Active Integrity Constraints: from Theory to Implementation

Luís Cruz-Filipe[1], Michael Franz[1], Artavazd Hakhverdyan[1], Marta Ludovico[2], Isabel Nunes[2], and Peter Schneider-Kamp[1]

[1] Department of Mathematics and Computer Science
University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
lcf@imada.sdu.dk, mf@bfdata.dk, artavazd19@gmail.com,
petersk@imada.sdu.dk
[2] Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749-016 Lisboa, Portugal
marta.al.ludovico@gmail.com, in@fc.ul.pt

Abstract. The problem of database consistency relative to a set of integrity constraints has been extensively studied since the 1980s, and is still recognized as one of the most important and complex in the field. In recent years, with the proliferation of knowledge repositories (not only databases) in practical applications, there has also been an effort to develop implementations of consistency maintenance algorithms that have a solid theoretical basis.
The framework of active integrity constraints (AICs) is one example of such an effort, providing theoretical grounds for rule-based algorithms for ensuring database consistency. An AIC consists of an integrity constraint together with a specification of actions that may be taken to repair a database that does not satisfy it. Both denotational and operational semantics have been proposed for AICs.
In this paper, we describe `repAIrC`, a prototype implementation of the algorithms previously proposed targetting SQL databases, i.e., the most prolific type of databases. Using `repAIrC`, we can both validate an SQL database with respect to a given set of AICs and compute possible repairs in case the database is inconsistent; the tool is able to work with the different kinds of repairs that have been considered, and achieves optimal asymptotic complexity in their computation. It also implements strategies for parallelizing the search for repairs, which in many cases can make untractable problems become easily solvable.

1 Introduction

Databases are among the most prolific software items in today's world, being components of virtually every non-trivial software system used in practice – from mobile apps to enterprise management systems. Besides facilitating efficient storage and retrieval of data, one of the main tasks of database management systems is ensuring data integrity, i.e., guaranteeing semantic relationships between data not captured by the syntactic structure of the database hold at all times.

Typical database management systems allow the user to specify integrity constraints on the data as logical statements that are required to be satisfied at any given point in time. The typical database consistency tasks include guaranteeing that such constraints still hold after updating databases [1], and determining what repairs have to be made when the constraints are violated [13], without making any assumptions about how the inconsistencies came about. Repairing an inconsistent database is a highly complex process; also, it is widely accepted that human intervention is often necessary to choose an adequate repair [10]. That said, every progress towards automation in this field is nevertheless an important contribution, and criteria to choose among different possible repairs allow for a reasonable level of semi-automation.

In particular, the framework of active integrity constraints [5, 11] was introduced more recently with the goal of giving operational mechanisms to compute repairs of inconsistent databases. This framework has subsequently been extended to consider preferences [3] and to find "best" repairs automatically [7] and efficiently [6]. Active integrity constraints (AICs) are expressive enough to encompass the majority of integrity constraints that are typically found in practice, and they allow the definition of preferred ways to calculate repairs, through specific actions to be taken in specific inconsistent situations.

To the best of our knowledge, no real-world implementation of an AIC–enhanced database system exists today. This paper presents a prototype tool that implements the tree–based algorithms for computing repairs presented in [5, 7]. While not yet ready for productive deployment, this implementation can work successfully with virtually any database management system supporting access through SQL, and is readily extendible to other (nearly arbitrary) database management systems thanks to its modular design.

This paper is structured as follows. We summarize related work in Section 1.1. Section 2 recapitulates previous work on active integrity constraints and repair trees. Section 3 introduces our tool, `repAIrC`, and describes its implementation, focusing on the new theoretical results that were necessary to bridge the gap between theory and practice. Section 4 then discusses how parallel computation capabilities are incorporated in `repAIrC` to make the search for repairs more efficient. The discussion in these two sections is illustrated by a non-trivial running example. Section 5 summarizes our achievements and gives a brief outlook into future developments. This paper extends the work previously described in [8].

1.1 Related Work

The problem of maintaining data consistency when changing a database has been the focus of intensive research for over three decades. The survey paper [1], which extensively describes the state of the art in 1988, remains actual in its characterization of the concept of "good" update, and identification of three main change operations: insertion of new facts, deletion of existing facts, and modification of information.

Database changes can be caused by two distinct scenarios, which lead to the distinct notions of database update and database revision [10, 13]. A *database*

update occurs whenever the world changes and the database needs to be updated to reflect this fact; a *database revision* happens when new knowledge is obtained about a world that did not change. This distinction is especially relevant in deductive databases and open-world knowledge bases, where the known information is not assumed to be complete. In spite of their conceptual difference, updates and revisions can in practice be addressed by similar techniques. In particular, they both often demand changes that conflict with the integrity constraints, and the database must be repaired in order to regain consistency.

Over the years, several authors have proposed alternative approaches to the problem of how to repair an inconsistent database. One possibility is to read integrity constraints as rules that suggest possible actions to repair inconsistencies [1]; another is to express database dependencies through logic programming, namely in deductive databases [14, 16, 17]. A more algorithmic approach uses event-condition-action rules [19, 20], where actions are triggered by specific events, for which a procedural semantics has been defined. This paper focuses on the formalism of active integrity constraints [11], which will be described in more detail in the next section.

Several algorithms for computing repairs of inconsistent databases have been proposed and studied throughout the years, focusing on the different ways integrity constraints are specified and on the different types of databases under consideration [12, 14, 15, 17]. This multitude of approaches is not an accident: deciding whether an inconsistent database can be repaired is typically a Π_2^P- or co-Σ_2^P- complete problem, and there is no reason to believe in the existence of general-purpose algorithms for this problem, but one should rather focus on developing specific algorithms for particular interesting cases [10]. The formalism of active integrity constraints also establishes a hierarchy of database (weak) repairs, which can be used to define preferences among these and obtain more automation in the process [5].

2 Active integrity constraints

Active integrity constraints (AICs) were introduced in [11] and further explored in [4, 5], which define the basic concepts and prove complexity bounds for the problem of repairing inconsistent databases. These authors introduce declarative semantics for different types of repairs, obtaining their complexity results by means of a translation into revision programming. In practice, however, this does not yield algorithms that are applicable to real-life databases; for this reason, a direct operational semantics for AICs was proposed in [7], presenting database-oriented algorithms for finding repairs. The present paper describes a tool that can actually execute these algorithms in collaboration with an SQL database management system.

2.1 Syntax and Declarative Semantics

For the purpose of this work, we can view a database simply as a set of atomic formulas over a typed function-free first-order signature Σ, which we will assume

throughout to be fixed. Let $\mathcal{A}t$ be the set of closed atomic formulas over Σ. A database $\mathcal{I}$ *entails* literal L, $\mathcal{I} \models L$, if $L \in \mathcal{A}t$ and $L \in \mathcal{I}$, or if L is $\mathsf{not}\ a$ with $a \in \mathcal{A}t$ and $a \notin \mathcal{I}$.

An integrity constraint is a clause

$$L_1, \ldots, L_m \supset \bot$$

where each L_i is a literal over Σ, with intended semantics that $\forall(L_1 \wedge \ldots \wedge L_m)$ should not hold. As is usual in logic programming, we require that if L_i contains a negated variable x, then x already occurs in $L_1, \ldots, L_{i-1}$. We say that $\mathcal{I}$ *satisfies* integrity constraint r, $\mathcal{I} \models r$, if, for every instantiation θ of the variables in r, it is the case that $\mathcal{I} \not\models L\theta$ for some L in r; and $\mathcal{I}$ satisfies a set η of integrity constraints, $\mathcal{I} \models \eta$, if it satisfies each integrity constraint in η.

If $\mathcal{I} \not\models \eta$, then $\mathcal{I}$ may be updated through *update actions* of the form $+a$ and $-a$, where $a \in \mathcal{A}t$, stating that a is to be inserted in or deleted from $\mathcal{I}$, respectively. A set of update actions $\mathcal{U}$ is *consistent* if it does not contain both $+a$ and $-a$, for any $a \in \mathcal{A}t$; in this case, $\mathcal{I}$ can be updated by $\mathcal{U}$, yielding the database

$$\mathcal{U}(\mathcal{I}) = (\mathcal{I} \cup \{a \mid +a \in \mathcal{U}\}) \setminus \{a \mid -a \in \mathcal{U}\}\ .$$

The problem of database repair is to find $\mathcal{U}$ such that $\mathcal{U}(\mathcal{I}) \models \eta$.

Definition 1. *Let $\mathcal{I}$ be a database and η a set of integrity constraints. A* weak repair *for $\langle \mathcal{I}, \eta \rangle$ is a consistent set $\mathcal{U}$ of update actions such that: (i) every action in $\mathcal{U}$ changes $\mathcal{I}$; and (ii) $\mathcal{U}(\mathcal{I}) \models \eta$. A* repair *for $\langle \mathcal{I}, \eta \rangle$ is a weak repair $\mathcal{U}$ for $\langle \mathcal{I}, \eta \rangle$ that is minimal w.r.t. set inclusion.*

The distinction between weak repairs and repairs embodies the standard principle of *minimality of change* [21].

The problem of deciding whether there exists a (weak) repair for an inconsistent database is NP-complete [5]. Furthermore, simply detecting that a database is inconsistent does not give any information on how it can be repaired. In order to address this issue, those authors proposed active integrity constraints (AICs), which guide the process of selection of a repair by pairing literals with the corresponding update actions.

In the syntax of AICs, we extend the notion of update action by allowing variables. Given an action α, the literal corresponding to it is $\mathsf{lit}(\alpha)$, defined as a if $\alpha = +a$ and $\mathsf{not}\ a$ if $\alpha = -a$; conversely, the update action corresponding to a literal L, $\mathsf{ua}(L)$, is $+a$ if $L = a$ and $-a$ if $L = \mathsf{not}\ a$. The *dual* of a is $\mathsf{not}\ a$, and conversely; the dual of L is denoted L^D. An *active integrity constraint* is thus an expression r of the form

$$L_1, \ldots, L_m \supset \alpha_1 \mid \ldots \mid \alpha_k$$

where the L_i (in the *body* of r, $\mathsf{body}\,(r)$) are literals and the α_j (in the *head* of r, $\mathsf{head}\,(r)$) are update actions, such that

$$\left\{\mathsf{lit}(\alpha_1)^D, \ldots, \mathsf{lit}(\alpha_k)^D\right\} \subseteq \{L_1, \ldots, L_m\}\ .$$

The set $\mathsf{lit}(\mathsf{head}\,(r))^D$ contains the *updatable* literals of r. The *non-updatable* literals of r form the set $\mathsf{nup}(r) = \mathsf{body}\,(r) \setminus \mathsf{lit}\,(\mathsf{head}\,(r))^D$.

The natural semantics for AICs restricts the notion of weak repair.

Definition 2. *Let $\mathcal{I}$ be a database, η a set of AICs and $\mathcal{U}$ be a (weak) repair for $\langle \mathcal{I}, \eta \rangle$. Then $\mathcal{U}$ is a* founded (weak) repair *for $\langle \mathcal{I}, \eta \rangle$ if, for every action $\alpha \in \mathcal{U}$, there is a closed instance r' of $r \in \eta$ such that $\alpha \in \mathsf{head}\,(r')$ and $\mathcal{U}(\mathcal{I}) \models L$ for every $L \in \mathsf{body}\,(r') \setminus \left\{\mathsf{lit}(\alpha)^D\right\}$.*

The problem of deciding whether there exists a weak founded repair for an inconsistent database is again NP-complete, while the similar problem for founded repairs is Σ_2^P-complete. Despite their natural definition, founded repairs can include circular support for actions, which can be undesirable; this led to the introduction of justified repairs [5].

We say that a set $\mathcal{U}$ of update actions is *closed* under r if $\mathsf{nup}(r) \subseteq \mathsf{lit}(\mathcal{U})$ implies $\mathsf{head}\,(r) \cap \mathcal{U} \neq \emptyset$, and it is closed under a set η of AICs if it is closed under every closed instance of every rule in η. In particular, every founded weak repair for $\langle \mathcal{I}, \eta \rangle$ is by definition closed under η.

A closed update action $+a$ (resp. $-a$) is a *no-effect* action w.r.t. $(\mathcal{I},\mathcal{U}(\mathcal{I}))$ if $a \in \mathcal{I} \cap (\mathcal{U}(\mathcal{I}))$ (resp. $a \notin \mathcal{I} \cup (\mathcal{U}(\mathcal{I}))$). The set of all no-effect actions w.r.t. $(\mathcal{I},\mathcal{U}(\mathcal{I}))$ is denoted by $\mathsf{ne}_{\mathcal{I}}(\mathcal{U})$. A set of update actions $\mathcal{U}$ is a justified action set if it coincides with the set of update actions forced by the set of AICs and the database before and after applying $\mathcal{U}$ [5].

Definition 3. *Let $\mathcal{I}$ be a database and η a set of AICs. A consistent set $\mathcal{U}$ of update actions is a* justified action set *for $\langle \mathcal{I}, \eta \rangle$ if it is a minimal set of update actions containing $\mathsf{ne}_{\mathcal{I}}(\mathcal{U})$ and closed under η. If $\mathcal{U}$ is a justified action set for $\langle \mathcal{I}, \eta \rangle$, then $\mathcal{U} \setminus \mathsf{ne}_{\mathcal{I}}(\mathcal{U})$ is a justified weak repair for $\langle \mathcal{I}, \eta \rangle$.*

In particular, it has been shown that justified repairs are always founded [5]. The problem of deciding whether there exist justified weak repairs or justified repairs for $\langle \mathcal{I}, \eta \rangle$ is again a Σ_2^P-complete problem, becoming NP-complete if one restricts the AICs to contain only one action in their head (*normal* AICs).

2.2 Operational Semantics

The declarative semantics of AICs is not very satisfactory, as it does not capture the operational nature of rules. In particular, the quantification over all no-effect actions in the definition of justified action set poses a practical problem. Therefore, an operational semantics for AICs was proposed in [7], which we now summarize.

Definition 4. *Let $\mathcal{I}$ be a database and η be a set of AICs.*

- *The* repair tree *for $\langle \mathcal{I}, \eta \rangle$, $T_{\langle \mathcal{I}, \eta \rangle}$, is a labeled tree where: nodes are sets of update actions; each edge is labeled with a closed instance of a rule in η; the root is $\emptyset$; and for each consistent node n and closed instance r of a rule in η, if $n(\mathcal{I}) \not\models r$ then for each $L \in \mathsf{body}\,(r)$ the set $n' = n \cup \left\{\mathsf{ua}(L)^D\right\}$ is a child of n, with the edge from n to n' labeled by r.*

– *The* founded repair tree *for* $\langle \mathcal{I}, \eta \rangle$, $T^{f}_{\langle \mathcal{I},\eta \rangle}$, *is constructed as* $T_{\langle \mathcal{I},\eta \rangle}$ *but requiring that* $\mathsf{ua}(L)$ *occur in the head of some closed instance of a rule in* η.
– *The* well-founded repair tree *for* $\langle \mathcal{I}, \eta \rangle$, $T^{wf}_{\langle \mathcal{I},\eta \rangle}$, *is also constructed as* $T_{\langle \mathcal{I},\eta \rangle}$ *but requiring that* $\mathsf{ua}(L)$ *occur in the head of the rule being applied.*
– *The* justified repair tree *for* $\langle \mathcal{I}, \eta \rangle$, $T^{j}_{\langle \mathcal{I},\eta \rangle}$, *has nodes that are* pairs *of sets of update actions* $\langle \mathcal{U}, \mathcal{J} \rangle$, *with root* $\langle \emptyset, \emptyset \rangle$. *For each node* n *and closed instance* r *of a rule in* η, *if* $\mathcal{U}_n(\mathcal{I}) \not\models r$, *then for each* $\alpha \in \mathsf{head}\,(r)$ *there is a descendant* n' *of* n, *with the edge from* n *to* n' *labeled by* r, *where:* $\mathcal{U}_{n'} = \mathcal{U}_n \cup \{\alpha\}$; *and* $\mathcal{J}_{n'} = (\mathcal{J}_n \cup \{\mathsf{ua}(\mathsf{nup}(r))\}) \setminus \mathcal{U}_n$.

The properties of repair trees are summarized in the following results, whose detailed proofs can be found in [7].

Theorem 1. *Let* $\mathcal{I}$ *be a database and* η *be a set of AICs. Then:*

1. $T_{\langle \mathcal{I},\eta \rangle}$ *is finite.*
2. *Every consistent leaf of* $T_{\langle \mathcal{I},\eta \rangle}$ *is labeled by a weak repair for* $\langle \mathcal{I}, \eta \rangle$.
3. *If* $\mathcal{U}$ *is a repair for* $\langle \mathcal{I}, \eta \rangle$, *then there is a branch of* $T_{\langle \mathcal{I},\eta \rangle}$ *ending with a leaf labeled by* $\mathcal{U}$.
4. *If* $\mathcal{U}$ *is a founded repair for* $\langle \mathcal{I}, \eta \rangle$, *then there is a branch of* $T^{f}_{\langle \mathcal{I},\eta \rangle}$ *ending with a leaf labeled by* $\mathcal{U}$.
5. *If* $\mathcal{U}$ *is a justified repair for* $\langle \mathcal{I}, \eta \rangle$, *then there is a branch of* $T^{j}_{\langle \mathcal{I},\eta \rangle}$ *ending with a leaf labeled by* $\mathcal{U}$.
6. *If* η *is a set of normal AICs and* $\langle \mathcal{U}, \mathcal{J} \rangle$ *is a leaf of* $T^{j}_{\langle \mathcal{I},\eta \rangle}$ *with* $\mathcal{U}$ *consistent and* $\mathcal{U} \cap \mathcal{J} = \emptyset$, *then* $\mathcal{U}$ *is a justified repair for* $\langle \mathcal{I}, \eta \rangle$.

Not all leaves will correspond to repairs of the desired kind; in particular, there may be weak repairs in repair trees. Also, both $T^{f}_{\langle \mathcal{I},\eta \rangle}$ and $T^{j}_{\langle \mathcal{I},\eta \rangle}$ typically contain leaves that do not correspond to founded or justified (weak) repairs – otherwise the problem of deciding whether there exists a founded or justified weak repair for $\langle \mathcal{I}, \eta \rangle$ would be solvable in non-deterministic polynomial time. The leaves of the well-founded repair tree for $\langle \mathcal{I}, \eta \rangle$ correspond to a new type of weak repairs, called *well-founded weak repairs*, not considered in the original works on AICs.

2.3 Parallel Computation of Repairs

The computation of founded or justified repairs can be improved by dividing the set of AICs into independent sets that can be processed independently, simply merging the computed repairs at the end [6]. Here, we adapt the definitions given therein to the first-order scenario. Two sets of AICs η_1 and η_2 are independent if the same atom does not occur in a literal in the body of a closed instance of two distinct rules $r_1 \in \eta_1$ and $r_2 \in \eta_2$. If η_1 and η_2 are independent, then repairs for $\langle I, \eta_1 \cup \eta_2 \rangle$ are exactly the unions of a repair for $\langle \mathcal{I}, \eta_1 \rangle$ and $\langle \mathcal{I}, \eta_2 \rangle$.

When one considers founded, well-founded or justified repairs, this notion can be made stronger. Since those semantics use the information in the heads of the rules, rather than in the bodies, we can obtain similar results by considering

strong independence: η_1 and η_2 are strongly independent if, for each $r_1 \in \eta_1$ and $r_2 \in \eta_2$, there is no atom occurring both in the head of a closed instance of r_1 and in the body of a closed instance of r_2, or conversely.

If an atom occurs in a literal in the body of a closed instance of a rule in η_2 and in an action in the head of a closed instance of a rule in η_1, but not conversely, then we say that η_1 *precedes* η_2. Founded/justified (but not well-founded) repairs for $\eta_1 \cup \eta_2$ can be computed in a stratified way, by first repairing $\mathcal{I}$ w.r.t. η_1, and then repairing the result w.r.t. η_2.

Splitting a set of AICs into independent sets or stratifying it can be solved using standard algorithms on graphs, as we describe in Section 4.

3 The tool

The tool `repAIrC` is implemented in Java, and its simplified UML class diagram can be seen in Figure 1. Structurally, this tool can be split into four main separate components, centered on the four classes marked in bold in that figure.

- Objects of type `AIC` implement active integrity constraints.
- Implementations of interface `DB` provide the necessary tools to interact with a particular database management system; currently, we provide functionality for SQL databases supported by JDBC.
- Objects of type `RepairTree` correspond to concrete repair trees; their exact type will be the subclass corresponding to a particular kind of repairs.
- Class `RunRepairGUI` provides the graphical interface to interact with the user.

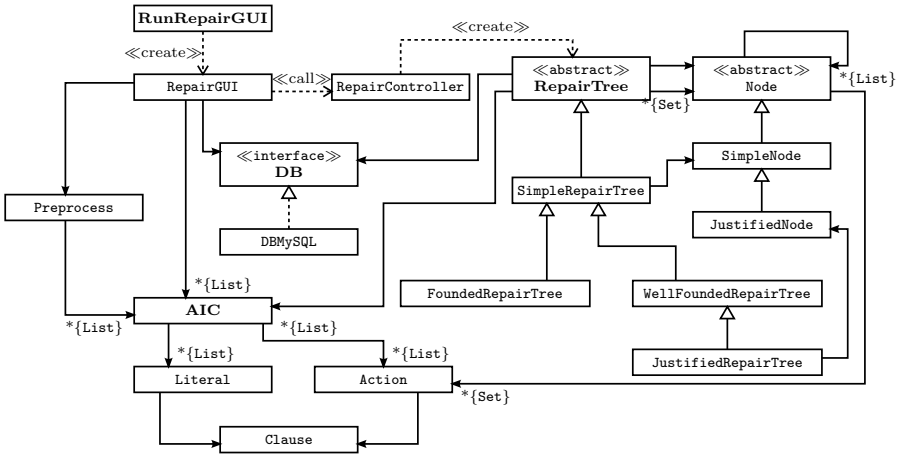

Fig. 1. Class diagram for `repAIrC`.

An important design aspect has to do with extensibility and modularity. A first prototype focused on the construction of repair trees, and used simple text

files to mimick databases as lists of propositional atoms, in the style of [5, 7]. Later, parallelization capabilities were added (as explained in Section 4), requiring changes only to `RepairController` – the class that controls the execution of the whole process. Likewise, the extension of `repAIrC` to SQL databases and the addition of the stratification mechanism only required localized changes in the classes directly concerned with those processes.

The next subsections detail the implementation of the four classes `AIC`, `DB`, `RepairTree` and `RunRepairTreeGUI`.

3.1 Representing Active Integrity Constraints

In the practical setting, it makes sense to diverge a little from the theoretical definition of AICs.

- Real-world tables found in DBs contain many columns, most of which are typically irrelevant for a given integrity constraint.
- The columns of a table are not static, i.e., columns are usually added or removed during a database's lifecycle.
- The order of columns in a table should not matter, as they are identified by a unique column name.

To deal pragmatically with these three aspects, we will use a more database-oriented notation to write down atoms, namely allowing the arguments to be provided in any order, but requiring that the column names be provided. The special token `$` is used as first character of a variable. So, for example, the literal `hasInsurance(firstName=$X, type='basic')` will match any entry in table `hasInsurance` having value `basic` in column `type` and any value in column `firstName`; this table may additionally have other columns. Negative literals are preceded by the keyword `NOT`, while actions must begin with `+` or `-`. Literals and actions are separated by commas, and the body and head of an AIC are separated by `->`. The AIC is finished when `;` is encountered, thus allowing constraints to span several lines.

AICs are provided in a text file, which is parsed by a parser generated automatically using JavaCC and transformed into objects of type `AIC`. These contain a body and a head, which are respectively `List<Literal>` and `List<Action>`; for consistency with the underlying theory, `Literal` and `Action` are implemented separately, although their objects are isomorphic: they contain an object of type `Clause` (which consists of the name of a table in the database and a list of pairs column name/value) and a flag indicating whether they are positive/negated (literals) or additions/removals (actions).

Example 1. Consider an employee database containing tables `empl` (employee), `categ` (category), `supvsr` (supervisor), `unsup` (unsupervised) and `insured`, among others. This database includes the following AICs.

$$\mathsf{empl}(X), \mathsf{empl}(Y), \mathsf{supvsr}(X,Y), \mathsf{supvsr}(Y,X) \supset -\mathsf{supvsr}(X,Y) \quad (r_1)$$
$$\mathsf{empl}(X), \mathsf{empl}(Y), \mathsf{supvsr}(X,Y), \mathsf{unsup}(Y) \supset -\mathsf{unsup}(Y) \quad (r_2)$$
$$\mathsf{empl}(X), \mathsf{cat}(X, \mathsf{junior}), \mathsf{unsup}(X) \supset -\mathsf{cat}(X, \mathsf{junior}) \quad (r_3)$$
$$\mathsf{empl}(X), \mathsf{not\ insured}(X, \mathsf{basic}) \supset +\mathsf{insured}(X, \mathsf{basic}) \quad (r_4)$$

Intuitively, (r_1) states that two employees cannot supervise each other, and the preferred way to correct this error is by changing the supvsr table. Rule (r_2) states that employees who have a supervisor are not unsupervised, and its head assumes that the information in the supvsr table is more correct. Rule (r_3) states that junior employees cannot be unsupervised, and the last rule states that all employees must have a basic insurance.

These AICs are written in the concrete text-based syntax of the `repAIrC` tool as

```
empl(id=$X), empl(id=$Y),
  supvsr(master=$X,slave=$Y), supvsr(master=$Y,slave=$X)
  -> - supvsr(master=$X,slave=$Y);

empl(id=$X), empl(id=$Y),
  supvsr(master=$X,slave=$Y), unsup(empId=$Y)
  -> - unsup(empId=$Y);

empl(id=$X), cat(type=junior, empId=$X), unsup(empId=$X)
  -> - cat(type='junior',empId=$X);

empl(id=$X), NOT insured(empId=$X, type='basic')
  -> + insured(empId=$X, type='basic');
```

respectively, assuming the corresponding column names for the attributes. Note that, thanks to our usage of explicit column naming, the order of the columns in each table is not important.

3.2 Interfacing with the Database

Database operations (queries and updates) are defined in the `DB` interface, which contains the following methods.

- `getUpdateActions(AIC aic)`: queries the database for the instances of `aic` not satisfied in its current state, and returns a `Collection<Collection<Action>>` that contains the corresponding instantiations of the head of `aic`.
- `update(Collection<Action> actions)`: applies all update actions in the collection `actions` to the database (void).
- `undo(Collection<Action> actions)`: undoes the effect of all update actions in `actions` (void).
- `aicsCompatible(Collection<AIC> aics)`: checks that all the elements of `aics` are compatible with the structure of the database.

– `disconnect()`: disconnects from the database (void). The connection is established when the object is originally constructed.

Some of these methods require more detailed comments. The construction of the repair tree also requires that the database be changed interactively, but upon conclusion the database should be returned to its original state. In theory, this would be achievable by applying the `update` method with the duals of the actions that were used to change the database; but this turns out not to be the case for deletion actions. Since the AICs may underspecify the entries in the database (because some fields are left implicit), the implementation of `update` must take care to store the values of all rows that are deleted from the database. In turn, the `undo` method will read this information every time it has to undo a deletion action, in order to find out exactly what entries to re-add.

The method `aicsCompatible` is necessary because the AICs are given independently of the database, but they must be compatible with its structure – otherwise, all queries will return errors. Including this method in the interface allows the AICs to be tested before any queries are made, thus significantly reducing the number of exceptions that can occur during program execution.

Currently, `repAIrC` includes an implementation `DBMySQL` of `DB`, which works with SQL databases. The interaction between `repAIrC` and the database is achieved by means of JDBC, a Java database connectivity technology able to interface with nearly all existing SQL databases. In order to determine whether an AIC is satisfied by a database, method `getUpdateActions` first builds a single SQL query corresponding to the body of the AIC. This method builds one `SELECT` statement for the positive literals in the body of the AIC, and another for the negative literals, if they occur. Each time a new variable is found, the table and column where it occurs are stored, so that future references to the same variable in a positive literal can be unified by using inner joins. If there is a `SELECT` statement for the negative literals, it is then connected to the other one using a `WHERE NOT EXISTS` condition. Variables in the negative literals must necessarily appear first in a positive literal in the same AIC; therefore, they can then be connected by a `WHERE` clause instead of an inner join.

Example 2. The bodies of the two last integrity constraints in Example 1 generate the following SQL queries.

```
SELECT * FROM empl t0
  INNER JOIN cat t1
  ON t0.id=t1.empId
  INNER JOIN unsup t2
  ON t0.id=t2.empId
  WHERE t1.type = 'junior'
```

```
SELECT * FROM empl t0
 WHERE NOT EXISTS
(SELECT * FROM insured t1
   WHERE t1.empId=t0.id
   AND t1.type='basic')
```

3.3 Implementing Repair Trees

The implementation of the repair trees directly follows the algorithms described in Section 2. Different types of repair trees are implemented using inheritance,

so that most of the code can be reused in the more complex trees. The trees are constructed in a breadth-first manner, and all non-contradictory leaves that are found are stored in a list. At the end, this list is pruned so that only the minimal elements (w.r.t. set inclusion) remain – as these are the ones that correspond to repairs.

While constructing the tree, the database has to be temporarily updated and restored. Indeed, to calculate the descendants of a node, we first need to evaluate all AICs at that node in order to determine which ones are violated; this requires querying a modified version of the database that takes into account the update actions in the current node.

In order to avoid concurrency issues, we use a transaction-style methodology to perform these updates, where we first change the database, then perform the necessary SQL queries, and finally rollback to the original state, guaranteeing that other threads interacting with the database during this process neither see the modifications nor lead to inconsistent repair trees. This becomes of particular interest when the parallel processing tools described in Section 4 are put into place. Although this adds some overhead to the execution time, at the end of that section we discuss why scalability is not a practically relevant concern.

After finding all the leaves of the repair tree, a further step is needed in the case one is looking for founded or justified repairs, as the corresponding trees may contain leaves that do not correspond to repairs with the desired property. This step is skipped if all AICs are normal, in view of the results from [7]. For founded repairs, we directly apply the definition: for each action α, check that there is an AIC with α in its head and such that all other literals in its body are satisfied by the database.

For justified repairs, the validation step is less obvious. Directly following the definition requires constructing the set of no-effect actions, which is essentially as large as the database, and iterating over subsets of this set. This is obviously not possible to do in practical settings. Therefore, we use some criteria to simplify this step.

Lemma 1. *If a rule r was not applied in the branch leading to $\mathcal{U}$, then $\mathcal{U}$ is closed under r.*

Proof. Suppose that r was never applied and assume $\mathsf{nup}(r) \subseteq \mathsf{ne}_{\mathcal{I}}(\mathcal{U})$. Then necessarily $\mathsf{head}\,(r) \cap \mathsf{ne}_{\mathcal{I}}(\mathcal{U}) \neq \emptyset$, otherwise r would be applicable and $\mathcal{U}$ would not be a repair.

By construction, $\mathcal{U}$ is clearly also closed for all rules applied in the branch leading to it.

Let $\mathcal{U}$ be a candidate justified weak repair. In order to test it, we need to show that $\mathcal{U} \cup \mathsf{ne}_{\mathcal{I}}(\mathcal{U})$ is a justified action set (see [7]), which requires iterating over all subsets of $\mathcal{U} \cup \mathsf{ne}_{\mathcal{I}}(\mathcal{U})$ that contain $\mathsf{ne}_{\mathcal{I}}(\mathcal{U})$. Clearly this can be achieved by iterating over subsets of $\mathcal{U}$.

But if $\mathcal{U}^* \subseteq \mathcal{U}$, then $\mathsf{nup}(r) \cap \mathcal{U}^* = \emptyset$; this allows us to simplify the closedness condition to: if $\mathsf{nup}(r) \subseteq \mathsf{ne}_{\mathcal{I}}(\mathcal{U})$, then $\mathcal{U}^* \cap \mathsf{head}\,(r) = \emptyset$. The antecedent needs

then only be done once (since it only depends on $\mathcal{U}$), whereas the consequent does not require consulting the database.

The following result summarizes these properties.

Lemma 2. *A weak repair $\mathcal{U}$ in a leaf of the justified repair tree for $\langle \mathcal{I}, \eta \rangle$ is a justified weak repair for $\langle \mathcal{I}, \eta \rangle$ iff, for every set $\mathcal{U}^* \subseteq \mathcal{U}$, if $\mathsf{nup}(r) \subseteq \mathsf{ne}_{\mathcal{I}}(\mathcal{U})$, then $\mathcal{U}^* \cap \mathsf{head}(r) = \emptyset$.*

The different implementations of repair trees use different subclasses of the abstract class `Node`; in particular, nodes of `JustifiedRepairTree`s must keep track not only of the sets of update actions being constructed, but also of the sets of non-updatable actions that were assumed. These labels are stored as `Set<Action>` using `HashSet` from the Java library as implementation, as they are repeatedly tested for membership everytime a new node is generated.

For efficiency, repair trees maintain internally a set of the sets of update actions that label nodes constructed so far as a `Set<Node>`. This is used to avoid generating duplicate nodes with the same label. Since this set is used mainly for querying, it is again implemented as a `HashSet`. Nodes with inconsistent labels are also immediately eliminated, since they can only produce inconsistent leaves.

3.4 Interfacing with the User

The user interface for `repAIrC` is implemented using the standard Java GUI widget toolkit `Swing`, and is rather straightforward. On startup, the user is presented with the dialog box depicted in Figure 2.

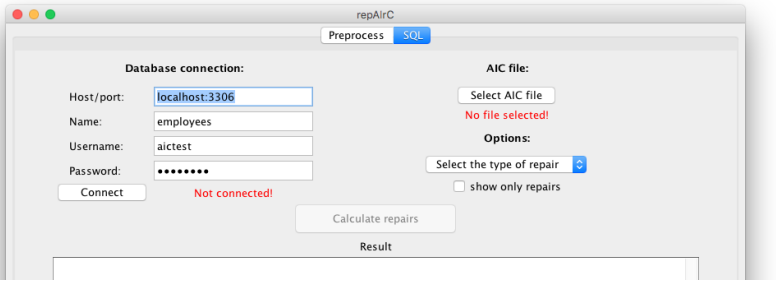

Fig. 2. The initial interface screen for `repAIrC`, providing the user with options to connect to a database, load a file with AICs, choose the desired type of repairs, and compute them.

The user can then provide credentials to connect to a database, as well as enter a file containing a set of AICs. If the connection to the database is successful and the file is successfully parsed, `repAIrC` invokes the `aicsCompatible` method required by the implementation of the `DB` interface (see Section 3.2) and verifies that all tables and columns mentioned in the set of AICs are valid tables

and columns in the database. If this is not the case, then an error message is generated and the user is required to select new files; otherwise, the buttons for configuration and computation of repairs become active.

Once the initialization has succeeded, one can check the database for consistency and obtain different types of repairs, computed using the repair tree described above. As it may be of interest to obtain also weak repairs, the user is given the possibility of selecting whether to see only the repairs computed, or all valid leaves of the repair tree – which typically include some weak repairs. In both cases the necessary validations are performed, so that leaves that do not correspond to repairs (in the case of founded or justified repairs) are never presented. A typical step in the interaction is shown in Figure 3.

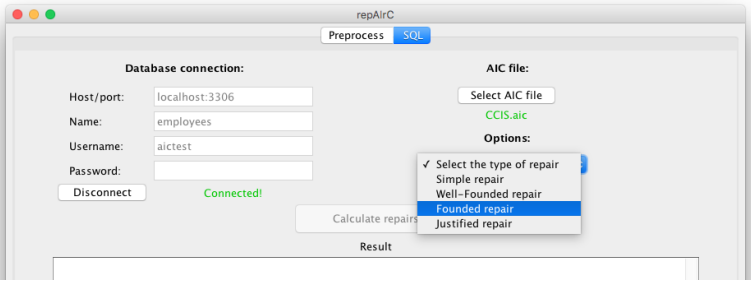

Fig. 3. The interface screen for `repAIrC` after the connection to the database has been successfully established. The drop-down menu is expanded, illustrating the different possibilities.

Example 3. We illustrate the usage of `repAIrC` with the set of AICs from Example 1, over a database in the following state.

EMPL
jane
john
mark

CAT	
jane	boss
john	clerk
mark	junior

SUPVSR	
jane	john
john	jane
john	mark

UNSUP
jane
mark

INSURED	
jane	gold
john	basic

Observe the inconsistencies in this database: john and jane mutually supervise each other; mark is both supervised by john and marked as unsupervised; he is also a junior employee marked as unsupervised. Finally, neither jane nor mark have a basic insurance.

An example output screen after successful computation of the founded repairs for this database can be seen in Figure 4. The four repairs are:

```
{+insured(empId=$jane, type='basic'),
 -supvsr(master=$john, slave=$jane),
 -unsup(empId=$mark),
 +insured(empId=$mark, type='basic')}
```

```
{+insured(empId=$jane, type='basic'),
 -cat(type='junior', empId=$mark),
 -supvsr(master=$john, slave=$jane),
 -supvsr(master=$john, slave=$mark),
 +insured(empId=$mark, type='basic')}

{+insured(empId=$jane, type='basic'),
 -supvsr(master=$jane, slave=$john),
 -unsup(empId=$jane),
 -unsup(empId=$mark),
 +insured(empId=$mark, type='basic')}

{+insured(empId=$jane, type='basic'),
 -cat(type='junior', empId=$mark),
 -supvsr(master=$jane, slave=$john),
 -supvsr(master=$john, slave=$mark),
 -unsup(empId=$jane),
 +insured(empId=$mark, type='basic')}
```

Observe that constants included in the original AICs are marked with quotes, whereas those obtained by instantiating variables are marked with a dollar sign.

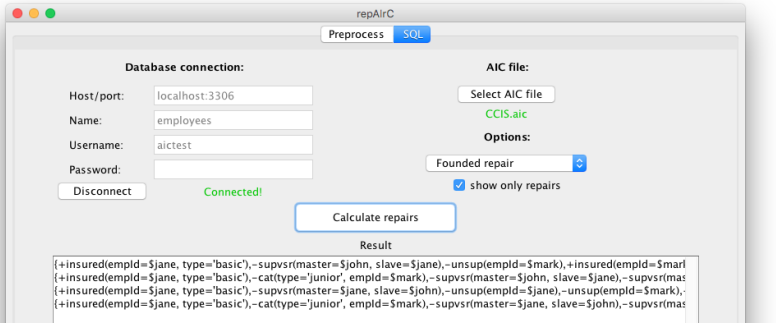

Fig. 4. Possible founded weak repairs of the inconsistent database w.r.t. the AICs from Example 1.

4 Parallelization and Stratification

As described in Section 2.3, it is possible to parallelize the search for repairs of different kinds by splitting the set of AICs into independent sets; in the case of founded or justified repairs, this parallelization can be taken one step further by also stratifying the set of AICs. Even though finding partitions and/or stratifications is asymptotically not very expensive (it can be solved in linear time by

the well-known graph algorithms described below), it may still take noticeable time if the set of AICs grows very large.

Since, by definition, partitions and stratifications are independent of the actual database, it makes sense to avoid repeating their computation unless the set of AICs changes. For this reason, parallelization capabilities are implemented in `repAIrC` in a two-stage process. Inside `repAIrC`, the user can switch to the `Preprocess` tab, which provides options for computing partitions and stratifications of a set of AICs. This results in an annotated file which still can be read by the parser; in the main tab, parallel computation is automatically enabled whenever the input file is annotated in a proper manner. Figure 5 shows the base view of the preprocessing tab.

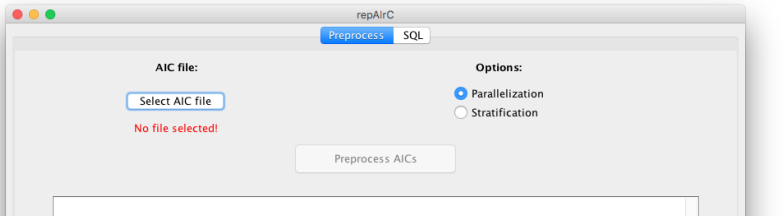

Fig. 5. Interface for computing parallelization and stratification of AICs.

4.1 Parallelization

Computing optimal partitions in the spirit of [6] is not feasible in a setting where variables are present, as this would require considering all closed instances of all AICs – but it is also not desirable, as it would also result in a significant increase of the number of queries to the database. Instead, we work with the definition of strong independency given in Section 2. Given a set of AICs, `repAIrC` constructs the adjacency matrix for the undirected graph whose nodes are AICs and such that there is an edge between r_1 to r_2 iff r_1 and r_2 are not independent. A partition is then computed simply by finding the connected components in this graph by a standard graph algorithm. The pseudo-code for the corresponding method `partition`, which takes a set of AICs and returns the set of its partition into independent subsets, is given in Figure 6.

The partitions computed can then be written to a file. Each partition begins with the line

```
#PARTITION_BEGIN_[NO]#
```

where `[NO]` is the number of the current partition, and ends with

```
#PARTITION_END#
```

and the AICs in each partition are inserted in between, in the standard format.

```
procedure partition(aics)
  visited[v] ← ⊥ (∀ v∈aics)
  partitions ← []
  foreach v ∈ aics do
    if ¬visited[v] then
      part ← []
      findCC(v,part,aics,visited)
      partitions += [part]
```

```
procedure findCC(u,part,
                 aics,visited)
  part += [u]
  visited[u] ← ⊤
  foreach v ∈ aics do
    if ¬visited[v] and u⪯v then
      findCC(v,part,aics,visited)
```

Fig. 6. Algorithm for partitioning a set of AICs into independent subsets. The test $\mathtt{u} \preceq \mathtt{v}$ is done by reading the previously computed adjacency matrix.

Example 4. The AICs in Example 1 can be split in three independent classes, which can be processed in parallel. Figure 7 shows the result of applying the parallelization algorithm to the AICs in Example 1. The three classes correspond to $\{r_1, r_2\}$, $\{r_3\}$ and $\{r_4\}$.

Computing the founded repairs of our example database using this parallelized set of AICs yields the same results as in Example 3, albeit in a different order, but takes approx. 0.43 seconds, whereas the unparallelized version required approx. 6 seconds.

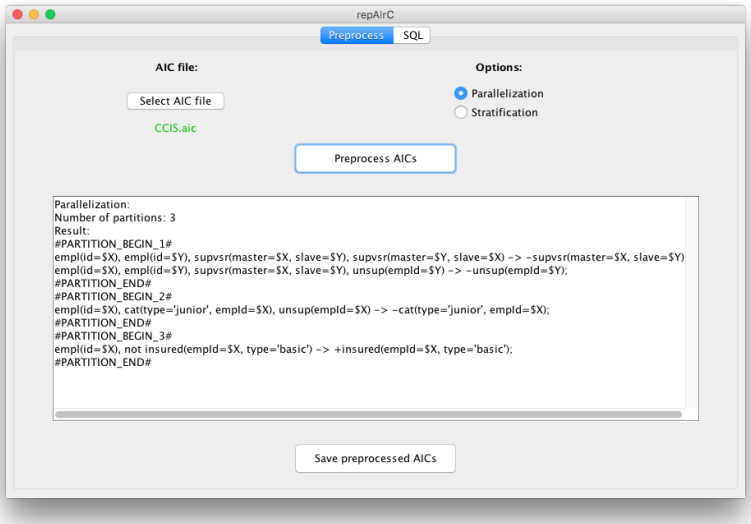

Fig. 7. Parallelization of the set of AICs from Example 1.

As pointed out in Section 2, we can only split a set of AICs into strongly independent sets if we are computing founded, well-founded or justified repairs. Therefore, if `repAIrC` is given a set of partitioned AICs and asked to compute all repairs, it will produce a warning message and ignore the parallelization.

4.2 Stratification

To compute the partitions for stratification, we need to find the strongly connected components of a similar graph. This is now a directed graph where there is an edge from r_1 to r_2 if r_1 precedes r_2. The implementation is a variant of Tarjan's algorithm [18], adapted to give also the dependencies between the connected components; the pseudo-code is given in Figure 8.

The computed stratification is then presented in a similar syntax to the previous one, to which a dependency section is added, between the special delimiters `#DEPENDENCIES_BEGIN#` and `#DEPENDENCIES_END#`, and it can again be written to a file. The dependencies are included in this section as a sequence of strings `X -> Y`, one per line, where `X` and `Y` are the numbers of two partitions and `Y` precedes `X`.

Example 5. The two AICs r_1 and r_2 in Example 1 cannot be parallelized, as was seen in Example 4, since they both use the `supvsr` table, which can be changed by r_1. They can however be stratified, as r_2 only changes `unsup`, which is not used by r_1. Preprocessing this example by `repAIrC` returns the output in Figure 9. Now each AIC is indeed in a separate set, and there is a dependency $r_1 \prec r_2$ – meaning that we can repair the database w.r.t. r_1 before considering r_2.

Computing the founded repairs of our example database using this stratified set of AICs now takes only approx. 0.07 seconds.

These examples illustrate the practical speedup obtained by splitting the set of AICs. Indeed, the independence of the several AICs in Example 1 is very clear in the repairs computed in Example 3, as they all share common parts corresponding to repairing r_3 and r_4. By processing all AICs separately we drastically reduce the size of the trees we need to build: parallelization allows us to build three trees instead of one single tree whose branches are all possible interleavings of the branches in the three (necessarily smaller) trees. Likewise, stratification again replaces one tree by two smaller ones, eliminating some interleavings of branches. In general, by stratifying AICs, we get an exponential decrease on the size of the repair trees being built – and therefore also on the total runtime.

However, as mentioned in Section 2, stratification only works when computing founded or justified repairs. If `repAIrC` is fed a stratified set of AICs and asked to compute e.g. well-founded repairs, it will warn the user that these options are incompatible and ignore the stratification.

In addition to alleviating the exponential blowup of the repair trees, parallelization and stratification also allow for a multi-threaded implementation, where repair trees are built in parallel in multiple concurrent threads. To ensure that the dependencies between the partitions are respected, the threads are instructed to wait for other threads that compute preceding partitions. In

```
procedure stratify(aics)
   visited[v] ← ⊥ (∀ v∈aics)
   lowlink ← empty map
   dependencies ← ∅
   partitions ← []
   i ← 0
   foreach v ∈ aics do
      if ¬visited[v] then
         findSCC(v,aics,i,lowlink,visited,partitions,[])
   foreach u,v ∈ aics do
      if u ≼ v and p[u] ≠ p[v] then
         dependencies ← dependencies ∪ (p[u],p[v])

procedure findSCC(v,aics,i,lowlink,visited,partitions,stack)
   visited[v] ← ⊤
   lowlink[v] ← i
   i++
   push v into stack
   isRoot ← ⊤
   foreach w ∈ aics do
      if v ≼ w and ¬visited[w] then
         findSCC(w,aics,i,lowlink,visited,partitions,stack)
      if lowlink[v] > lowlink[w] then
         lowlink[v] ← lowlink[w]
         isRoot ← ⊥
   if isRoot then
      part ← []
      do
         pop x from stack
         part += [x]
         lowlink[x] ← ∞
      while x ≠ v
      partitions += [part]
```

Fig. 8. Algorithm for stratifying a set of AICs. In procedure `stratify`, the notation `p[u]` denotes the (only) element of `partitions` containing `u`.

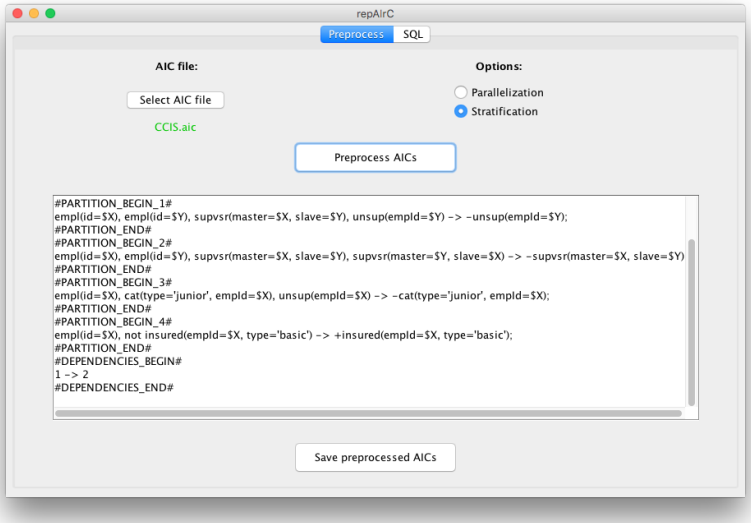

Fig. 9. Partitions computed by `repAIrC` for the AICs in Example 1.

Example 5, the thread processing partition 2 would be instructed to first wait for the thread processing partition 1 to finish.

Our example showed that significant speedups were observable even when processing small parallelizable sets of AICs. For larger sets of AICs, parallelization and stratification are necessary to obtain feasible runtimes. In one test case, which allowed for 15 partitions to be processed independently, the stratified version computed the founded repairs in approximately 1 second, whereas the sequential version did not terminate within a time limit of 15000 seconds. This corresponds to a speedup of at least four orders of magnitude, demonstrating the practical impact of the contributions of this section.

4.3 Practical Assessment

In the theoretical worst case, parallelization and stratification will have no impact on the construction of the repair tree, as it is possible to construct a set of AICs with no independent subsets. However, the worst case is not the general case, and it is reasonable to expect that real-life sets of AICs will actually have a high parallelization potential.

Indeed, integrity constraints typically reflect high-level consistency requirements of the database, which in turn capture the hierarchical nature of relational databases, where more complex relations are built from simpler ones. Thus, when specifying *active* integrity constraints there will naturally be a preference to correct inconsistencies by updating the more complex tables rather than the most primitive ones.

Furthermore, in a real setting we are not so much interested in repairing a database once, but rather in ensuring that it remains consistent as its information changes. Therefore, it is likely that inconsistencies that arise will be localized to a particular table. The ability to process independent sets of AICs separately guarantees that we will not be repeatedly evaluating those constraints that were not broken by recent changes, focusing only on the constraints that can actually become unsatisfied as we attempt to fix the inconsistency.

For the same reason, scalability of the techniques we implemented is not a relevant issue: there is no practical need to develop a tool that is able to fix hundreds of inconsistencies efficiently simultaneously, since each change to the database will likely only impact at most a few AICs.

5 Conclusions and Future Work

We described a prototype implementation of `repAIrC`, a tool to check the integrity of real-world SQL databases with respect to a given set of active integrity constraints. Furthermore, `repAIrC` implements a set of previously published algorithms to compute repairs of inconsistent databases, being able to deal with the different semantics for active integrity constraints that have been proposed so far. It can also split a set of AICs using known results on parallelization and stratification and perform parallel computations of independent repairs, thereby achieving a practical improvement that can reach several orders of magnitude. The theoretical soundness of `repAIrC` follows from results previously published in [3, 5–7, 11]. We believe that it is the first step towards an implementation of consistency maintenance features in database management systems based on a strong theoretical background.

We could go one step further in the automation process and instruct `repAIrC` to apply repairs to the database automatically. However, this does not seem a good strategy: in general, there are several possible repairs, and it has long been pointed out [10] that there will always be some instances where human intervention is necessary to sort out among the different possibilities. On the contrary, the design of `repAIrC` is such that the computation of repairs is isolated from and transparent to other concurrent uses of the database. This is accomplished by using standard SQL transaction and rollback mechanisms.

For real-world applications, the next logical step is to move beyond databases into more generic reasoning systems. There are currently several models for heterogeneous knowledge management systems, of which the framework of heterogeneous nonmonotonic multi-context systems [2] is one of the most general that have been positively received by the community. Multi-context systems are nowadays used in practice, as implementations already exist, and they are flexible enough to adapt to several different usage scenarios. A subset of the authors of this paper is currently working on defining integrity constraints for multi-context systems [9], which we plan to extend to active integrity constraints in the near future. Extending `repAIrC` to this more encompassing framework would

then be the natural next step. We believe the modularity embedded in its design will be a key ingredient towards this task.

Finally, on the more technical side, we also intend to increase `repAIrC`'s performance by means of the integration of a local database cache. In this way, `repAIrC` will be able to execute the repeated update/undo actions required during the construction of the different repair trees without interacting with the external database, thereby reducing the significant overhead introduced by that connection.

Acknowledgments. This work was supported by the Danish Council for Independent Research, Natural Sciences, and by FCT/MCTES/PIDDAC under centre grant to BioISI (Centre Reference: UID/MULTI/04046/2013). Marta Ludovico was sponsored by a grant "Bolsa Universidade de Lisboa / Fundação Amadeu Dias". The authors would also like to thank Graça Gaspar and Patrícia Engrácia for many interesting discussions on the topic of active integrity constraints.

References

1. Serge Abiteboul. Updates, a new frontier. In Marc Gyssens, Jan Paredaens, and Dirk van Gucht, editors, *ICDT*, volume 326 of *LNCS*, pages 1–18. Springer, 1988.
2. Gerhard Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *AAAI*, pages 385–390. AAAI Press, 2007.
3. Luciano Caroprese, Sergio Greco, and Cristian Molinaro. Prioritized active integrity constraints for database maintenance. In Kotagiri Ramamohanarao, P. Radha Krishna, Mukesh K. Mohania, and Ekawit Nantajeewarawat, editors, *DASFAA*, volume 4443 of *LNCS*, pages 459–471. Springer, 2007.
4. Luciano Caroprese, Sergio Greco, and Ester. Zumpano. Active integrity constraints for database consistency maintenance. *IEEE Transactions on Knowledge and Data Engineering*, 21(7):1042–1058, 2009.
5. Luciano Caroprese and Miroslaw Truszczyński. Active integrity constraints and revision programming. *Theory and Practice of Logic Programming*, 11(6):905–952, November 2011.
6. Luís Cruz-Filipe. Optimizing computation of repairs from active integrity constraints. In Christoph Beierle and Carlo Meghini, editors, *FoIKS*, volume 8367 of *LNCS*, pages 361–380. Springer, 2014.
7. Luís Cruz-Filipe, Patrícia Engrácia, Graça Gaspar, and Isabel Nunes. Computing repairs from active integrity constraints. In Hai Wang and Richard Banach, editors, *TASE*, pages 183–190. IEEE, 2013.
8. Luís Cruz-Filipe, Michael Franz, Artavazd Hakhverdyan, Marta Ludovico, Isabel Nunes, and Peter Schneider-Kamp. `repAIrC`: A tool for ensuring data consistency by means of active integrity constraints. In Ana Fred, Jan Dietz, David Aveiro, Kecheng Liu, and Joaquim Filipe, editors, *IC3K*, volume 3, pages 17–26. SCITEPRESS, November 2015.
9. Luís Cruz-Filipe, Isabel Nunes, and Peter Schneider-Kamp. Integrity constraints for general-purpose knowledge bases. In *FoIKS*. Springer, 2016. Accepted for publication.
10. Thomas. Eiter and Georg Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Artificial Intelligence*, 57(2–3):227–270, 1992.

11. Sergio Flesca, Sergio Greco, and Ester Zumpano. Active integrity constraints. In Eugenio Moggi and David Scott Warren, editors, *PPDP*, pages 98–107. ACM, 2004.
12. Antonis C. Kakas and Paolo Mancarella. Database updates through abduction. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *VLDB*, pages 650–661. Morgan Kaufmann, 1990.
13. Hirofumi Katsuno and Alberto O. Mendelzon. On the difference between updating a knowledge base and revising it. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *KR*, pages 387–394. Morgan Kaufmann, 1991.
14. V. Wiktor Marek and Miroslaw Truszczynski. Revision programming, database updates and integrity constraints. In Georg Gottlob and Moshe Y. Vardi, editors, *ICDT*, volume 893 of *LNCS*, pages 368–382. Springer, 1995.
15. Enric Mayol and Ernest Teniente. A survey of current methods for integrity constraint maintenance and view updating. In Peter P. Chen, David W. Embley, Jacques Kouloumdjian, Stephen W. Liddle, and John F. Roddick, editors, *ER (Workshops)*, volume 1727 of *LNCS*, pages 62–73. Springer, 1999.
16. Shamim A. Naqvi and Ravi Krishnamurthy. Database updates in logic programming. In Chris Edmondson-Yurkanan and Mihalis Yannakakis, editors, *PODS 1988*, pages 251–262. ACM, 1988.
17. Teodor C. Przymusinski and Hudson Turner. Update by means of inference rules. *J. Log. Program.*, 30(2):125–143, 1997.
18. Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
19. Ernest Teniente and Antoni Olivé. Updating knowledge bases while maintaining their consistency. *VLDB J.*, 4(2):193–241, 1995.
20. Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.
21. Marianne Winslett. *Updating Logical Databases*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.