Future Generation Computer Systems 144 (2023) 179–191

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

# Generating valid test data through data cloning

Xavier Oriol *, Ernest Teniente, Marc Maynou, Sergi Nadal

*Universitat Politècnica de Catalunya, Barcelona, Spain*

ARTICLE INFO

*Article history:*
Received 30 July 2022
Received in revised form 18 December 2022
Accepted 22 February 2023
Available online 24 February 2023

Dataset link: https://mydisk.cs.upcedu/s/E8dxK6X9kWam3nN

*Keywords:*
Database testing
Test data
Data cloning

ABSTRACT

One of the most difficult, time-consuming and error-prone tasks during software testing is that of manually generating the data required to properly run the test. This is even harder when we need to generate data of a certain size and such that it satisfies a set of conditions, or business rules, specified over an ontology. To solve this problem, some proposals exist to automatically generate database sample data. However, they are only able to generate data satisfying primary or foreign key constraints but not more complex business rules in the ontology.

We propose here a more general solution for generating test data which is able to deal with expressive business rules. Our approach, which is entirely based on the chase algorithm, first generates a small sample of valid test data (by means of an automated reasoner), then clones this sample data, and finally, relates the cloned data with the original data. All the steps are performed iteratively until a valid database of a certain size is obtained. We theoretically prove the correctness of our approach, and experimentally show its practical applicability.



## 1. Introduction

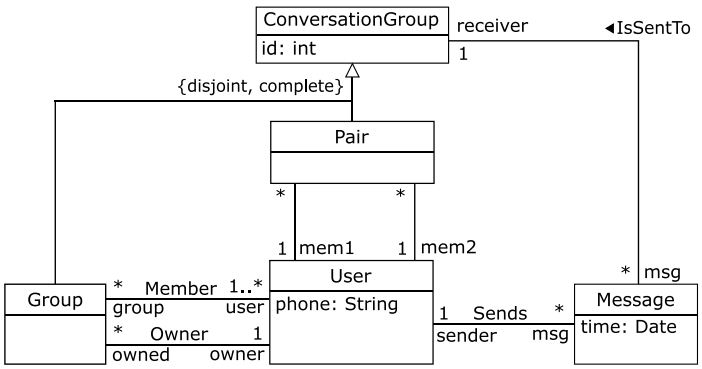
Data is an essential component of a software system. During software testing, it is crucial to feed the system with some sample data to be able to simulate the execution of the system in a real environment. Without proper test data, it is almost impossible to ensure the correct behaviour of the system once deployed [1]. Test data is often generated manually or through ad-hoc programmed functions. This entails significant economic and time costs, as well as being an error-prone activity. This manual generation also makes practically impossible for the data being manually created to be consistent with the business rules and the constraints of the system [2], not to say when the data is provided with some ontological layer, such as in the context of ontology-based data access [3], where conditions might become arbitrarily complex. This is because the business rules of a software system use to involve very complex data implications that can hardly be captured solely by human intervention.

As an example, consider the UML [4] schema for a message application shown in Fig. 1. Shortly, *Users* send *Messages* that are received by *ConversationGroups*, which might be *Groups*, or *Pairs*. A *Pair* is a pair of two users, while a *Group* has several members and a unique owner. We assume that *Groups*, *ConversationGroups*, and *Pairs* are identified by id, *Users* by phone, and *Messages* by sender, receiver, and data. Besides the cardinality constraints, the schema contains also four business rules stated as SQL assertions [5] which correspond to conditions that must always be satisfied in the domain represented by the schema and that, thus, must hold for the data to be consistent. These SQL assertions avoid users making pairs with themselves, owning groups they do not belong to, or sending messages to conversation groups they are not part of. The practical importance of using business rules in software development, whether specified as SQL assertions or through any other language, is highlighted in several blogs, in stack overflow or GitHub repositories (see for instance, [6–8]).

To generate sufficiently large test data for scenarios like our running example, constraints and business rules in the schema must be taken into account to ensure they are satisfied by the generated data. Hence, software engineers have to develop ad-hoc methods that generate the data in such a way that reach the desired size without violating any of these rules. This is well-known to be a complex and arduous task which makes close to impossible for a software engineer to manually code this program. Current proposals to automatically generate test data satisfying business rules do not serve our purposes. Indeed, previous proposals either allow only to generate a small sample instances of data satisfying the conditions [9–11], or are able to generate a large set of data but considering only very limited database constraints on it (i.e., primary/foreign key constraints and attribute checks) [12,13]. This limitation is rooted in the fact that, when considering general business rules, the traditional approach is based on relying in a SAT solver, whose complexity is NP-hard if we limit the search scope to some boundary, or even undecidable when not limiting it at all.

* Corresponding author.
*E-mail addresses:* xoriol@essi.upc.edu (X. Oriol), teniente@essi.upc.edu (E. Teniente), marc.maynou@upc.edu (M. Maynou), snadal@essi.upc.edu (S. Nadal).

https://doi.org/10.1016/j.future.2023.02.020




```
ASSERTION MembersAreNotTheSame
 CHECK( NOT EXISTS(
   SELECT "Pair with itself"
   FROM PairMem1 PM1 JOIN
     PairMem2 PM2 ON PM1.id=PM2.id
   WHERE PM1.mem1 = PM2.mem2 ))

ASSERTION MessageToPairIsFromMember
 CHECK( NOT EXISTS(
  SELECT "Message not from a member"
  FROM Message M
   JOIN Pair P ON (M.receiver = P.id)
   WHERE M.sender <> P.mem1
   AND M.sender <> P.mem2))
```

```
ASSERTION OwnerIsMember
 CHECK( NOT EXISTS(
  SELECT "Owner who is not member"
  FROM Group G
  WHERE G.owner NOT IN
   (SELECT M.user FROM Member M)))

ASSERTION MessageToGroupIsFromMember
 CHECK( NOT EXISTS(
  SELECT "Message not from a member"
  FROM Message M
   JOIN Group G ON M.receiver = G.id
   WHERE M.sender NOT IN
    (SELECT Mem.user
     FROM Member Mem
     WHERE Mem.groupId = G.id)))
```

**Fig. 1.** Application schema with constraints and business rules expressed as SQL assertions.

The main goal of this paper is to overcome the previous limitations by proposing an approach to automatically generate a large set of representative test data satisfying a given set of expressive business rules defined in a certain database schema. The test criterion for assessing the validity of the generated data is that no business rule will be violated when this data is inserted in the database.

Our approach is agnostic as far as the language used for expressing the business rules, provided that the language is first order (i.e., we accept business rules written in SQL, UML/OCL, SPARQL, Datalog, or OWL). This is because, in the core, our method relies on data dependencies such as tuple-generating dependencies (TGDs), disjunctive embedded dependencies (DEDs), equality-generating dependencies (EGDs), and denials. It is well known in the literature how to translate business rules specified in such languages into this formalism [14,15]. The independence of our approach from the language used to specify business rules makes it usable for different kinds of data platforms, and in particular for those enabling data-intensive storage.

In a nutshell, our method, works as follows. Assuming an initial small sample set of data satisfying the business rules, which can been obtained for instance by applying an existing automated reasoner [9–11], the core idea under our approach is that of extensively cloning parts of this initial database to obtain a large set of representative data. In particular, the approach has two main phases: the cloning phase, and the merging phase. The former is intended to grow the initial database instance while keeping the consistency of the data. To achieve it, whenever part of this cloned data does not satisfy some of the business rules, we also clone the necessary data from the initial sample to repair this violation. The merging phase is intended to connect the newly created data during the clone phase, with the original data from the database. Indeed, without this step, we would only end up with a large database of unconnected facts. During this step, we merge some of the cloned data with the original data they are cloned from. To do so, we have to additionally analyse which data can be merged without incurring into a violation of some business rules.

As we later show, both processes can be implemented with a chase algorithm. Indeed, roughly speaking, cloning can be achieved by means of TGDs, and merging can be achieved by means of EGDs. Furthermore, considering that scalability is paramount for our objective of achieving large test databases, we show that our approach can run in linear time with regards to the desired size of the final database. Intuitively, this is because we can run our approach of cloning and merging in small chunks of a fixed size, where each chunk takes some constant time to execute. Hence, the final execution time is linearly proportional to the number of chunks executed to achieve the desired database size.

It is worth mentioning that the test data we generate are valid with respect to a given schema and set of business rules. Thus, if any of these is modified we should execute again our algorithm in order to obtain valid test data for the new schema and rules. Nevertheless, this does not pose a significant problem because of the efficiency of our proposal. The new data obtained is the one that should be inserted within the database and used for testing with the new schema.

Summarizing, the main contributions of our approach are the following:

- We propose an approach to automatically generate a large set of representative test data satisfying a set of expressive business rules.

- We formally proof the correctness of the approach, and its linear data-complexity.
- We provide experiments showing that the run time of our approach beats current alternatives from the literature, in essence, by experimentally showing that our approach runs linearly whereas SAT solving alternatives grows exponentially.
- We provide experiments analysing in detail how the different parameters of our cloning method affect its performance.

Throughout the paper we will provide all details and reproducibility instructions of the experiments we have performed.

**Outline.** The rest of the paper is structured as follows. Next, Section 2 reviews preliminary concepts. Section 3, presents our approach to clone the initial sample set of data. Section 4, discusses on the characteristics that a chase algorithm should satisfy in order to be applicable to our approach. In Section 5, we experimentally validate our approach. Section 6 reviews related work. Finally, we summarize our conclusions and point out future work in Section 7. Our paper is complemented with an extended running example an evaluation on a real-world scenario in A.

## 2. Preliminaries

**Terms, and atoms.** 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., $p(t_1, \ldots, t_n)$. 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* (and to be an instance of $p$). We distinguish a special (infinite) set of constants we refer as *labelled nulls*, which we denote #0, #1, #2, ...

**TGDs, DEDs, Denials, and EGDs**. A tuple-generating dependency (TGD) is a rule of the form $\forall\bar{x}.\phi(\bar{x}) \rightarrow \exists\bar{y}.\psi(\bar{x},\bar{y})$, where $\phi$ and $\psi$ are conjunctions of atoms. $\phi$ is said to be the left-hand side (LHS) of the rule, and $\psi$ to be its right-hand side (RHS). The variables appearing in the LHS ($\bar{x}$) are referred as *universal*, whereas those only appearing in the RHS ($\bar{y}$) are referred as *existential* variables. Disjunctive embedded dependencies (DEDs) are a natural extension of TGDs where the RHS of the rule can contain disjunctions. I.e., a DED is a rule of the form: $\forall\bar{x}.\phi(\bar{x}) \rightarrow \bigvee_{i=1..m} \exists\bar{y_i}.\psi_i(\bar{x},\bar{y_i})$. Denials are a natural extension of DEDs where the number of disjunctions is 0. I.e., a denial is a rule of the form: $\forall\bar{x}.\phi(\bar{x}) \rightarrow \bot$. Intuitively, a denial expresses a condition that should never hold in the database. An equality generating dependency (EGD) is a rule of the form: $\forall\bar{x}.\phi(\bar{x}) \rightarrow x_1 = x_2$ where $\phi$ is a conjunction of atoms and $x_i \in \bar{x}$. We refer to any TGD, DED, Denial and EGD to be a constraint. Hereinafter, we might omit quantifiers as they can be understood by context.

**Substitution.** A *substitution* $\theta$ is a set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$ where each variable $x_i$ is unique. The domain of a substitution is the set of all $x_i$ and is referred as $dom(\theta)$. We say that $\theta$ is ground if every $t_i$ is a constant. The atom $a\theta$ is the atom resulting from simultaneously substituting any occurrence of $x_i$ in $l$ for its corresponding $t_i$. We define the conjunction $\phi\theta$ as the conjunction resulting from simultaneously applying the substitution $\theta$ to all the atoms in $\phi$.

**Databases, and database consistency.** A database $D$ is a set of ground atoms we call facts. We say that a ground atom $a$ is true in $D$ iff $a \in D$, and denote it as $D \models a$. Similarly, we say that a negated ground atom $a$ is true in $D$ iff $a \notin D$. We extend this notion of satisfaction to conjunctions and, ultimately, to constraints. That is, a constraint $c$ is satisfied in $D$ (noted as $D \models c$) iff for all substitutions of universal variables that makes its LHS true, there exists a substitution for its existential variables that makes the RHS true. Given a set of constraints, we say that $D$ is consistent if it satisfies all of them. It is worth noting that, in the case of denials, if there is a substitution for the universal variables that makes the LHS true, the denial is violated.

**Chase.** The chase is an algorithm for deducing new facts from previous facts according to the constraints. In essence, it consists in, whenever some TGD $c$ is violated into some database $D$, because of some substitution $\theta$ for its universal variables, instantiate the RHS of the rule with $\theta\sigma$, where $\sigma$ is a substitution from the existential variables into new fresh labelled nulls. Whenever an EGD is violated into $D$, because of some substitution $\theta$ for its universal variables, where $x_1\theta \neq x_2\theta$, there are two possibilities: (1) if some $x_i\theta$ is a labelled null, the chase replaces any appearance of $x_i\theta$ for $x_j\theta$ in $D$; (2) if both are constants, the chase terminates without finding any solution. Similarly, if a denial is violated when chasing some database, the chase returns no solution. Finally, DEDs makes the chase execution to create a search space for finding a solution. Indeed, each disjunction represents a different way that the chase can apply to repair the violation. Intuitively, DEDs makes the chase to create several chase-branches (one for each disjunction in the RHS of the DED) to find a solution, whereas Denials and EGDs cuts some of these branches. We refer the reader to [16] for an in-depth introduction to the chase algorithm.

**Homomorphism.** Given two databases $D_1$ and $D_2$, an homomorphism from $D_1$ to $D_2$ is a function $h$ from constants in $D_1$ into constants from $D_2$ s.t.: (1) every constant that is not a labelled null is mapped into itself. That is, only labelled nulls might be mapped to other constants different than themselves; (2) if an atom $a(c_1, \ldots, c_n)$ exists in $D_1$, $a(h(c_1), \ldots, h(c_n))$ appears in $D_2$.

## 3. Our approach

Our goal is to obtain a large database from an original small one, where all the constraints are satisfied. We assume that the original database already satisfies the constraints. This can be achieved by means of an automated reasoner (e.g. [9–11]), or by manually preparing the data. We also assume that the constraints to satisfy are given under the form of TGDs, DEDs, EGDs, and denials, which are known to be expressive enough to deal beyond typical dependency database constraints (e.g., unique constraints, foreign keys, etc.). These TGDs, DEDs, EGDs and denials can be automatically translated from any first-order language (e.g., a translation from SQL can be achieved following [15]).

The key idea underlying our approach is to clone small parts of the original database to obtain a larger one. In particular, we clone some initial parts, originally requested by the user,[1] and then, if those new cloned parts violate some constraint, we clone the necessary data from the original database to repair such violation. Revising our running example, assume that we aim to clone some *Group* $g$ into another *Group* $g'$. There is a constraint forcing each group to have at least one member. Then, we will not only clone $g$ into $g'$, but also clone some member $u$ from $g$ into $u'$, and make $u'$ be a member of $g'$. Hence, we ensure that $g'$ also has one member, and avoid violating the minimum cardinality constraint.

In this way, we ensure that the clones satisfy the constraints, however, in order to obtain a more representative database, we apply a process of merging some of the cloned data, i.e., equating its values. Indeed, if we only cloned data, we would end up with a big database of unrelated facts, that is, data that cannot be joined (e.g., following the previous example, we would only be creating new groups, with totally new members, which would never be related with the original database). The process of merging is aimed

[1] This initial parts can be selected also randomly if the user has no requirement. Our approach is agnostic from where the original requests for cloning comes from.

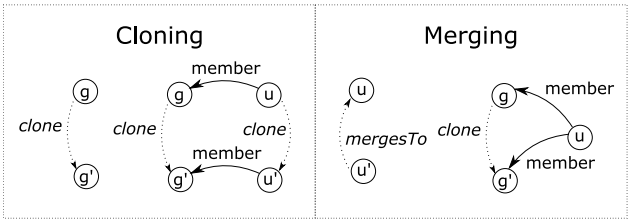



**Fig. 2.** Cloning and Merging phases over the running example.

at relating cloned facts with the original facts of the database, so that we can join the original data with the cloned one. For instance, in the process of merging, we can make $u'$ to be equal to $u$, making the groups $g$ and $g'$ share some member. All this process is summarized in Fig. 2.

To merge cloned pieces of data, we must be careful with the constraints. Certainly, merging data might alter the evaluation of a constraint, thus, potentially violating it. Indeed, imagine that there is a constraint forcing each user to belong to 1 group only. This constraint makes impossible to merge the previous users $u'$ and $u$. To avoid such situation, we first compute the constants that are unmergeable due to a potential violation. Once the unmergeable data is computed, we can merge the rest of data ensuring the consistency of the final database.

All processes (i.e., cloning, computing unmergeable facts, and merging) can be run with a chase algorithm. Indeed, we can encode all such processes with TGDs and EGDs. Intuitively, the cloning process can be accomplished by compiling the initial constraints into TGDs/EGDs that clones data and their required repairs to avoid violations. The process of computing unmergeable facts can be accomplished by compiling the initial constraints into a set of TGDs/DEDs that derives which facts cannot be merged. Finally, the merging process consists in applying an EGD rule that merges cloned data in case there is no unmergeable issue. Fig. 3, depicts the high-level overview of our pipeline. It is worth to remark that, since our approach relies on the chase algorithm, the output might contain *labelled nulls*. However, such labelled nulls can be postprocessed, and replaced for real values, taken from a dictionary or randomly generated, with the condition that different labelled nulls must be replaced for different values.

On the following, we revisit all the processes separately. For each of them, we describe its rational, how to obtain the TGDs/EGDs rules, and argue its correctness by proving that, at the end of each process, every constraint is satisfied.

**Constraints prerequisites** For our purposes, we require the TGDs, DEDs and EGDS not to contain any cartesian product in its LHS. That is, for any given two atoms $a_1$ and $a_n$ from the LHS of any TGD/DED, there is a list of atoms $a_1, a_2, \ldots, a_n$ such that, each $a_i$ belongs to the LHS of the rule, and between any $a_i$ and $a_{i+1}$ there is, at least, one variable in common. In practice, this limitation affects the cardinality constraints of n-ary associations ($n > 2$). Indeed, in modelling languages such as UML, you can specify a constraint such as: *for any pair of user and group, there is, at lest, one message*. This kind of constraints, which are note very frequent, cannot be dealt within our approach.

### 3.1. Cloning

The rationale behind the cloning process is twofold: on the one hand, we must clone all the data requested by the user to clone, and on the other, we must clone, recursively, all the data required to satisfy the constraints of the database.

This phase is the unique one that expands the data of the original database. As we are going to see, the size of the finally obtained database is $x * k$, where $x$ is the number of clones requested by the user, and $k$ a constant determined by the schema and the current data. As a result, the user of our approach can control the size of the final output by means of increasing $x$ (which will cause a linear increase in the final output).

#### 3.1.1. Cloning user request

For the first purpose, we start, for each predicate $p$ with arity $n$, creating a TGD such as the following:

$$p(a_1 \ldots a_n), PcloneReq(a_1 \ldots a_n, a'_1 \ldots a'_n) \rightarrow cloneP(a_1 \ldots a_n, a'_1 \ldots a'_n).$$

Intuitively, this rule says that, if we have an atom $p(a_1, \ldots, a_n)$, and the user requests to clone $p$ (represented in the rule with $PcloneReq(a_1 \ldots a_n, a'_1 \ldots a'_n)$), then, we have to clone $p$ with the same attributes. When we have to clone some tuple into another, we must create the new tuple, and remember that each attribute is a clone from the original one. This can be achieved creating, for each relation *cloneP* predicate, the following TGD:

$$cloneP(a_1 \ldots a_n, a'_1 \ldots a'_n) \rightarrow p(a'_1, \ldots, a'_n), clone(a_1, a'_1), \ldots, clone(a_n, a'_n)$$

For ease of presentation we might use the predicate $cloneAll(\overline{a}, \overline{a}')$ to refer to $clone(a_1, a'_1), \ldots, clone(a_n, a'_n)$. Then, the previous rule is summarized as:

$$cloneP(\overline{a}, \overline{a}') \rightarrow p(\overline{a}'), cloneAll(\overline{a}, \overline{a}')$$

For instance, given the predicate *user*, we will build the following TGDs:

$$user(u), UserCloneReq(u, u') \rightarrow cloneUser(u, u').$$
$$cloneUser(u, u') \rightarrow User(u'), clone(u, u').$$

#### 3.1.2. Cloning repairs

For the second purpose we must clone, for each violation occurred within the cloned data, the corresponding repair for the original data. Note that not all constraints have repairs (i.e., denials). However, as we are going to see, denials will not suppose any problem with our technique. There are three kinds of constraints with repairs: (1) TGDs, (2) DEDs, (3) EGDs. We review each case separately.

**TGD constraints.** For each TGDs of the form: $body(\overline{x}, \overline{y}) \rightarrow head(\overline{y}, \overline{z})$ where *body* and *head* are conjunctions of atoms, we build the cloning TGD:

$$body(\overline{x}', \overline{y}'), cloneAll(\overline{y}, \overline{y}'), head(\overline{y}, \overline{z}) \rightarrow cloneAll(\overline{z}, \overline{z}'), head(\overline{y}', \overline{z}')$$

Intuitively, the rule says that, when the cloned data satisfies the body of the original TGD (and hence, might violate the constraint), we check which is the head of the original data we have cloned, and we clone it. E.g., the min cardinality constraint stating that each group has, at least, one member (which is encoded as the TGD $Group(g, o) \rightarrow Member(g, u)$), brings the cloning TGD:

$$Group(g', o'), clone(g, g'), clone(o, o'), Member(g, u) \rightarrow clone(u, u'), Member(g', u')$$

**DED constraints.** For each DED of the form: $body(\overline{x}, \overline{y}) \rightarrow head_1(\overline{y}, \overline{z_1}) \vee \ldots \vee head_n(\overline{y}, \overline{z_n})$ we build $n$ TGDs with the form:

$$body(\overline{x}', \overline{y}'), cloneAll(\overline{y}, \overline{y}'), head_1(\overline{y}, \overline{z}) \rightarrow cloneAll(\overline{z}, \overline{z}'), head_1(\overline{y}', \overline{z}')$$
$$\ldots$$
$$body(\overline{x}', \overline{y}'), cloneAll(\overline{y}, \overline{y}'), head_n(\overline{y}, \overline{z}) \rightarrow cloneAll(\overline{z}, \overline{z}'), head_n(\overline{y}', \overline{z}')$$

Again, the rationale behind this TGDs is to copy the repair from the original data into the cloned data. With this purpose, there is one TGD for each possible way to repair the constraint. E.g., the complete hierarchy constraint stating that each conversation group is a group or a pair (which is encoded as the DED

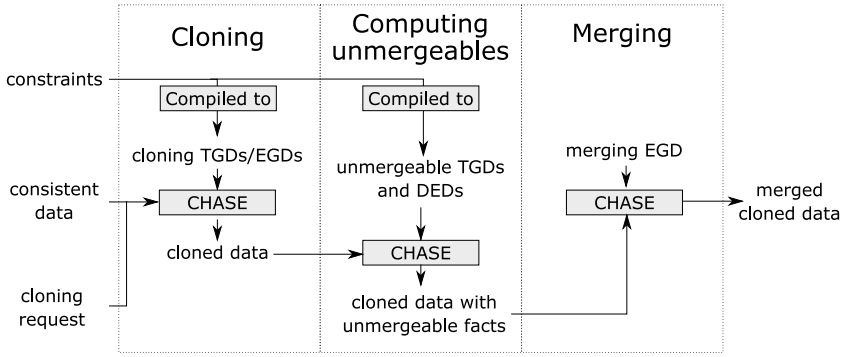



**Fig. 3.** Cloning pipeline.

$ConversationGroup(g) \rightarrow Group(g) \vee Pair(g)$) brings the cloning TGDs:

$ConversationGroup(g'), clone(g, g'), Group(g) \rightarrow Group(g')$.

$ConversationGroup(g'), clone(g, g'), Pair(g) \rightarrow Pair(g')$.

**EGD constraints.** For each EGD of the form: $body(\overline{x}) \rightarrow x_i = x_j$ we maintain, as it is, in the cloning TGDs. These EGDs are mandatory to ensure that, if cloning twice the same repair $r$ for an entity $e$, and such $e$ have a functional dependency with $r$, the two clones of $r$ are the same (and thus, we maintain the functional dependency in the cloned data). E.g., the EGD stating that, fixed a Pair $g$, there is only 1 user playing the role of the 1st member (which is encoded as $PairMem1(g, u), PairMem1(g, u2) \rightarrow u = u2$) is added in the set of cloningTGDs.

*3.1.3. Justification of the correctness*

When chasing the previously defined rules we obtain a new database that satisfies all the constraints defined, whether they have a repair or not. The intuition is that, since the original data satisfies all the constraints, only the cloned data could violate them. However, the cloned data comes from the original data, which means that, if the cloned data is inconsistent, the original data would also be inconsistent.

**Theorem 1.** *Let* cloningTGDs(C) *be the set of cloning TGDs from an original set of constraints C. Given a database D, and a set of constraints C s.t. $D \models C$, and some clone request R, we have: after chasing the original database D with cloningTGDs(C) we obtain a new database $D'$ s.t. $D' \models C$.*

**Proof.** The proof is based on two steps. First, we build an homomorphism from the labelled nulls into the constants. In the second step, we use such homomorphism to show that no constraint is violated in $D'$.

Step (1) Consider the function *original* from labelled nulls into constants defined as:

$original(a') = a$ **iff** $clone(a, a')$

Such function, intuitively, retrieve the original data where some labelled null is cloned from. By construction, *original* is an homomorphism: its functional (every labelled null is mapped, at most, to one constant), its total (every labelled null is mapped, at least, to one constant), and if $p(\overline{x'})$ is true, $p(original(\overline{x'}))$ is also true. To realize the last point, note that, in any RGD, we only derive $p(\overline{x'})$ if $p(original(\overline{x'}))$ appears in the LHS of the TGD.

Step (2) We now show, exploiting the homomorphism *original*, that no constraint is violated in $D'$. EGDs are not violated because they are executed along the chase. So, the unique constraints that might have been violated are TGDs, DEDs, and denials. We start showing that no denial is violated, and then, move to TGDs and DEDs.

To show that no denial is violated, we use a proof by contradiction, hence, we assume that, for some denial $c \in C$, we have $D' \not\models c$. Since $D' \not\models c$, we have that there is a substitution $\sigma$ from the variables in $c$ into constants such $D' \not\models c\sigma$. Now, consider $\sigma_o$ to be the substitution $\sigma$ after applying the *original* homomorphism to its labelled nulls. We have $D \not\models c\sigma_o$ (contradiction).

To show that no TGD/DED is violated, we use a proof by contradiction, hence, we assume that, for some tgd/ded $c \in C$, we have $D' \not\models c$. Since $D' \not\models c$ we have that there is a substitution $\sigma$ from the variables of the LHS in $c$ into constants such $D' \not\models c\sigma$. Since there is no cartesian product in tgd/ded, and by construction, no labelled nulls appears with a constant in any atom (in exception of *clone*, which does not appear in any $c$), either $\sigma$ maps all variables into constants, or maps all variables into labelled nulls, but does not mix them. If $\sigma$ maps all variables into constants, there is a violation in the original database $D$ (contradiction). Otherwise, consider $\sigma_o$ to be the substitution $\sigma$ after applying the *original* homomorphism to its labelled nulls. Since $D' \not\models c\sigma$, we have that $D' \models LHS(c\sigma)$ (where $LHS(c)$ represents the LHS of $c$). Then, by the homomorphism, we have $D \models LHS(c\sigma_o)$. Hence, since $D$ is consistent, $D$ has repaired $C$ when it is triggered by the values $\sigma_o$. Therefor, by construction, the cloning TGD corresponding to $c$ has created the values $RHS(c\sigma_o)$, which would repair the violation (contradiction). The case for deds follows analogously. □

Apart from being correct, it is important to show that this chase terminates. Intuitively, the chase terminates because we only clone repairs from the original database and, since the original database is finite, cloning repairs from it is also finite. Formally:

**Theorem 2.** *Let* cloningTGDs(C) *be the set of cloning TGDs from an original set of constraints C. Given a database D, and a set of constraints C s.t. $D \models C$, and some clone request R, chasing D with R and* cloningTGDs(C) *terminates, provided that the chase is complete (such as the core chase [16]).*

**Proof.** Assume that we have $n$ clone requests. Clearly, cloning $n$ times the original database satisfies the request (i.e., for each tuple $p(\overline{t})$ from $D$, write $p(\overline{t'})$ in the cloned database together the facts $cloneAll(\overline{t}, \overline{t'})$). Lets call each of these cloned databases $D_1, \ldots, D_n$. Clearly, $D_1 \cup \cdots \cup D_n$ is a solution when chasing *cloningTGDs(C)* with $D$ and $R$, and it is finite. By definition, a complete chase would find such solution (or a subset of it, in case a subset of it is also a solution). □

### 3.2. Computing unmergeable nulls

The idea here is to compute which labelled nulls cannot be merged into its original constants. Then, in the next step, we can merge those labelled nulls which are not unmergeable.

There are two possible reasons for not merging a labelled null into its original constant: (1) a user request, (2) merging them might encompass a constraint violation. We review both situations separately.

#### 3.2.1. Unmergeable because of a request

When a user requests to clone some tuple $p(\overline{a})$ into $p(\overline{a}')$, we should not merge $\overline{a}$ with $\overline{a}'$ to satisfy the user request of having a copy of $p$. Consequently, we need the following TGD for computing unmergeable nulls:

$$PcloneReq(a_1 ... a_n, a'_1 ... a'_n) \rightarrow unmergeable(a'_1), \ldots, unmergeable(a'_n)$$

For instance, in our running example, we would create the unmergeable rule for users: $UserCloneReq(u, u') \rightarrow unmergeable(u')$.

#### 3.2.2. Unmergeable because of a constraint violation

When merging a null with its corresponding true constant, it might be the case that we violate some constraint. To avoid the violation, the key idea consists in keeping isolated the data from the cloned data w.r.t. the constraints. For instance, consider the following EGD:

$$p(x, y), p(y, z) \rightarrow y = z$$

We know that, by precondition, when just considering the constants from the original database $D$, the EGD is satisfied. Furthermore, when just considering the labelled nulls from the cloned data $D'$, the EGD is still satisfied. However, when merging some labelled null with some constant, we might violate the constraint. E.g., consider that $D = \{p(1, 2)\}$, which satisfies the constraint, together the cloned data $D' = \{p(\#1, \#2)\}$, which also satisfies the constraint. Note that, if merging #1 into 1, we would get a violation with the values $p(1, 2), p(1, \#2)$ since we have not equated #2 with 2. Naturally, we could repair the EGD by means of equating them, but this way to repair, if carried without control, might end up equating the whole set of labelled nulls with their corresponding original constants, thus, eliminating all the generated clones. For this reason, we advocate not to repair constraints on the unmergeable computation step (which is a process carried on the cloning step), but to avoid the violation instead. For a similar reason, we should avoid violating TGDs and DEDs.

To avoid such violations, the idea is to not merge a labelled null that might be used in a join. Hence, in our previous example, we would generate the following unmergeable TGD rule: $p(x, y), clone(x, x'), p(x', z') \rightarrow unmergeable(x')$. More in general, for each EGD, TGD, or DED $c$ we must proceed as follows: for each pair of atoms $p(\overline{x_p})$ and $q(\overline{x_q})$ from $LHS(c)$, which have some variables in common $\overline{x_c} \subset \overline{x_p} \cap \overline{x_q}$ (with $x_c \neq \emptyset$), we must create the following DEDs:

$$p(\overline{x_p}), cloneAll(\overline{x_c}, \overline{x_c}'), q(\overline{x'_q}) \rightarrow unmergeable(x'_1) \vee \ldots \vee unmergeable(x'_n)$$
$$q(\overline{x_q}), cloneAll(\overline{x_c}, \overline{x_c}'), p(\overline{x'_p}) \rightarrow unmergeable(x'_1) \vee \ldots \vee unmergeable(x'_n)$$

where $x'_i \in \overline{x_c}'$.

In our running example, the maximum cardinality constraint stating that one group can only have one owner (which is encoded with the following EGD: $Group(g, o), Group(g, o2) \rightarrow o = o2$) brings the unmergeable TGD: $Group(g, o), clone(g, g'), Group(g', o2) \rightarrow unmergeable(g')$ meaning that we cannot merge groups since this might make a group to have two owners, and hence, violate the maximum cardinality constraint.

#### 3.2.3. Justification of correctness

This step does not compute any fact that might violate any constraint. Indeed, we only compute unmergeable labelled nulls. Also, this step terminates, since the number of labelled nulls that might be indicated to be unmergeable is finite.

### 3.3. Merging

The merging process is based on, by means of EGDs, equating some labelled nulls into their original constants. In this way, the cloned data joins the original data. In order to ensure that no merge violates any constraint, we only merge those clones which are not unmergeable. Hence, as a result, we obtain a new bigger database, satisfying the clones requested by the user, where the original data joins with the cloned data, but in such a way that all constraints are satisfied.

Formally, the merge process contains one unique EGD rule:

$$clone(x, x'), not(unmergeable(x')) \rightarrow x = x'$$

Note that, in this case, in the LHS of the rule, we have a negated atom. Hence, we need a chase capable of dealing with basic negation, and apply the chase in a stratified manner (i.e., we cannot start chasing those rules that depends on the absence of some atom $a$, until all the dependencies that might generate $a$ instances have been executed). It is not difficult to see that our approach is already well-stratified if applying the chase in the order we have presented the paper: first the rules for computing clones, then the rules for computing unmergeables, and finally the rules for computing merges.

#### 3.3.1. Justification of correctness

Intuitively, during the process of merging, the homomorphism between the clones and the original data is preserved. Then, since no constraint is violated in the original database, no constraint can be violated with the cloned data. Furthermore, since the labelled nulls never joins the constraints in the joins relevant for the constraints, no constraint is violated in the union of the data with the cloned data. Formally:

**Theorem 3.** *Let* mergingEGDs(C) *be the set of merging EGDs from an original set of constraints C. Given a cloned database $D'$ from D, and a set of constraints C s.t. $D \models C$, we have: after chasing the original database D with mergingEGDs(C) and $D'$, we obtain a new database $D^m$ s.t. $D^m \models C$.*

**Proof.** Since $D'$ is a clone from $D$, there is an homomorphism *original* from $D'$ into $D$. Furthermore, it is easy to see that the homorphism is preserved during the merge. As a result, there is an homomorphism between $D^m$ and $D$. Making an abuse of notation, we might still call it *original* (indeed, its just the same homomorphism but projecting the labelled nulls that disappear from $D^m$).

Similarly to Theorem 1, no denial is violated in $D^m$ because of the homomorphism to $D$, which is consistent. We lack proving that no TGD/DED/EGD is violated. We prove that no TGD is violated since the other cases follows analogously. The proof is based on contradiction. Assume that there is some TGD $c \in C$ that is violated in $D^m$. Consider the substitution $\sigma$ for the variables in $LHS(c)$ that witnesses such violation. There are two cases: $c$ has only one literal on its LHS, or it has many.

When there is one literal, we know $D^m \models LHS(c\sigma)$. Because of the homomorphism, there is also some $\sigma_o$ s.t. $D \models LHS(c\sigma)$. Hence, there is a repair for such violation in $D$. Such repair appears, by construction, in $D'$. Since there is an homomorphism from $D'$ into $D^m$, such repair also appears in $D^m$. Hence, $D^m \models c$ (contradiction).

When there are more literals, clearly $\sigma$ cannot be composed only of constants (since it would show a violation in $D$, which is consistent), nor can be only composed of labelled nulls (since it would point a violation in $D'$, which is also consistent). Hence, it must mix constraints and labelled nulls. Since, by assumption, $c$ has no cartesian product, pick any two literals $p$ and $q$ that have a join on the variables $x_c$ where $p\sigma$ does not contain any labelled null, and $q\sigma$ contains some labelled null. It is clear that $p\sigma$ and $q\sigma$ share the same values for $\overline{x_c}$ (i.e., those established in $\sigma$). However, by construction, for some $x_i \in \overline{x_c}$, their values cannot be the same (they could not be merged)(contradiction). □

## 4. Chase characteristics

In this section, we go deeper into analysing the chase algorithm that we can use to run our approach. As previously said, we require our chase-algorithm to be complete in order to ensure that we always find a solution. On the following, we argue that our approach can be parallelized (running several chase instances at the same time), and that it can run non-deterministically, which is a property that can be interesting for generating random test data. Interestingly, our approach can run each clone request isolatedly. That is, if we have 10 clone requests, we can run each clone request in a different chase instance, with the same original database, and finally, merge all the results in a single output database. The rationale behind this is that each clone requests generates a group of clones (of the original database) that is independant with regards to the clones generated by the other clone requests.

In practice, this means that (1) our proposal can run in linear time w.r.t. number of clone requests, and (2) our proposal can be easily parallelized. Indeed, the complexity of running a single clone request is determined by the original database schema and database size, which are fixed parameters, hence, having a constant execution time $k$. Therefore, running $x$ clone requests is expected to take $x * k$ time. With regards to the parallelization, since each clone requests is independant from the others, they can be run in parallel, provided that each parallel execution uses a different (infinite) set of labelled nulls to avoid a non-intended clash of nulls. Formally, the following theorems state that our approach is sound and complete when run in parallel.

**Theorem 4** (*Soundness*)**.** *Given some constraints $C$, a database $D$ s.t. $D \models C$, and two databases $D_1$ and $D_2$ obtained by means our approach over some clone requests $Req_1$ and $Req_2$ (respectively), using two different sets of labelled nulls, the union $D_1 \cup D_2$ is consistent ($D_1 \cup D_2 \models C$) and satisfies the requests $Req_1 \cup Req_2$.*

**Proof.** We need to proof two things: (1) $D_1 \cup D_2$ satisfies the request. (2) $D_1 \cup D_2$ is consistent. We do so separately. $D_1 \cup D_2$ satisfies the request because, indeed, to satisfy the request, we only need to contain the requested clone. Since $D_1 \cup D_2$ subsumes both $D_1$ and $D_2$, and each $D_i$ contains the request $Req_i$, then, $D_1 \cup D_2$ satisfies the request also. To see that $D_1 \cup D_2$ is consistent, it is enough to realize that there is an homomorphism from $D_1 \cup D_2$ into $D$. Indeed, by construction there is an homomorphism $h_1$ from $D_1$ into $D$, and another homomorphism $h2$ from $D_2$ into $D$. Then, $h_1 \cup h_2$ is an homomorphism from $D_1 \cup D_2$ into $D$ thanks to the fact that $D_1$ and $D_2$ do not have any labelled null in common. Once the homomorphism is established, the proof continues in a similar fashion as the consistency proof from Theorem 1. □

**Theorem 5** (*Completeness*)**.** *Given some constraints $C$, a database $D$ s.t. $D \models C$, and a clone request $Req = Req_1 \cup Req_2$ (with $Req_i \neq \emptyset$), we can run our approach in parallel, using two different sets of labelled nulls, obtaining two databases $D_1$ and $D_2$ s.t. the union $D_1 \cup D_2$ is consistent ($D_1 \cup D_2 \models C$) and satisfies the requests $Req_1 \cup Req_2$.*

**Proof.** By precondition, it is always possible to partition the request $Req$ into two different sets $Req_1$ and $Req_2$. Running our approach in parallel, in each $Req_i$, will bring a solution, for sure, since our approach always finds a solution (provided that the chase is complete). By Theorem 1, we know that the union of both solutions is a solution for $Req_1 \cup Req_2$. □

As a consequence, we have that our approach runs in linear time:

**Theorem 6.** *Given some constraints $C$, a database $D$ s.t. $D \models C$, and a clone request $Req$, we can run our approach in linear time wrt the size of $Req$.*

**Proof.** By Theorems 4 and 5 we can reduce the problem of running our approach over $Req$ into running $|Req|$ times our approach. In each time, the size of the problem is fixed: indeed, the size of the schema is fixed, the size of the original database $D$ is fixed, and the clone request is fixed to one. Hence, for each of these executions, our approach might take a constant amount of time $k$. Hence, our approach, after running all the $|Req|$ executions will take $|Req| * k$ time. □

Furthermore, from here it is easy to see that the output of the approach is also linear:

**Theorem 7.** *Given some constraints $C$, a database $D$ s.t. $D \models C$, and a clone request $Req$, the output of our approach is linear w.r.t. the size of $Req$.*

**Proof.** The lower bound is given by the fact that each request encompasses its own clone, and the upper bound is given by the fact that the approach runs in linear time. □

We suggest to use a non-deterministic chase. That is, typically, when a chase execution finds a DED, it first explores if there is a solution in the first disjunction (generating a chase-branch) and, afterwards, if it does not find any, explores the next disjunction (generating a new chase-branch). In our approach, each chase-branch always has a solution, that is, our chase will never backtrack. Thus, it is better if the chase randomly selects the DED disjunction to apply. In this way, the resulting database will be more random, which is a desirable property for testing. Formally:

**Theorem 8.** *Any chase-branch of our approach always yields a solution.*

**Proof.** It has been already shown that our approach always finds a solution (provided that the chase is complete), we know show that such solution can be found in any chase-branch. In the first and third steps, cloning merging, there is only one chase-branch (since there are no DEDs), hence, there is always a solution. In the second step, computing unmergeables, there are only TGDs and DEDs (no EGDs nor denials involved). However, each TGD and DED has, on the RHS, *unmergeable* atoms which do not appear in the LHS of any other rule. Hence, clearly, executing such TGDs/EGDs will eventually terminate (there will be no infinite solution), and no branch will be ever cut (there is no denial nor EGD that can cut the chase-branch). □

As a final comment, we also argue that the merging rule, from the merging final stage, could be only applied with some probability. Indeed, if we always merge all the possible clones, we might end-up with the smallest database satisfying the clone requests and the constraints. This might not be desirable for

testing purposes, where more random casuistics are required. Hence, we argue that this final step should be applied in a non-deterministic manner. Alternatively, we can also add, at random, more *unmergeable* facts to obtain the same behaviour.

## 5. Experiments

In this section, we evaluate the practical applicability of our approach. To that end, we conduct two independent experiments. On the one hand, the objective of the first experiment is to compare our method against a baseline solution and show its superiority, by evaluating the evolution of the response time. To that end, we perform the evaluation using the running example introduced earlier, which serves as representative scenario of a real-world application. Then, on the other hand, the objective of the second experiment is to scrutinize in detail how the different parameters of our cloning method affect its performance. Thus, here we assess the performance on a larger variety of cases, including corner cases which are rarely found in practice. To achieve that we generate synthetic schemata and sets of constraints using a state-of-the-art graph instance and query workload generator. All details and reproducibility instructions can be found in [17].

### 5.1. Comparison against a baseline

**Experimental setting.** As previously discussed, there does not exist in the literature a solution for large-scale test data generation that considers general business rules. Hence, we here compare our method against SVTe [18], a baseline SAT solver solution. This is a tool that, if a database schema is correct (i.e., with no contradictions), generates a minimum set of instances satisfying all constraints. As input database schema and set of business rules we use the running example introduced in Fig. 1. To compare the methods, we systematically generate an increasing amount of instances of each class. Note that, due to the presence of business rules and cardinality constraints present in the schema, the generation of an instance of a particular class will trigger the generation of instances of other classes (e.g., to generate a message, the tool must generate a conversation in which it takes place, and also a user, which must belong to the conversation, to play the role of message sender).

For each class and number of instances ($N$), we evaluate our approach and measure the execution time in seconds ($T$). To account for variability, we evaluate the methods under the same conditions three times and consider the median values. Experiments were performed on a single-threaded Java program running on Windows 10 machine with an Intel(R) Core(TM) i7-10700K processor and with 16 GB of RAM.

**Experimental results.** In Fig. 4, we present the runtime performance comparison of the baseline solution and our method. To ease the interpretation of results, the curves generated by the collected data points have been smoothed by natural cubic splines. In general, we can clearly observe that the baseline method is limited by its NP-hard complexity, which causes the runtime to exponentially explode even for a low number of instances to be generated. Oppositely, the trend presented by our cloning approach is clearly linear, as theoretically proven in Section 4. Such differences are most present for those classes that are affected by a higher number of business rules (e.g., *Message* and *Member*), as the number of backtracking steps in the SAT solver grows exponentially. Oppositely, for those classes that are neither covered by business rules nor cardinality constraints (e.g., *User*) our approach and the baseline solution perform likewise, both in the range of subseconds.

**Table 1**
Average values of minimum cardinality constraints, maximum cardinality constraints and business rules for an increasing number of classes.

| Classes | Min. card. | Max. card. | Business rules |
|---|---|---|---|
| 10 | 1.69 | 1.60 | 1.23 |
| 20 | 5.02 | 5.33 | 3.29 |
| 30 | 7.63 | 7.53 | 4.82 |
| 40 | 11.65 | 12.55 | 6.76 |

**Table 2**
Summary of variables and measures used to scrutinize our method.

| | |
|---|---|
| **Variables** | |
| $S$ | Number of tables in the schema |
| $P_c$ | Probability to a select a cardinality constraint |
| $P_m$ | Probability of merging |
| $N$ | Number of instances to clone |
| **Measures** | |
| $R$ | Number of restrictions in the schema |
| $C_{pre}$ | Number of cloned instances before merging |
| $C_{post}$ | Number of cloned instances after merging |
| $T$ | Execution time in seconds |

### 5.2. Scrutinizing our method

**Experimental setting.** To systematically assess our approach, we generate artificial scenarios adhering to certain restrictions. To that end, we used gMark [19], a graph instance and workload generator, whose output we interpreted as database schemas. Precisely, for an increasingly varying number of nodes, we generated graphs following a Gaussian distribution ($\mu = 3$ and $\sigma = 1$) consisting of two kinds of nodes (i.e., *classes* and *attributes*) and two kinds of edges (i.e., attribute membership into a class, and references between attributes of different classes). Such graph is translated to an equivalent relational schema where we consider one-to-one and one-to-many cardinalities. For business rules, we consider all possible non-overlapping cyclic subgraphs, which allow to express `NOT EXISTS` constraints such as those presented in the running example in Fig. 1. Table 1, depicts the statistics in terms of average values for minimum and maximum cardinality constraints, as well as for business rules, on synthetically generated schemata with a number of classes in the range 10..40. Finally, in order to materialize instances in the artificial schema that adheres to all constraints, we, again, use SVTe. Here, SVTe is used to generate the minimal set of instances that satisfy all constraints and serve as seed for the cloning phase.

We consider each scenario to consist of a relational schema, a set of cardinality constraints and business rules, and a minimum set of instances that satisfy all constraints. With such structure, we evaluate our cloning and merging approach. Hence, to evaluate our approach under different situations, we customize the generation of artificial scenarios using the following variables: (1) the number of tables in the resulting schema ($S$); (2) the probability to select each cardinality constraint ($P_c$); (3) the probability for a cloned instance (and candidate to be merged) to be actually merged ($P_m$); and (4) the number of instances to clone in a single execution ($N$). For each combination of variables, we evaluate our approach and measure the number of restrictions in the schema ($R$), the number of cloned instances before merging ($C_{pre}$) and after merging ($C_{post}$), and the execution time in seconds ($T$). Table 2, provides an overview of the experimental variables and measures. To account for variability, we also generate three scenarios under the same conditions and consider the median values in order to dismiss the presence of outliers among different executions.

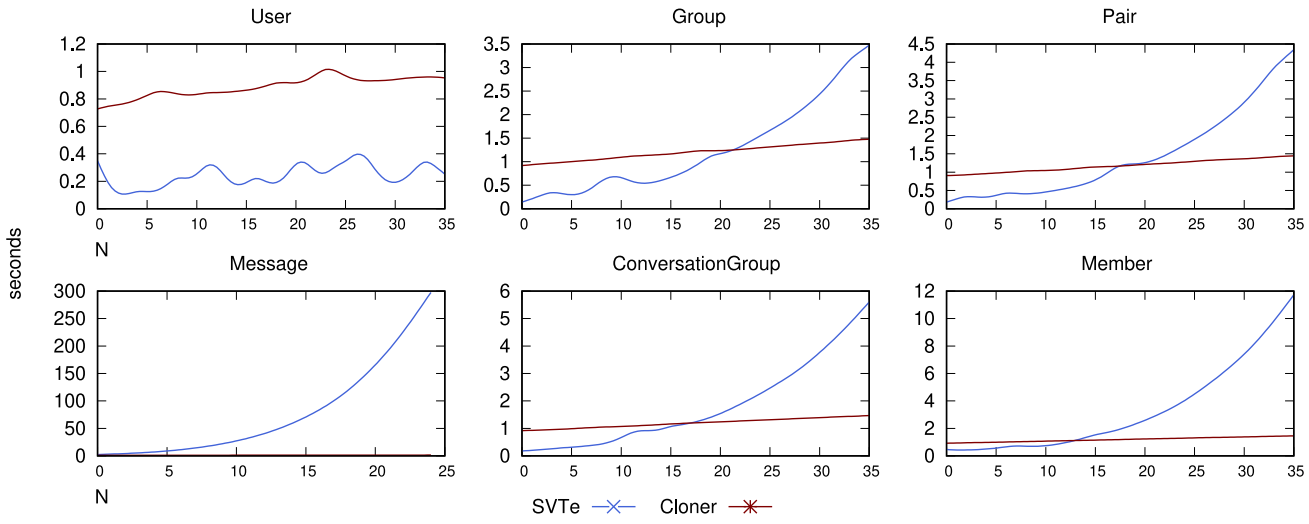



**Fig. 4.** Evolution of $T$ (y-axis) w.r.t. $N$ (x-axis) and class.

| | $S$ | $P_c$ | $P_m$ | $N$ | $R$ | $C_{pre}$ | $C_{post}$ | $T$ |
|---|---|---|---|---|---|---|---|---|
| $S$ | | | | | | | | |
| $P_c$ | 0.00 | | | | | | | |
| $P_m$ | 0.00 | 0.00 | | | | | | |
| $N$ | 0.00 | 0.00 | 0.00 | | | | | |
| $R$ | 0.84 | 0.35 | 0.01 | 0.00 | | | | |
| $C_{pre}$ | 0.70 | 0.19 | 0.00 | 0.35 | 0.78 | | | |
| $C_{post}$ | 0.69 | 0.19 | −0.03 | 0.35 | 0.78 | 1.00 | | |
| $T$ | 0.55 | 0.20 | 0.00 | 0.19 | 0.70 | 0.88 | 0.88 | |

**Fig. 5.** Correlation matrix for experimental variables. Darker denotes higher values.

**Correlation among measured variables.** We, first, analyse how correlated are the measured variables, which serve as indicator on which ones to put the focus of analysis and provide information on what variables have the highest impact on the measurements. As depicted in Fig. 5, both measures for number of instances (pre and post merge) are highly correlated with the execution time of the algorithm. Hence, it is only necessary to report one of them. We, additionally, see that the number of restrictions is highly correlated with the execution time. This is an expected result, given that data maintenance is considerably harder when the number of constraints grows.

**Evolution of execution time.** We, next, analyse how $T$ evolves with increasing the size of the schema and cloned instances. To that end, we plot in a logarithmic scale, its evolution for different values of $N$ and $S$. As depicted in Fig. 6, we can conclude that the execution time is increasingly complex for larger schemata (which also entails more restrictions and more complexity). Yet, the complexity of cloning linearly grows with the number of cloned instances. This result has been obtained executing the pipeline on a single-thread program. Thus, this results could be improved by parallelizing the process.

**Impact of schema and constraints.** Finally, here, we analyse what is the impact of the size of the schema $S$ and number of constraints over the time $T$. Recall that, as previously described, for a given schema size we generate all possible minimum and maximum cardinality constraints, as well as business rules, expressed as cyclic subgraphs. Thus, it suffices to studying the evolution of $T$ on a range of values of $S$ to also determine the impact of constraints. Fig. 7, depicts in a logarithmic scale the evolution of $T$ for different values of $S$ and $N$. Note this is an alternative perspective of the results presented in Fig. 6, here swapping $S$ and $N$. As we earlier determined, the probability of merging $P_m$ has no impact on the response time, thus in Fig. 7 we use a constant value of $P_m = 20\%$. We can conclude that, oppositely as earlier where we showed the linear scalability of our approach over an increasing number of cloned instances, growing the size of the schema and constraints has an exponential blowup on the response time. This result is just a natural consequence of the exponential gap between data complexity (the complexity of dealing with constraints, increasing the data), and expressive complexity (the complexity of dealing with constraints, when increasing the constraints), where the expressive complexity tends to be exponentially bigger than the data complexity [20].

## 6. Related work

We review related work regarding relational data generators, automated reasoners, chase implementations and other test data generators.

**Relational data generators.** Generating relational test data considering constraints have been done before using randomized or guided search techniques. For instance, Bruno and Chaudhuri [21] propose a Data Generation Language (DGL), a language that helps the user create relational data through its built-in operations. This approach is good for creating test data satisfying some particular statistical distributions, but the satisfaction of the constraints relies on the capability of the user to build a program in DGL that satisfies them. In contrast, Houkjaer et al. [22] made a more automatic approach where the user can generate test data by specifying properties, such as data distributions and association cardinalities, rather than implementing functions. This approach can deal with primary key, foreign key and check constraints, but cannot deal with more generic constraints as we do. A more generic approach can be found in [23], which is based on writing all constraints as cardinality constraints and apply a linear programming technique. However, the optimizations they apply to make feasible the data generations comes at the cost of introducing some errors, i.e., they do not guarantee the satisfaction of all the constraints for all the data.

**Automated reasoners.** The problem of generating test data satisfying constraint can be solved through automated reasoners.

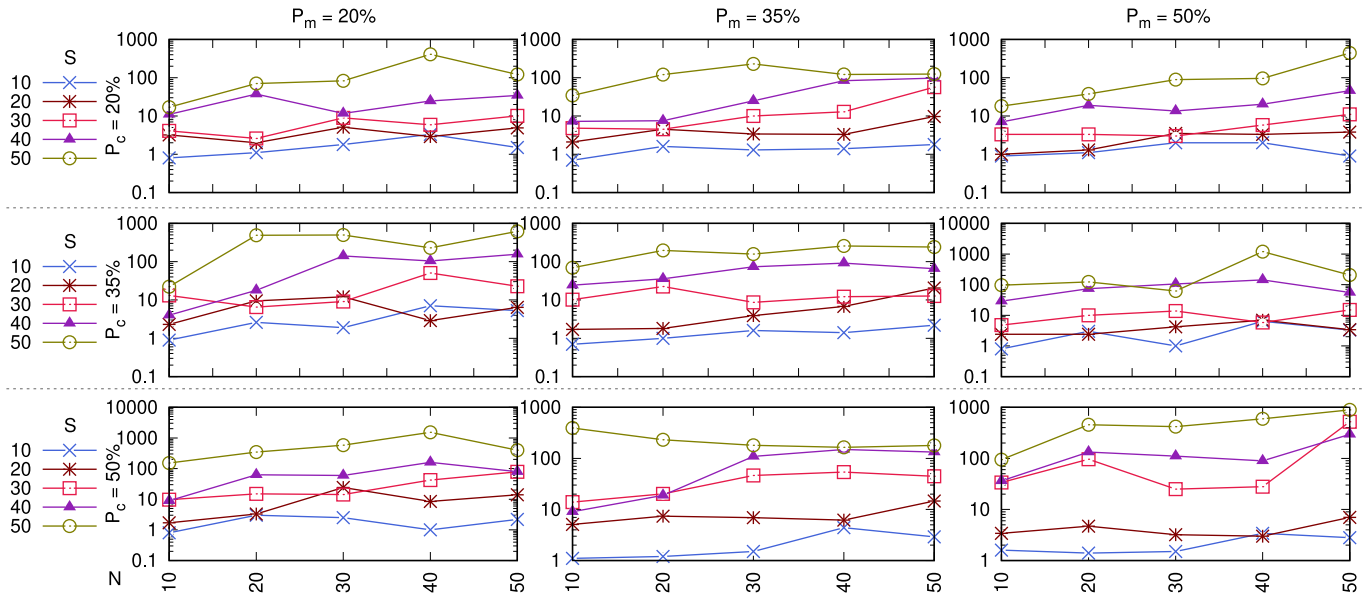



**Fig. 6.** Evolution of $T$ (y-axis) w.r.t. $N$ (x-axis) and $S$ (legend).

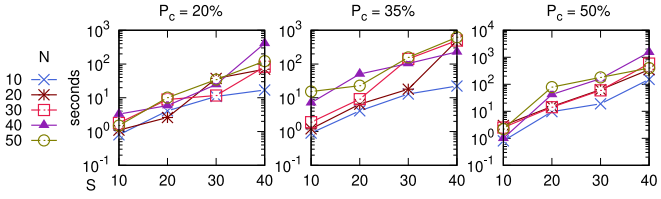



**Fig. 7.** Evolution of $T$ (y-axis) w.r.t. $S$ (x-axis) and $N$ (legend).

Automated reasoners are first-order model finders capable of building a finite database satisfying all the given constraints. Examples of automated reasoners are [9–11,24]. The most important point of such reasoners is that they are capable of dealing with more expressive constraints than we do in our approach. However, the counterpart they have is that their execution times are much higher. Thus, it is almost impossible to generate test data with a hundred instances, as we do. Some automated reasoners try to combine automated reasoning with heuristics [25], however, we still argue that they still have the trade-off between efficiency versus completeness.

**Chase implementations.** It can be argued that, since our approach defines the constraints under the form of TGD/DEDs/EGDs and denials, a normal chase, such as [26,27], can be used to generate such test data. However, we argue that our approach is faster. Indeed, in our case, our chase never performs backtracking, whereas a typical chase implementation, under the presence of DEDs and denials, must create several chase-branches, cut the branches according to the denials, and perform, in essence, a blind search. We avoid the blind search by means of repairing the constraints in the way the original database did, thus, ensuring that we always know where the solution exists.

**Other test data generators.** The authors of [12] presented a method for generating test data with the aim to check the correctness of the SQL schema defined, and in particular, with regards to the constraints defined. In contraposition to our work, this approach can only deal with typical SQL constraints (not null, unique, primary key, foreign key constraints, etc.) and is not though for generating data of a predefined size, as we do in our approach. Similarly, the authors of [13] presented an approach for, given a SQL query, generating some test data for it. Their idea focuses on generating test data that is relevant for the fixed query, and its method is based, on its core, in the Alloy reasoner [10]. Our approach can be seen as a complement for theirs. Indeed, we are able to pick their initial test data generated, and create bigger databases satisfying their constraints. The very same authors extended this work to make the computation incremental [28], however, the core is still based on a blind search algorithm (CSP), which we argue might not be efficient enough. To avoid such blind searches, the work presented in [29] advocates to use guided searches such as genetic algorithms. The problem we argue that exists in such approach is that this kind of searches are not complete, hence, they might fail to find a database satisfying all the constraints when it really exists. In contrast, our approach guarantees to always be able to create a new bigger database state. There are also other data generators that, however, only works with a fixed schema. This is the case, for instance, of the LUBM benchmark or the Berlin SPARQL benchmark [30,31], which have an ad-hoc implementation to build some data that satisfies their constraints. We argue that our approach might also be used for generating benchmarking data with the advantage of being capable of dealing with any relational schema, limited to the constraints we can deal with, instead of a fixed one.

## 7. Conclusions and further work

We have proposed an approach to generate a large set of representative data for a database schema which satisfies also a set of business rules of the application domain. Our approach builds upon a simple sample database instance that can be generated with automated reasoners and then clones it until a large and representative set of data has been generated. Our approach is independant of the particular language chosen, provided that it is first-order, which makes it suitable for business rules specified in OWL, Datalog, SQL, or specification languages such as UML/OCL. We have shown that our approach can be run with the traditional chase procedure, proved its correctness, and even shown that it has linear time complexity w.r.t. the desired size of the final database. As further work, we would like to investigate how to permit even more merges in our approach, and deal with even more expressive business rules.

## CRediT authorship contribution statement

**Xavier Oriol:** Conceptualization, Formal analysis, Writing – original draft, Writing - review & editing. **Ernest Teniente:** Supervision, Writing – original draft, Writing – review & editing. **Marc Maynou:** Software, Validation, Investigation, Writing – original draft. **Sergi Nadal:** Validation, Investigation, Writing – original draft, Writing – review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

All the details and reproducibility instructions for the experiments can be found in the companion website https://mydisk.cs.upcedu/s/E8dxK6X9kWam3nN

## Acknowledgements

This work is partially supported by the SUDOQU project, PID2021-126436OB-C21 from MCIN/AEI, 10.13039/501100011033, FEDER, UE and by the Generalitat de Catalunya, Spain (under 2017-SGR-1749); Sergi Nadal is partly supported by the Spanish Ministerio de Ciencia e Innovación, as well as the European Union - NextGenerationEU, under project FJC2020-045809-I.

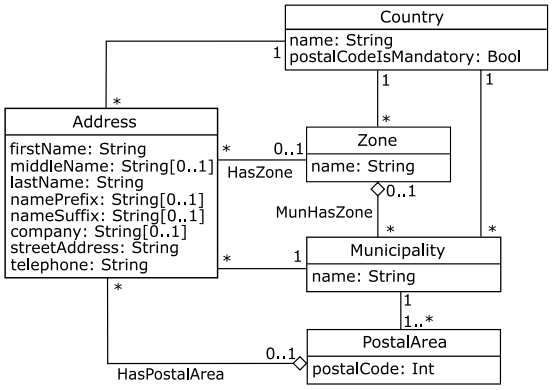
## Appendix A. The magento use case

In this appendix, we showcase the applicability of our cloning approach on a different use case presented in Fig. 1. Next, we also extend our experimental evaluation to asses the performance of our method in this new use case. We consider the domain of e-Commerce platforms, and choose Magento[2] as one of its major representative platforms. The schema of Magento is divided in two (sub)schemata, the structural and the behavioural ones. The former, contains the static knowledge, such as master data, while the latter contains the features the system offers to its users modelled per kind of use case. All code for this scenario can be found in [32].

### A.1. Schema and constraints

For the sake of simplicity, in this appendix, we focus on the *Locations* fragment of the schema as described in [33]. Such fragment contains information about a customer *Address* as depicted in Fig. A.8. Shortly, the *Locations* hierarchy specifies *PostalAreas*, which identify that geographical zone sharing a postal code. Then, a *Municipality* is an administrative entity which denotes cities, towns or villages. Finally, a *Zone* is a state or a province in a *Country*. Such schema contains also several constraints, we consider the following business rules: (1) *ZonesInAddressesSameCountry*, stating that if the address country has zones, the address is also associated to a zone of this country; (2) *PostalCodeInAddress*, stating that if the country where an address is located forces to specify a postal area, the address is associated to one postal area; and (3) *PostalAreaBelongsToMunicipality*, stating that the postal area of an address belongs to the municipality of that address. These rules are expressed as SQL assertions in Fig. A.8. Such schema can be translated to the set of relations depicted in Listing 1, which are expressed in Datalog notation.

[2] https://about.magento.com/




```
ASSERTION ZonesInAddressesSameCountry
 CHECK( NOT EXISTS(
   SELECT "Address with zone in different country"
   FROM Address a JOIN HasZone hz
     ON hz.Address = a.oID
   WHERE a.country <> hz.country  ))

ASSERTION PostalCodeInAddress
 CHECK( NOT EXISTS(
  SELECT "Postal code not in address"
  FROM Address a JOIN Country c ON a.Country = c.Name
  WHERE c.PostalCodeIsMandatory AND
  NOT EXISTS ( SELECT *
    FROM HasPostalArea hpa
    WHERE hpa.Address = a.oID ))

ASSERTION PostalAreaBelongsToMunicipality
 CHECK( NOT EXISTS(
  SELECT "Postal area not in municipality"
  FROM Address a JOIN HasPostalArea hpa ON
    a.oID = hpa.Address AND a.Country = hpa.Country
  JOIN PostalArea pa ON pa.postalCode= hpa.postalCode
    AND pa.country = hpa.country)
  WHERE pa.municipality <> hpa.municipality))
```

**Fig. A.8.** Application schema for *Locations* in Magento with constraints and business rules expressed as SQL assertions.

```
Country(Name, PostalCodeIsMandatory)
Municipality(Name, Country)
Zone(Name, Country)
PostalArea(PostalCode, Municipality, Country)
Address(oID, FirstName, MiddleName, LastName,
    NamePrefix, NameSuffix, Company, StreetAddress,
    Telephone, Fax, Country, Municipality)
MunHasZone(Municipality, Country, Zone)
HasZone(Address, Zone, Country)
HasPostalArea(Address, PostalCode, Country)
```

Listing 1: Translation of the Magento UML schema to database relations

### A.2. A cloning execution

Let us now consider the execution of our cloning pipeline. To that end, we consider the set of database instances depicted in Listing 2 as minimal set of instances to start the cloning process. This is a consistent set of instances with respect to the schema constraints. Note we use the term ._ to denote labelled nulls of attributes that are not part of any primary key or foreign key.

```
Country(c1,true)
Country(c2,false)
Country(c3,true)
Zone(z1,c1)
Zone(z2,c2)
Zone(z3,c2)
Zone(z4,c3)
Zone(z5,c3)
Municipality(m1,c1)
Municipality(m2,c1)
Municipality(m3,c2)
Municipality(m4,c3)
Municipality(m5,c3)
PostalArea(pa1,m1,c1)
PostalArea(pa2,m2,c1)
PostalArea(pa3,m4,c3)
Address(a1,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,c1,m1)
Address(a2,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,c2,m3)
Address(a3,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,c3,m5)
HasZone(a1,z1,c1)
HasZone(a2,z2,c2)
HasZone(a3,z4,c3)
HasPostalArea(a1,pa1,c1)
HasPostalArea(a2,pa2,c1)
MunHasZone(m1,c1,z1)
MunHasZone(m3,c2,z3)
```

Listing 2: Minimal database for the Magento use case

Then, from the set of instances in Listing 2, after executing our cloning pipeline we have the additional set of instances depicted in Listing 3. As it can be easily observed, these instances are also consistent with respect to the schema constraints.

```
Country(c4,true)
Country(c5,false)
Country(c6,true)
Zone(z6,c4)
Zone(z7,c5)
Municipality(m6,c4)
Municipality(m7,c4)
Municipality(m8,c5)
Municipality(m9,c5)
Municipality(m10,c6)
PostalArea(pa4,m7,c4)
PostalArea(pa5,m9,c5)
PostalArea(pa6,m10,c6)
Address(a4,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,c4,m7)
Address(a5,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,c5,m8)
Address(a6,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,c5,m9)
Address(a7,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,._,c6,m10)
HasZone(a5,z7,c5)
HasPostalArea(a4,pa4,c4)
HasPostalArea(a6,pa5,c5)
HasPostalArea(a7,pa6,c6)
MunHasZone(m6,c4,z6)
```

Listing 3: Cloning results from the minimal database

### A.3. Evaluation on the magento use case

Here, we provide a systematic evaluation of our approach on the Magento use case. Following the same approach presented in Section 5.1, we use SVTe to generate a minimal set of database instances that satisfies all schema constraints and assess the performance of our cloning method. For each class, we evaluate the time (in seconds) to clone $N$ instances from 50 to 1000 using a merge probability of 30%. As depicted in Fig. A.9, which shows the evolution of response time to clone, in such scenario our approach follows the same trend as in the previously presented experimental results showing a clear linear scalability.

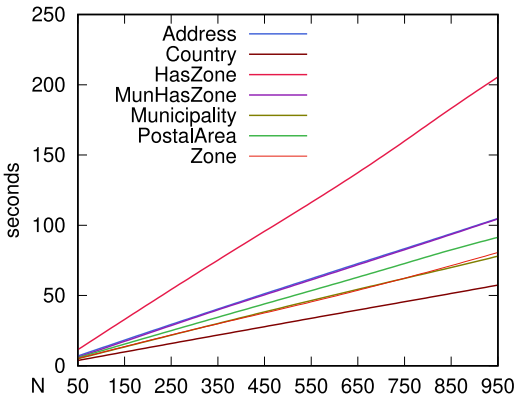



**Fig. A.9.** Evolution of the response time to clone (y-axis) w.r.t. the number of instances (x-axis) and class.

## References


[1] G.J. Myers, The Art of Software Testing, second ed., Wiley, 2004.

[2] J. Edvardsson, A survey on automatic test data generation, in: Proceedings of the 2nd Conference on Computer Science and Engineering, Citeseer, 1999, pp. 21–28.

[3] A. Poggi, D. Lembo, D. Calvanese, G.D. Giacomo, M. Lenzerini, R. Rosati, Linking data to ontologies, in: Journal on Data Semantics X, Springer, 2008, pp. 133–173.

[4] OMG, Unified modeling language (UML) superstructure, version 2.0, 2005.

[5] The SQL 92 Standard, ANSI Standard, 1992.

[6] SQL assertions - declarative multirow constraints, 2022, https://community.oracle.com/tech/apps-infra/discussion/4390732/sql-assertions-declarative-multi-row-constraints. (Accessed 08 December 2022).

[7] SQL assertion statement, 2022, https://stackoverflow.com/questions/45564667/sql-assertion-statement. (Accessed 08 December 2022).

[8] OCL-repository, 2022, https://github.com/jcabot/ocl-repository. (Accessed 08 December 2022).

[9] J. Cabot, R. Clarisó, D. Riera, UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming, in: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, 2007, pp. 547–548.

[10] E. Torlak, D. Jackson, Kodkod: A relational model finder, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2007, pp. 632–647.

[11] G. Rull, C. Farré, A. Queralt, E. Teniente, T. Urpí, AuRUS: explaining the validation of UML/OCL conceptual schemas, Softw. Syst. Model. 14 (2) (2015) 953–980.

[12] P. Mcminn, C.J. Wright, G.M. Kapfhammer, The effectiveness of test coverage criteria for relational database schema integrity constraints, ACM Trans. Softw. Eng. Methodol. (TOSEM) 25 (1) (2015) 1–49.

[13] C. De La Riva, M.J. Suárez-Cabal, J. Tuya, Constraint-based test database generation for SQL queries, in: Proceedings of the 5th Workshop on Automation of Software Test, 2010, pp. 67–74.

[14] X. Oriol, E. Teniente, G. Rull, TINTIN: a tool for incremental integrity checking of assertions in SQL server, in: Advances in Database Technology-EDBT 2016, 19th International Conference on Extending Database Technology, Bordeaux, France, March 15-16, Proceedings, 2016, pp. 632–635.

[15] X. Oriol, E. Teniente, A. Tort, Computing repairs for constraint violations in UML/OCL conceptual schemas, Data Knowl. Eng. 99 (2015) 39–58.

[16] A. Deutsch, A. Nash, J. Remmel, The chase revisited, in: Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2008, pp. 149–158.

[17] Reproducibility of cloning experiments, 2022, https://mydisk.cs.upc.edu/s/E8dxK6X9kWam3nN. (Accessed 14 December 2022).

[18] C. Farré, G. Rull, E. Teniente, T. Urpí, SVTe: a tool to validate database schemas giving explanations, in: DBTest, ACM, 2008, p. 9.

[19] G. Bagan, A. Bonifati, R. Ciucanu, G.H.L. Fletcher, A. Lemay, N. Advokaat, gMark: Schema-driven generation of graphs and queries, IEEE Trans. Knowl. Data Eng. 29 (4) (2017) 856–869.
[20] M.Y. Vardi, The complexity of relational query languages, in: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, 1982, pp. 137–146.
[21] N. Bruno, S. Chaudhuri, Flexible database generators, in: Proceedings of the 31st International Conference on Very Large Data Bases, 2005, pp. 1097–1107.
[22] K. Houkjær, K. Torp, R. Wind, Simple and realistic data generation, in: Proceedings of the 32nd International Conference on Very Large Data Bases, 2006, pp. 1243–1246.
[23] A. Arasu, R. Kaushik, J. Li, Data generation using declarative constraints, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, 2011, pp. 685–696.
[24] M. Kuhlmann, L. Hamann, M. Gogolla, Extensive validation of OCL models by integrating SAT solving into USE, in: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Springer, 2011, pp. 290–306.
[25] G. Soltana, M. Sabetzadeh, L.C. Briand, Practical constraint solving for generating system test data, ACM Trans. Softw. Eng. Methodol. (TOSEM) 29 (2) (2020) 1–48.
[26] F. Geerts, G. Mecca, P. Papotti, D. Santoro, Cleaning data with llunatic, VLDB J. (2019) 1–26.
[27] A. Bonifati, I. Ileana, M. Linardi, ChaseFUN: a data exchange engine for functional dependencies at scale, in: EDBT, 2017, pp. 534–537.
[28] M.J. Suárez-Cabal, C. de la Riva, J. Tuya, R. Blanco, Incremental test data generation for database queries, Autom. Softw. Eng. 24 (4) (2017) 719–755.
[29] J. Castelein, M. Aniche, M. Soltani, A. Panichella, A. van Deursen, Search-based test data generation for SQL queries, in: Proceedings of the 40th International Conference on Software Engineering, 2018 pp. 1220–1230.
[30] Y. Guo, Z. Pan, J. Heflin, LUBM: A benchmark for OWL knowledge base systems, J. Web Semant. 3 (2–3) (2005) 158–182.
[31] C. Bizer, A. Schultz, The berlin sparql benchmark, Int. J. Semant. Web Inform. Syst. (IJSWIS) 5 (2) (2009) 1–24.
[32] Appendix scenario, 2022, https://mydisk.cs.upc.edu/s/XpXyyy7tAy2TK2r. (Accessed 14 December 2022).
[33] A. Ramirez Papell, Esquema Conceptual De Magento, Un Sistema De Comerç Electrònic, Universitat Politècnica de Catalunya, 2011.

**Xavier Oriol** received the Ph.D. in Computer Science in 2017 from Universitat Politècnica de Catalunya (UPC) with International mention (Cum Laude). He is a researcher and project leader at inLab FIB, and teaches software engineering courses in the same faculty. His research interests include incremental integrity checking, semantics, and automated reasoning on conceptual schemas.

**Ernest Teniente** (Ph.D., UPC) is a full professor at the Department of Service and Information System Engineering at the Universitat Politècnica de Catalunya – BarcelonaTech. Current director of the inLab FIB. He has also been a visiting researcher at the Politecnico di Milano and at the Universita' di Roma Tre, in Italy. He has been active in the fields of software engineering and databases for the past 20 years. He conducts research on conceptual modelling, automated reasoning on conceptual schemas, database updating problems and data integration.

**Marc Maynou** received his B.E. degree in computer science from the Technical University of Catalonia in 2021. He is currently pursuing the M.S. degree in data science in the same university whilst also working as a software engineer and research assistant at the DTIM research group.

**Sergi Nadal** received the PhD in Computer Science in 2019 from Universitat Politècnica de Catalunya (UPC) and Université Libre de Bruxelles (ULB). He is a Juan de la Cierva Formación postdoctoral fellow and teaching assistant in the Database Technologies and Information Management (DTIM) group in UPC. His research interests lie on systems aspects of data and information management.