different researchers or projects with the same name, there is a constraint violation. The third one states a constraint violation if R leads a project P where s/he does not work in. The last formula asserts a violation if for some project P, there is a leader L that earns less than a worker R.

Given the logic formulas, we can realize which structural events might make these formulas true (and thus, rise a violation), and which of them might make them false (and thus, repair the violation).

To do so, we rely on the event rule equivalences [18]. The event rule equivalences define when a structural event makes a literal true/false in the new state of the data after applying the events. In particular, consider P^N to be the literal P evaluated in the new data state. Then, the event rule equivalences tells us that:

$$P^N(\overline{x}) \equiv ins\_P(\overline{x}) \vee (P(\overline{x}) \wedge \neg del\_P(\overline{x}))$$
$$\neg P^N(\overline{x}) \equiv del\_P(\overline{x}) \vee (\neg P(\overline{x}) \wedge \neg ins\_P(\overline{x}))$$

Intuitively, the literal $P(\overline{x})$ is true in the new state after applying the structural events if we have inserted $P(\overline{x})$ through some insertion structural event, or $P(\overline{x})$ was already true in the data state and we have not deleted it. Similarly,

$\neg P(\overline{x})$ is true in the new state after applyng the structural events if we have delted $P(\overline{x})$ through some deletion structural event, or $P(\overline{x})$ was already false in the data state and we have not inserted it.

Applying the previous equivalences to our logic denials, we obtain what we call event-dependency constraints (EDCs), that is, denials that tells which structural events rise a constraint violation. For instance, for the first denial we obtain:

```
:- ins_Researcher(R1, N, S1), ins_Researcher(R2, N, S2), R1<>R2
:- ins_Researcher(R1,N,S1), Researcher(R2,N,S2), not del_Researcher(R2,N,S2),
       R1<>R2
:- Researcher(R1,N,S1), not del_Researcher(R1,N,S1), ins_Researcher(R2,N,S2),
       R1<>R2
:- Researcher(R1,N,S1), not del_Researcher(R1,N,S1), Researcher(R2,N,S2),
       not del_Researcher(R2,N,S2), R1<>R2
```

The first EDCs states that there is a constraint violation if we apply two different structural events for inserting a researcher with the same name. The second and third one specify that if we insert a new researcher with a name N, and this name N belongs to some researcher in the current data, but we do not remove this researcher, there is a constraint violation. Finally, the last rule tells us that if we have two researchers with the same name and we do not remove any of them, there is a cosntraint violation.

Intuitively, the structural events that appear positively in an EDC are the structural events that might cause a violation, while those that appear negatively in an EDC are the structural events that might repair the violation (since they make the body of the EDC, which detects the violation, to evaluate to false). For instance, *ins_Researcher* is a structural event that can cause a violation of the *ResearcherPK* constraint, while *del_Researcher* is a structural event that can repair it.

It is worth to hightlight that the number of EDCs obtained from one denial grows exponentially with the lenght of the denial encoding. However, some optimizations can be applied to reduce the number and size of the denials [3]. Indeed, considering the classical optimization that the initial data state does not violate any constraint, and that there is homomorphism between denials two and three, the unique EDCs required are:

```
:- ins_Researcher(R1, N, S1), ins_Researcher(R2, N, S2), R1<>R2
:- ins_Researcher(R1,N,S1), Researcher(R2,N,S2), not del_Researcher(R2,N,S2),
       R1<>R2
```

3.3 Detecting operations and constraints interactions through the structural events

At this point, we want to analyze, using the structural events previously determined, which kind of interactions might have two operations w.r.t. some constraint. To do so, and benefiting from the fact that all our approach is based on logics, we are going to use an unfolding technique. In essence, our idea is to unfold the body of the EDCs obtained in Section 3.2, which tells us which structural events cause a violation/repair, with the rules from Section 3.1, which

specifies which structural events are implied by the operations. As a result, we obtain some new rules that directly define which operations can violate/repair some constraint.

For instance, if we unfold the previous EDCs with the logic rules that tells that hiring a researcher makes an insertion structural event of a researcher, we obtain:

```
:- hireResearcher(N, S1), hireResearcher(N, S2)
:- hireResearcher(N, S1), Researcher(R2,N,S2), not (del_Researcher(R2,N,S2)),
       R1<>R2
```

Intuitively, the first rule states that two executions of *hireResearcher* can collaborate to rise a constraint violation (i.e., a violation of *ResearcherPk* constraint). The second rule tells us that, *hireResearcher* might be compensated with an operation that deletes researchers. However, since there is no operation to delete researchers, there is no interaction according to this rule.

We now bring an example of a detection of a *compensation interaction*. Consider the EDCs obtained from the *LeaderIsMember* constraint:

```
:- ins_Leads(R,P), del_WorksIn(R,P)
:- ins_Leads(R,P), not (WorksIn(R,P)), not (ins_WorksIn(R,P))
:- Leads(R,P), not (del_Leads(R,P)), del_WorksIn(R,P)
```

Intuitively, the first EDC states that there is a violation if we insert that R is going to lead a project P s/he is leaving. The second asserts a violation if we insert that R is going to lead a project P s/he is not working in and that he is not going to work in. Finally, the third EDC detects a violation if we delete R from working in P, when R is leading P and we do not delete R as a leader of P.

Then, when unfolding the EDCs according to the rules from Section 3.1, which encodes the operations behavior, we have:

```
:- addLeader(RN,PN), Researcher(R, RN, S), Project(P, PN), removeMember(RN,PN)
:- addLeader(RN,PN), Researcher(R, RN, S), Project(P, PN),  not(WorksIn(R,P)),
                not (addMember(RN,PN))
```

Thus, we see that the operations *addLeader* and *removeMember* has a collaboration interaction to violate *LeaderIsMember*, since they both appear positively in the body of a constraint, while *addLeader* and *addMember* has a compensation interaction, since *addMember* appears negatively and *addLeader* positively in the same constraint. Hence, *addLeader* and *removeMember* should be serialized for precondition-time checking techniques, whereas *addMember* and *addLeader* should be serialized (preferribly in this order) for postcontition-time techniques.

4 Implementation

We have implemented our approach in OpExec [8], an artifact-centrict business process model executor. Briefly, this tool is capable of loading the structural and behavioral models of the system at compile time, encoded in logics, and, at runtime, execute the operations invoked by the user into a relational database.

In OpExec, we integrated an implementation of a precondition-time checking technique [19]. This technique assumed that all operations were executed isolatedly, i.e., not concurrently, and thus, required an automatic serialization technique as the one we have discussed in this paper.

The implementation of our technique is summarized in Figure 3. In OpExec, a user loads, in compile time, the structural and behavioral models into a *Controller*. Then, when the user wants to execute the models, the user uses the Controller to create a *ProcessExecutor*. The ProcessExecutor contains an artifactID, which is an id number to identify all the information related to such process. At runtime, the user invokes an operation from the behavioral model through the ProcessExecutor. This processExecutor, then, creates an OperationExecThread, which is a new Thread that will execute the operation invoked by the user into the database.

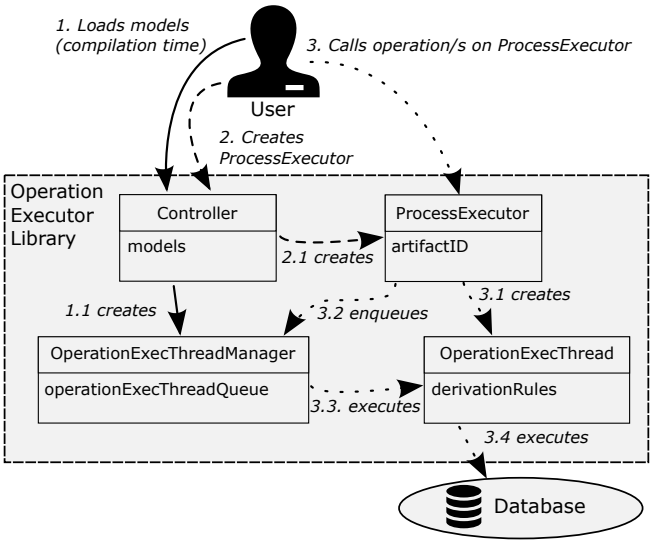


Fig. 3. Architecture of a Model Executor with an Integrity Checking Technique

The integrity checking part is implemented in the *OperationExecThread* which, intuitively, checks whether its structural events are going to violate any constraint according to the current contents of the data. In case that there is any constraint violation, the OperationExecThread does not commit any change into the database, otherwise, the database is updated accordingly.

In order to enable multiple users invoke OpExec concurrently, and to guarantee that the integrity checking part detects all possible violations, we implemented the *OperationExecThreadManager*. When a new OperationExecThread is created, this Thread is enqueued in the OperationExecThreadManager, which is responsible of executing it as soon as it is safe to execute it, i.e., when it is

guaranteed that it will not interact, with any other currently running OperationExecThread, to cause a violation.

The technique discussed in our paper is fully implemented in the OperationExecThreadManager class. That is, at compile time, it receives the models and performs our interaction analysis to detect which operations can collaborate to raise a constraint violation. Then, at runtime, if we try to execute an operation which might interact with another operation which is currently being executed, the OperationExecThreadManager delays the execution of the first untill the second has finished.

Although our implementation is though for a precondition-time integrity checking, we understand that it might not be difficult to adapt it to work with a postcondition-time integrity checking such as those presented in [4,5,6,7].

5 Conclusions

We have presented an approach for adapting integrity checking techniques to systems with concurrent operations. Indeed, current integrity checking techniques do not take into account concurrent operation executions and, as we have seen, this concurrency might cause violations which cannot be detetected by these techniques.

To solve this situation, we have defined an approach for identifying which operations can bring problems to the integrity checking techniques when executed concurrently. As we have seen, the kind of operations that might bring problems depend on the kind of integrity checking technique applied. On the one hand, integrity checking techniques performed at precondition time should avoid concurrent executions of operations that might collaborate to cause a violation. On the other, integrity checking techniques performed at postcondition time should avoid analysing concurrently two operations if one compensates a violation from the other. Our approach can detect both kinds of interactions and thus, can be applied for both kinds of integrity checking techniques. To show the feasibility of our approach, we have implemented it in the OpExec model executor.

References

1. Olivé, A., Cabot, J.: A research agenda for conceptual schema-centric development. In: Conceptual Modelling in Information Systems Engineering. Springer (2007) 319–334
2. ANSI Standard: The SQL 92 Standard. (1992)
3. Oriol, X., Teniente, E.: Incremental checking of OCL constraints with aggregates through SQL. In: Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings. (2015) 199–213
4. Bergmann, G.: Translating ocl to graph patterns. In: Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings. (2014) 670–686
5. Uhl, A., Goldschmidt, T., Holzleitner, M.: Using an OCL impact analysis algorithm for view-based textual modelling. ECEASST **44** (2011)

6. Groher, I., Reder, A., Egyed, A.: Incremental consistency checking of dynamic constraints. In: Fundamental Approaches to Software Engineering. Springer (2010) 203–217
7. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software **82**(9) (2009) 1459–1478
8. De Giacomo, G., Oriol, X., Estañol, M., Teniente, E.: Linking data and BPMN processes to achieve executable models. In: Advanced Information Systems Engineering - 29th International Conference, CAiSE 2017, Essen, Germany, June 12-16, 2017, Proceedings. (2017) 612–628
9. Object Management Group (OMG): Unified Modeling Language (UML) Superstructure Specification, version 2.4.1. (2011) `http://www.omg.org/spec/UML/`.
10. Object Management Group (OMG): Object Constraint Language (UML), version 2.4. (2014) `http://www.omg.org/spec/OCL/`.
11. Combi, C., Oliboni, B., Weske, M., Zerbato, F.: Conceptual modeling of interdependencies between processes and data. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. SAC '18, New York, NY, USA, ACM (2018) 110–119
12. Przigoda, N., Hilken, C., Wille, R., Peleska, J., Drechsler, R.: Checking concurrent behavior in UML/OCL models. In: 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015. (2015) 176–185
13. Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. ACM TOSEM **21**(2) (2012) 13
14. Queralt, A., Teniente, E.: Reasoning on uml conceptual schemas with operations. In van Eck, P., Gordijn, J., Wieringa, R., eds.: Advanced Information Systems Engineering, Berlin, Heidelberg, Springer Berlin Heidelberg (2009) 47–62
15. Cabot, J.: From declarative to imperative uml/ocl operation specifications. In Parent, C., Schewe, K.D., Storey, V.C., Thalheim, B., eds.: Conceptual Modeling - ER 2007, Berlin, Heidelberg, Springer Berlin Heidelberg (2007) 198–213
16. Oriol, X.: Verificació i validació d'esquemes conceptuals uml/ocl amb operacions. (2012)
17. Oriol, X., Teniente, E., Tort, A.: Computing repairs for constraint violations in uml/ocl conceptual schemas. Data & Knowledge Engineering **99** (2015) 39 – 58 Selected Papers from the 33rd International Conference on Conceptual Modeling (ER 2014).
18. Olivé, A.: Integrity constraints checking in deductive databases. In: Proceedings of the 17th Int. Conference on Very Large Data Bases (VLDB). (1991) 513–523
19. Oriol, X., Teniente, E., Rull, G.: TINTIN: a tool for incremental integrity checking of assertions in SQL server. In: Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016. (2016) 632–635